MYSQL TUTORIAL

ABSTRACT

This paper demonstrates the SQL project: questions and the explained answers. That project covers a lot of aspects of SQL- Queries, subqueries, views, triggers, stored procedures and functions.

Made by: Vasif Asadov

Introduction

SQL – standard query language is the most widely used language in the data field. Therefore, it is important to have a good understanding of SQL before diving into the deep data fields. Therefore, i will apply SQL basic statements in this project, and then dive deeper into the more advanced topics, like triggers, veiws, stored procedures and etc. I will use Sakila database in this project. The database will be explained in the next section. The structure of the project is that there will be 50 questions ranging from basic topics to more advanced topics, and we should complete and solve all these questions. While solving the questions, we should give explanations about what we have done in the solution of the corresponding question. With this structure, we will be able to get hands-on experience with SQL language and be more confident in problem solving. The questions are prepared by the help of chatgpt, and solved by me.

Database information

In my project, I will use sakila database. This example dataset is not real-world dataset source, but it mimics the real-world video rental store's scenarios, as a result it is quite sufficient dataset to apply all basic, intermediate and advanced SQL queries on it. It can be downloaded in the official website of the mysql. Before applying the SQL statements, it is crucial to understand the dataset deeply, because in order to solve the advanced problems we must know dataset, the features, its statistics. Hence, let's try to understand the dataset.

Sakila dataset stores the video rental store's data. There are 13 tables in this dataset:

1. actor table – stores the information about actors in the film.
   - 'actor_id' – unique identifier for each actor (primary key)
   - 'first_name' - actor's first name
   - 'last_name' - actor's first name
   - 'last_update' – timestamp for the last update to the record

2. address table - contains address details for customers and stores.
   - `address_id`: unique identifier for each address (primary key)
   - `address`: street address.
   - `address2`: additional address information (optional).
   - `district`: district or region within the address.
   - `city_id`: reference to the `city` table (foreign key).
   - `postal_code`: postal or zip code.
   - `phone`: phone number associated with the address.
   - `last_update`: timestamp for t he last update to the record.

3. category table – defines the film categories or genres
   - `category_id`: unique identifier for each category (primary key)
   - `name`: name of the category (e.g., action, comedy).

- `last_update`: timestamp for the last update to the record.

4. city table – represents cities where adresses are located
- `city_id`: unique identifier for each city (primary key).
- `city`: name of the city.
- `country_id`: reference to the `country` table (foreign key).
- `last_update`: timestamp for the last update to the record.

5. country table – contains the country details
- `country_id`: unique identifier for each country (primary key).
- `country`: name of the country.
- `last_update`: timestamp for the last update to the record.

6. customer table – stores information about the customers
- `customer_id`: unique identifier for each customer (primary key).
- `store_id`: reference to the `store` table (foreign key).
- `first_name`: customer's first name.
- `last_name`: customer's last name.
- `email`: customer's email address.
- `address_id`: reference to the `address` table (foreign key).
- `active`: indicates if the customer account is active.
- `create_date`: date the customer was added.
- `last_update`: timestamp for the last update to the record.

7. film table - contains the details about the films
- `film_id`: unique identifier for each film (primary key).
- `title`: title of the film.
- `description`: brief description of the film.
- `release_year`: year the film was released.
- `language_id`: reference to the `language` table (foreign key).
- `original_language_id`: reference to the `language` table for the original language (nullable).
- `rental_duration`: number of days a film can be rented.
- `rental_rate`: rate at which the film is rented.
- `length`: duration of the film in minutes.
- `replacement_cost`: cost to replace the film if lost or damaged.
- `rating`: film rating (e.g., pg, r).
- `special_features`: special features of the film (e.g., trailers, behind the scenes).

- `last_update`: timestamp for the last update to the record.

8. film_actor table - junction table that manages the many-to-many relationship between films and actors.
- `actor_id`: reference to the `actor` table (foreign key).
- `film_id`: reference to the `film` table (foreign key).
- primary key: composite key of `actor_id` and `film_id`.

9. inventory table – shows which films are available in each store's inventory
- `inventory_id`: unique identifier for each inventory record (primary key).
- `film_id`: reference to the `film` table (foreign key).
- `store_id`: reference to the `store` table (foreign key).
- `last_update`: timestamp for the last update to the record.

10. language table – list languages available for the films
- `language_id`: unique identifier for each language (primary key).
- `name`: name of the language.
- `last_update`: timestamp for the last update to the record.

11. rental table – records rental transactions
- `rental_id`: unique identifier for each rental transaction (primary key).
- `rental_date`: date and time the film was rented.
- `inventory_id`: reference to the `inventory` table (foreign key).
- `customer_id`: reference to the `customer` table (foreign key).
- `return_date`: date and time the film was returned (nullable).
- `staff_id`: reference to the `staff` table (foreign key).
- `last_update`: timestamp for the last update to the record.

12. staff table – represents the employees managing the rental transactions
- `staff_id`: unique identifier for each staff member (primary key).
- `first_name`: staff member's first name.
- `last_name`: staff member's last name.
- `address_id`: reference to the `address` table (foreign key).
- `email`: staff member's email address.
- `store_id`: reference to the `store` table (foreign key).
- `active`: indicates if the staff member is currently active.

- `username`: staff member's username for login.
- `password`: staff member's password (usually stored hashed).
- `last_update`: timestamp for the last update to the record.

13. store table - represents individual stores in the rental chain.

- `store_id`: unique identifier for each store (primary key).
- `manager_staff_id`: reference to the `staff` table (foreign key) who manages the store.
- `address_id`: reference to the `address` table (foreign key).
- `last_update`: timestamp for the last update to the record.

Questions

1. Write a query to retrieve the first name, last name, and email of all customers who live in "california." (see solution)
2. List all films with their titles and rental durations where the rental duration is more than 5 days.(see solution)
3. Find all actors whose last name starts with "d." (see solution)
4. Display the total number of stores and their locations. (see solution)
5. Write a query to find the top 5 films based on their rental rate. Display the title, rental rate, and release year.(see solution)
6. List the first name, last name, and total amount spent by each customer. Sort the results by the total amount spent in descending order. (see solution)
7. Find the average rental duration of all films in the sakila database.(see solution)(see solution)
8. Retrieve the list of customers who have rented more than 10 films. Display their first name, last name, and the number of rentals. (see solution)
9. List all staff members along with the store they work at. Include the staff member's first name, last name, and the store address. (see solution)
10. Write a query to display each customer’s rental history, including the title of the film, rental date, and return date. (see solution)
11. Find the number of films available in each category. Display the category name and the number of films (see solution)
12. List customers who haven't rented any films in the last 6 months. (see solution)
13. Calculate the total revenue generated by each store. Display the store's address and the total revenue. (see solution)
14. Find the films that have a replacement cost greater than $20. Display the film title, replacement cost, and rental rate. (see solution)
15. Write a query to find the top 5 customers who have spent the most on rentals. Display their first name, last name, and the total amount spent (see solution)
16. List the top 10 most frequently rented films. Display the film title and the number of times it has been rented. (see solution)

17. Find actors who have appeared in films across more than 3 different categories. Display the actor's first name, last name, and the number of categories they have appeared in. (see solution)
18. Write a query to find the category that generates the highest revenue. Display the category name and the total revenue. (see solution)
19. Create a report that shows each customer's spending by year. Display the customer's first name, last name, year, and the total amount spent. (see solution)
20. List the top 10 films by rental rate, with their rank, in descending order. (window functions) (see solution)
21. Calculate the percentile rank of each film based on the number of times it has been rented. (window functions) (see solution)
22. Calculate the cumulative revenue generated by each store. (window functions) (see solution)
23. Rank customers based on their total payment amount. (window functions). (see solution)
24. Identify gaps in rental durations for each film. (window functions). (see solution)
25. Create a view that lists all customers who have spent more than $100 in total on rentals. (see solution)
26. Write a query that lists films rented in the last 30 days using a subquery. (see solution)
27. Create a view that lists all available films in the inventory, including the film title, category, and store location. (see solution)
28. Create a view that lists all customer first names, last names, and email addresses. This view will allow easy access to customer contact information. (see solution)
29. Create a view that displays the title and description of all films in the database. This will be useful for quickly referencing film details. (see solution)
30. Create a view that lists the first and last names of all actors in the database. This view will provide a simple way to retrieve actor information. (see solution)
31. Create a view that shows the addresses of all stores in the database, including city and postal code information. This will make it easier to retrieve store location details. (see solution)
32. Create a view that displays the title of each film along with the language it is available in. This view will help in quickly finding out which films are available in specific languages. (see solution)
33. Create a view that shows a summary of each customer's rental history. The view should include the customer's first name, last name, total number of rentals, and the total amount spent on rentals. (see solution)
34. Create a view that lists each film category along with the total number of films in that category and the average rental rate for films in that category. (see solution)
35. Create a view that provides a performance summary for each staff member. The view should include the staff member's first name, last name, total rentals processed, and the total revenue generated by that staff member. (see solution)
36. Create a view that lists all films with a rental rate above a certain threshold (e.g., $3.99). The view should include the film title, rental rate, and release year. (see solution)

37. Create a view that shows the current availability of films in the inventory. The view should include the film title, the total number of copies available, and the number of copies currently rented out. (see solution)
38. Create a view that calculates the lifetime value of each customer. The view should include the customer's first name, last name, total number of rentals, total amount spent, and the average amount spent per rental. Additionally, include the date of the customer's first rental and their most recent rental. This view will provide a comprehensive overview of customer value and behavior over time. (see solution)
39. Create a view that provides detailed statistics for each film, including the title, category, total number of times rented, total revenue generated, average rental duration, and the number of distinct customers who have rented the film. The view should also include the film's replacement cost and calculate the profitability of each film by subtracting the total rental revenue from the replacement cost. This view will offer insights into the performance and profitability of each film in the inventory. (see solution)
40. Create a view that compares the performance of each store. The view should include the store's address, total number of rentals, total revenue, the average revenue per rental, the number of distinct customers served, and the top 3 most rented films at each store. This view will allow for a side-by-side comparison of store performance and help identify trends or patterns in rental activity across different locations. (see solution)
41. Create a trigger that automatically sets a default value for the `active` column in the `customer` table to `1` (active) whenever a new customer record is inserted, if the value is not provided. This ensures that all new customers are marked as active by default. (see solution)
42. Create a trigger that automatically capitalizes the first name of a customer before it is inserted into the `customer` table. This ensures that all first names follow a consistent format. (see solution)
43. Create a trigger that logs every new customer added to the `customer` table into a `customer_log` table. The log should record the customer id, first name, last name, and the date when the record was inserted. (see solution)
44. Create a trigger that logs every time a film's rental rate is increased in the `film` table. The trigger should store the film id, old rental rate, new rental rate, and the date of the change into a `rental_rate_log` table. (see solution)
45. Create a trigger that automatically updates the `last_update` column in the `customer` table every time a customer's record is updated. This ensures that the `last_update` field always reflects the most recent change to the customer's information. Solution45
46. Create a trigger that logs deletions from the `rental` table. When a record is deleted, the trigger should insert a record into a `rental_deletions_log` table with details such as the rental id, deletion date, and the staff id who performed the deletion. (see solution)
47. Create a trigger that automatically updates the `return_date` in the `rental` table when a payment is recorded in the `payment` table for a specific rental. This trigger ensures that when a customer makes a payment, the corresponding rental is marked as returned, using the current date as the return date. (see solution)

48. Create a trigger that logs any new film inserted into the `film` table with a rental rate above a certain threshold (e.g., $4.99). The trigger should insert a record into a `high_rated_films_log` table with details such as the film title, rental rate, and insertion date whenever a film with a high rental rate is added. (see solution)
49. Create a trigger that logs any changes to a customer's email address in the `customer` table. The trigger should capture the old email, the new email, the customer id, and the date of the change, and store this information in a `customer_email_change_log` table. (see solution)
50. Create a stored procedure that takes a customer's last name as an input parameter and returns all the details of customers with that last name. This procedure should allow easy retrieval of customer information by their last name. (see solution)
51. Create a stored procedure that takes a film category name as an input parameter and returns a list of all films in that category. The procedure should include the film title, description, and rental rate. (see solution)
52. Create a stored procedure that takes a customer id and a new email address as input parameters and updates the email address of the specified customer. This procedure should help in easily updating customer contact information. (see solution)
53. Create a stored procedure that takes a store id as an input parameter and returns the total number of rentals processed by that store. This procedure will provide quick access to the rental count for any store. (see solution)
54. Create a stored procedure that takes a language id as an input parameter and returns a list of films available in that language. The procedure should include the film title and description. (see solution)
55. Create a stored procedure that takes a city name as an input parameter and returns the total number of customers living in that city. This procedure can help you quickly find out how many customers are located in a particular area. (see solution)
56. Create a stored procedure that takes a month and a year as input parameters and returns the total rental revenue for that specific month across all stores. The procedure should aggregate payments made during the specified period. (see solution)
57. Create a stored procedure that takes a customer id as an input parameter and checks the total number of rentals made by that customer. If the customer has rented more than a certain number of films (e.g., 50), update their status to a "vip" customer in a custom status column. (see solution)
58. Create a stored procedure that takes a film id as an input parameter and returns the number of available copies of that film across all stores. The procedure should also return a message indicating whether the film is available or out of stock. (see solution)
59. Create a function that takes a customer id as an input parameter and returns the full name of the customer in the format "first name last name". This function should concatenate the first and last names of the customer. (see solution)
60. Create a function that takes a category id as an input parameter and returns the total number of films available in that category. This function will provide a quick count of films within a specific category. (see solution)

s

SOLUTIONS

Basic Queries, Joins

Solution 1. write a query to retrieve the first name, last name, and email of all customers who live in "california."

- select *first_name*, *last_name* and *email* from customer table
- use address table to give the condition → district = "california"
- join customer and address tables by using *address_id* foreign key.

```
SELECT customer.first_name, customer.last_name,
customer.email FROM customer
LEFT JOIN  address ON customer.address_id = address.address_id
WHERE address.district = "California";
```

Solution 2. list all films with their titles and rental durations where the rental duration is more than 5 days.

- retrieve *title* and *rental_duration* from film table.
- add the given condition *rental_duration* > 5

```
SELECT title, rental_duration
FROM film WHERE rental_duration > 5;
```

Solution 3. Find all actors whose last name starts with "D."

```
SELECT first_name, last_name
FROM actor WHERE last_name LIKE "D%" ;
```

Solution 4. Display the total number of stores and their locations

- location is in the format of "country, districty, city". *country* from country table, district from address table and city from city table.
- use *count()* function to calculate number of stores in the store table.
- join address and stores tables on common key column – *address_id*.
- join address table with city table via *city_id*, city table with country table via *country_id*.
- classify (or *group by*) the calculations based on the country, city and address.
- order the

```
SELECT country.country,address.district, city.city,
COUNT(store.store_id) AS number_of_stores
FROM store
```

```
JOIN address ON store.address_id = address.address_id
JOIN city ON city.city_id = address.city_id
JOIN country ON country.country_id = city.country_id
GROUP BY 1,2,3;
```

Solution 5. Write a query to find the top 5 films based on their rental rate.

- Film titles and film rental rates are selected from film table.
- The results are ordered by the *rental rate* value in descending order
- Show only the top 5 films using *limit* function

```
SELECT film.title, film.rental_rate, film.release_year
FROM film
ORDER BY film.rental_rate DESC
LIMIT 5;
```

Solution 6. List the first name, last name, and total amount spent by each customer.

- Retrieve *first name*, *last name* from customer table.
- Use aggregation *sum()* function to calculate the amount from payment table.
- Group by the aggregation result based on the *customer id* to find amount per each customer.
- Make relations between customer and payment tables via common *customer_id* column.

```
SELECT customer.first_name, customer.last_name,
SUM(payment.amount) AS total_amount_spent
FROM customer
JOIN payment ON payment.customer_id = customer.customer_id
GROUP BY 1,2;
```

Solution 7. Find the average rental duration of all films in the Sakila database.

- just use ***avg()*** function to find average rental duration of all films

```
SELECT AVG(film.rental_duration) AS average_rental FROM film;
```

Solution 8. retrieve the list of customers who have rented more than 10 films. display their first name, last name, and the number of rentals.

- retrieve first name, last name from the customer table
- use *count()* function to calculate the rental id from the rental table by filtering the results according to the customer id
- make relationship between customer and rental tables via *customer_id* column.

- use *"having"* clause to add the condition to the relationship that only the customers who have total rental counts more than 10 will be retrieved.

```
SELECT
customer.first_name, customer.last_name,
COUNT(rental.rental_id) AS number_of_rentals
FROM customer
JOIN rental ON customer.customer_id = rental.customer_id
GROUP BY 1,2
HAVING number_of_rentals > 10;
```

Solution 9. list all staff members along with the store they work at. include the staff member's first name, last name, and the store address.

- retrieve the first name and last name from the staff table
- retrieve the store id from the store table
- use store_id from the staff table to make relationship between store table
- Use address id from the store table to make relationship between the address location.

```
SELECT staff.first_name, staff.last_name, address.address
FROM staff
LEFT JOIN store ON staff.store_id = store.store_id
JOIN address ON store.store_id = address.address_id;
```

Solution 10. write a query to display each customer's rental history, including the title of the film, rental date, and return date.

- retrieve the first name and last name from the customer table
- retrieve the film title from the film table
- retrieve the rental and return dates from the rental table
- use common customer_id column from the customer table to make relationship between the customer and rental tables.
- to make the relationship between the film and rental tables, firstly, we use common inventory_id to join inventory and rental tables, then by using common film_id column we achieve to make relationship between film and inventory.
  the join will be in this sequence:
  rental (*inventory_id*) → (*inventory_id*) inventory (*film_id*) →(*film_id*) film

```
SELECT customer.first_name, customer.last_name,
film.title AS film_title, rental.rental_date, rental.return_date
FROM customer
JOIN rental ON rental.customer_id = customer.customer_id
JOIN inventory ON rental.inventory_id = inventory.inventory_id
JOIN film ON inventory.film_id = film.film_id ;
```

Solution 11. find the number of films available in each category. display the category name and the number of films

- retrieve category name form the category table.
- find the count of film_id from film_category table
- filter the calculations based on the category name using group by function.

```
SELECT category.name AS category_name,
COUNT(film_category.film_id) AS Number_of_films
FROM category
JOIN film_category ON film_category.category_id =
category.category_id
GROUP BY 1
```

Solution 12. List customers who haven't rented any films in the last **6** months

- retrieve the customer first_name and last_name from the customer table.
- we should use rental_date column in order to find the date of each rental. i used here "2006-03-03 00:00:00" reference date which is used to resemble the current time. then find out the dates that are 6 months behind this reference date. timestampdiff() function can be used here. it returns the time difference between two dates by a given unit (day or month or year, that is given by ourselves)
- timestampdiff() function will find how many months passed since each rental_date. if the total number of months passed since the last rental for the specific customer is greater than 6, then we will select that customer and display it.
- additionally, there may be some customers that have not rented any films yet. rental_date values for these customes are null. hence we should consider this condition, too.
- briefly, calculate the passed months since the given defaul date grouping by the customer name and surname, filter those who have total passed months greater than 6 or those who have the rental_date values equal to null, select and display them.

```
SELECT customer.first_name, customer.last_name
FROM customer
LEFT JOIN rental ON customer.customer_id = rental.customer_id
  AND timestampdiff(month, rental.rental_date, "2006-03-03
00:00:00" ) > 6
    OR rental.rental_date IS NULL
GROUP BY 1,2;
```

Solution 13. calculate the total revenue generated by each store. display the store's address and the total revenue.

- total revenue must be calculated by applying sum() function to the amount column in the payment table. calculated results will be classified based on the store_id. therefore,

we should somehow join store and payment tables. we can join these tables in the following way:
store (store_id) → (store_id) inventory (inventory_id) → ()

store (store_id) → (store_id) inventory (inventory_id) → (inventory_id) rental (rental_id) →(rental_id) payment (amount)

- To find the address of each store, just join store and address tables with address_id.

(store_id) store (address_id) → (address _id) address (address)

```
SELECT store.store_id, address.address,
SUM(payment.amount) AS total_revenue
FROM store
JOIN address ON store.address_id = address.address_id
JOIN inventory ON inventory.store_id = store.store_id
JOIN rental ON rental.inventory_id = inventory.inventory_id
JOIN payment ON payment.rental_id = rental.rental_id
GROUP BY 1,2;
```

Solution 14. Find the films that have a replacement cost greater than $20. Display the film title, replacement cost, and rental rate

- simply, retrieve the title, replacement_cost and rental_rate from the film table, then add where clause that choose the samples that have replacement costs more than 20$.

```
SELECT film.title, film.replacement_cost, film.rental_rate
FROM film
WHERE film.replacement_cost > 20.00 ;
```

Solution 15. Write a query to find the top 5 customers who have spent the most on rentals. Display their first name, last name, and the total amount spent

- to find the top 5 customers with the highest total rentals, we need to calculate rental_id for each specific customer using the count() function, then order the retrieved results by the total rental counts in descending order. to do this, we should join these two tables with the following structure:

customer (customer_id) → (customer_id) rental (COUNT(rental_id))

- to find the total amount spent for each customer, we should calculate sum of payment amount in the payment table for each customer. joining these two tables will be done by the following structure:

customer (customer_id) → (customer_id) payment (SUM(amount))

- count and sum calculations are aggregations, therefore calculated results will be groupped by the first name and last name of each customer.

```
SELECT customer.first_name, customer.last_name,
SUM(payment.amount) AS total_amount_spent
FROM customer
JOIN payment
ON customer.customer_id = payment.customer_id
GROUP BY 1,2
ORDER BY 3 DESC;
```

Solution 16. list the top **10** most frequently rented films. display the film title **and** the number of times it has been rented.

- to display the 10 most frequently rented films, we should calculate the number of rentals for each film by using count() function, then order the results by rental counts in the descending order. then aggregate the calculations by film titles and limit the number of displayed films to 10 using limit keyword. tables will be joined in the following structure:

film (film_id) → (film_id) inventory (inventory_id) → (inventory_id) rental (count(rental_id))

```
SELECT film.title, COUNT(rental.rental_id) AS total_rentals
FROM film
JOIN inventory
  ON inventory.film_id = film.film_id
JOIN rental
  ON rental.inventory_id = inventory.inventory_id
GROUP BY 1
ORDER BY total_rentals DESC
LIMIT 10;
```

Solution 17. Find actors who have appeared in films across more than 3 different categories. Display the actor's first name, last name, and the number of categories they have appeared in.

- To find the number of categories in which actors played, we should, firstly, determine how many films did the actors play in, then group those films (that a specific actor played in) on the categories. If those films are grouped in more than 3 categories then it means the actor played in a number of films that have more than 3 category.
- To find this we need to calculate the number of categories by count() function from the film_category table. Then we should join this table to film table, then to actor table. The structure of the join statements is well-designed and more understandable below:

actor (actor_id) → (actor_id) film_actor (film_id) → (film_id) film (film_id) → (category_id) category (count(category_id))

```
SELECT actor.first_name, actor.last_name,
COUNT(film_category.category_id) AS number_of_categories
```

```
FROM actor
JOIN film_actor
  ON film_actor.actor_id = actor.actor_id
JOIN film
  ON film_actor.film_id = film.film_id
JOIN film_category
  ON film_category.film_id = film.film_id
GROUP BY 1,2
HAVING number_of_categories>3
ORDER BY number_of_categories ;
```

Solution 18. Write a query to find the category that generates the highest revenue. Display the category name and the total revenue.

- to find the category with the highest revenue, we should firslty find the amount of revenue for each category by using sum() function on the amount column in payment table; then classify the results by category names from category table, list the obtained results by revenue amount in descending order and finally apply limit function to show only the first sample which is the category with highest revenue. join structure is given below:

category (category_id) → (category_id) film_category (film_id) → (film_id) film (film_id) → (film_id) inventory (inventor_id) → (inventory_id) rental (rental_id) → (rental_id) payment (sum(amount))

```
SELECT category.name,
COUNT(payment.amount) AS total_revenue
FROM category
JOIN film_category
  ON category.category_id = film_category.category_id
JOIN film
  ON film_category.film_id = film.film_id
JOIN inventory
  ON inventory.film_id = film.film_id
JOIN rental
  ON rental.inventory_id = inventory.inventory_id
JOIN payment
  ON payment.rental_id = rental.rental_id
GROUP BY 1
ORDER BY 2 DESC
LIMIT 1;
```

Solution 19. Create a report that shows each customer's spending by year. Display the customer's first name, last name, year, and the total amount spent.

- retrieve the first_name, last_name from the customer table.
- years are defined by applying year() function to the rental dates.
- total payment is calculated by using sum() function on amount column in payment table.
- group the total payment amount by the year.

- join structrure is given below:

customer (customer_id) → (customer_id) rental (rental_id) → (rental_id) payment (sum(amount))

```
SELECT customer.first_name, customer.last_name,
YEAR(rental.rental_date) AS "year",
SUM(payment.amount) AS 'Total Amount Spent'
FROM customer
JOIN payment
  ON payment.customer_id = customer.customer_id
JOIN rental
  ON rental.rental_id = payment.rental_id
GROUP BY 1,2,3
ORDER BY 1;
```

2. Window functions in MySQL

Solution 20. List the top 10 films by rental rate, with their rank, in descending order.

- retrieve the film titles and rental rates from film table.
- to get their ranks, we should apply rank() function. rank() function will be applied to all films in the table, therefore partition by will not be used here.
- the ranks will be ordered by the rental rate in the descending order. it means the window function will start to give the rank (starting from 1) to each film based on the rental rate in descending order, means it will give the first rank (1) to the most rated film and the last rank to the film with the least rating.
- after getting the results, use limit function to show only the first 10 samples.

```
SELECT title, rental_rate,
RANK() OVER (ORDER BY rental_rate DESC) AS Rental_Rate_Rank
FROM film
ORDER BY rental_rate DESC
LIMIT 10;
```

Solution 21. Calculate the percentile rank of each film based on the number of times it has been rented.

- film titles will be retrieved from the film table.
- here percent_rank() function should be applied. because we want to find percentile rank for all films without any separation, partition by clause will not be used.
- the percent rank will get the percentile ranks for each film, and we should order them by their number of rentals.
- to find the number of rentals for each film, count() function will be applied on rental_id and the results will be grouped by film titles.

- join statements will applied on film and rental tables.

film (film_id) → (film_id) inventory (inventory_id) → (inventory_id) rental (count(rental_id))

```
SELECT film.title, COUNT(rental.rental_id) AS number_of_rentals,
PERCENT_RANK() OVER (ORDER BY COUNT(rental.rental_id))
  AS Perc_Rank_Film_Rentals
FROM film
JOIN inventory
  ON film.film_id = inventory.film_id
JOIN rental
  ON rental.inventory_id = inventory.inventory_id
GROUP BY 1;
```

Solution 22. Calculate the cumulative revenue generated by each store.

- cumulative revenue means that a new row will be added to our displayed results, and that will sum up all the previous results. as the new payments are done for the specific store, that row will add those new payment amounts, too. hence, it will be called cumulative revenue.
- cumulative revenue can be calculated by window function sum(). the function will be partitioned by the store_id, meaning that the calculations will be performed in each store separately, hence the calculations from one store will not affect anything in the another store.
- payment and store tables will be joined with the following structure:

```
SELECT store.store_id, payment.amount,
SUM(amount) OVER (PARTITION BY store.store_id ORDER BY SUM(amount)) AS
Cumulative_Revenue
FROM payment
JOIN staff
  ON payment.staff_id = staff.staff_id
JOIN store
  ON store.store_id = staff.store_id
GROUP BY 1,2;
```

Solution 23. Rank customers based on their total payment amount.

- this problem can be solved by simple query, but for the sake of understanding and practicing some intermediate statements and topics it is solved by using common table expression. common table expression (cte) is a method to make the complex queries more readable and divide the complex problems into small pieces, and then collecting the result of each piece, and make the complete statement more readable. ctes are created by using with and as statements as described in the sql code below and it creates a virtual imaginary table which includes the different columns or results of different

calculations perfomed on the columns of other tables. at the final select statements, all the calculated results and columns will be fetched from that cte (imaginary table). ctes are just statements, they are not tables that is stored in physical memory, however it is just temporary imaginary virtual storage for variables. the ctes can be joined with the original tables by using join statements.

- in this problem, total payment of the customers is calculated inside the cte and then retrieved from it and used together with the rank() function to rank the customers based on their total payment. hence while using the rank function we should apply order by to make the result list based on the total payment amount → if it is ordered by ascending order, then the customer with the smallest payment will be ranked first. if it is ordered in descending order, then the customer with the highest amount of payment will be first ranked customer.

```
WITH RankedCustomers AS (
  SELECT customer.first_name, customer.last_name,
     SUM(payment.amount) AS total_payment
     FROM customer
     JOIN payment
         ON payment.customer_id = customer.customer_id
  GROUP BY 1,2
)
SELECT first_name, last_name, total_payment,
RANK() OVER (ORDER BY total_payment DESC)  AS rank_total_payment
FROM RankedCustomers;
```

Solution 24. Identify gaps in rental durations for each film

- film titles will be retrieved from the film table.
- to identify the gaps in rental durations, firstly, one should understand what does gap mean. rental and returns date of each film is stored in the rental table. as a result film 1 is rented in march 5, 2005 and returned in march 8, 2005. then, for some days that film is not rented until the date of march 15, 2005 in which it is again rented. the days that the film is not rented is called the gaps. to find the gap, we need to know the previous returned date (march 8, 2005 in our case) and current rented date (march 15, 2005 in our case). by subtracting previous returned date from the current rental date with a unit of “day”, we can calculate how many days are there between each rental, or in other words how many days the film is not rented. sometimes film is rented by some other customer before the film is returned the previous customer. in that case, the difference between current rental date and previous return date will be negative. we should filter out these cases by adding new condition that the difference must be only pozitive.
- here the key point of the problem is to find the return date of the previous rental (or previous_rental_enddate or previous_return_date). lag() window function is used to

retrieve the value of the previous entries, therefore, in this solution, lag() function is used to get the previous rental's end date.

- after getting the end date (or return date) of the previous rental, datediff() function will be used to find the difference between the current rental date and previous return date. if the result is positive, it will be displayed as a gap of days in which the film is not rented, but if the result is negative, then it will be filtered out.

```
WITH RentalDates AS (
  SELECT film.film_id,
  rental.rental_date AS rental_start_date,
  rental.return_date AS rental_return_date
  FROM film
  JOIN inventory
        ON inventory.film_id = film.film_id
  JOIN rental ON rental.inventory_id = inventory.inventory_id
  ORDER BY 1,2
),


DatesWIthLag AS (
  SELECT film_id,
     rental_start_date,
     rental_return_date,
     LAG(rental_return_date) OVER (PARTITION BY film_id ORDER BY
rental_start_date) AS previous_rental_enddate
     FROM RentalDates
)


SELECT film_id, rental_start_date, rental_return_date,
previous_rental_enddate,
datediff(rental_start_date,previous_rental_enddate) AS gap_duration
FROM DatesWIthLag
WHERE datediff(rental_start_date,previous_rental_enddate) > 0;
```

3. Views in MySQL

Solution 25. Create a view that lists all customers who have spent more than $100 in total on rentals.

- Views store the sql statements and when it is required to apply some complex queries without re-writing it, they will be quite successful to solve that problem. Once you have written the query for any complex problem, you can log it into the views, so whenever you need it you just simply will select the name of view and the result of your query will appear.
- In this question, to list all customers who have spent more than 100$ for rentals, it is required to just simply write a query that calculates the number of rental ids form the rental table for each customer, then filter and select those whose total payment is more than 100$. After completion of the query it will be stored virtually inside the view and by using select statement the results of the query can be displayed. The joins and join

stucture will not be given in the next solutions, because it is well understood until here. By looking at the queries, it is easy to understand which tables are joined and what is the common column. Therefore, from now until the end, joins will not be discussed in this project. Only the logic behind the codes and the functions will be explained.

```
CREATE VIEW rich_customers AS
  SELECT customer.first_name,
     customer.last_name,
     SUM(payment.amount)  AS total_payment
     FROM customer
     JOIN payment
         ON customer.customer_id = payment.customer_id
  GROUP BY 1,2
     HAVING (SUM(payment.amount)  > 100)
     ORDER BY 3;
```

Solution 26. Write a query that lists films rented in the last 30 days using a subquery

- Titles of the films will be retrieved from the film table.
- Again here the date of "2006-03-03 00:00:00" will be used as a reference date.
- To find the films rented within the last 30 days, we can use datediff() function. The rental date of the film will be subtracted from the reference date, and if the result is smaller than 30, it means the rental date is within the last 30 days, otherwise the rentals are older than 30 days. this condition should be given inside the subquery. The films that meets the condition given in the subquery will be retrieved FROM this subquery.

```
CREATE VIEW rentals_last_30days AS
  SELECT film_id, title, rental_date
  FROM
        (SELECT film.film_id, film.title, rental.rental_date
        FROM film
        JOIN inventory
              ON inventory.film_id =  film.film_id
        JOIN rental
              ON rental.inventory_id = inventory.inventory_id
        ORDER BY 3 DESC) AS films

  WHERE datediff('2006-03-03 00:00:00',rental_date) < 30 ;
```

To display the created **view** use the following code:

```
SELECT * FROM rentals_last_30days;
```

Solution 27. Create a view that lists all available films in the inventory, including the film title, category, and store location

```
CREATE VIEW available_films AS
      SELECT film.title, category.name,
    CONCAT(country.country , ", " , city.city, ", ", address.address)
AS store_location
    FROM film
      JOIN film_category
            ON film_category.film_id = film.film_id
      JOIN category
            ON category.category_id = film_category.category_id
    JOIN inventory
            ON film.film_id = inventory.inventory_id
      JOIN store
            ON store.store_id = inventory.store_id
      JOIN address
            ON store.address_id = address.address_id
      JOIN city
            ON address.address_id = city.city_id
      JOIN country
            ON city.country_id = country.country_id;

SELECT * FROM available_films ;
```

Solution 28. Create a view that lists all customer first names, last names, and email addresses. This view will allow easy access to customer contact information.

- first name, last name and email address from the customer table will be retrieved.

```
CREATE VIEW customer_info AS
    SELECT customer.first_name, customer.last_name, customer.email
    FROM customer;

SELECT * FROM customer_info;
```

Solution 29. Create a view that displays the title and description of all films in the database. This will be useful for quickly referencing film details.

```
CREATE VIEW film_detail AS
      SELECT title, description
      FROM film;
SELECT * FROM film_detail;
```

Solution 30. Create a view that lists the first and last names of all actors in the database. This view will provide a simple way to retrieve actor information.

```
CREATE VIEW actor_info AS
    SELECT actor_id, first_name, last_name
```

```
    FROM actor;
SELECT * FROM actor_info;
```

Solution 31. Create a view that shows the addresses of all stores in the database, including city and postal code information. This will make it easier to retrieve store location details.

- this problem is done by simple join statements: store, address, city and country tables are joined to each other, sequentially. The resulted values will be concatenated by concat() function and the location is determined.

```
CREATE VIEW store_info AS
      SELECT
      store_id,
      CONCAT(country.country,", ", city.city, ", ",
            address.address, ", ", address.postal_code) AS address
      FROM store
    JOIN address ON store.address_id = address.address_id
      JOIN city ON city.city_id = address.city_id
      JOIN country ON city.country_id = country.country_id;

SELECT * FROM store_info;
```

Solution 32. Create a view that displays the title of each film along with the language it is available in. This view will help in quickly finding out which films are available in specific languages.

- film and language tables will be joined, then film title and language name will be retrieved from that joined tables.

```
CREATE VIEW film_languages AS
    SELECT
      language.name,
      film.title
    FROM film
    JOIN language
      ON film.language_id = language.language_id;
SELECT * FROM film_languages;
```

Solution 33. Create a view that shows a summary of each customer's rental history. The view should include the customer's first name, last name, total number of rentals, and the total amount spent on rentals.

- when multiple tables (more than two) are joined together, we can not apply more than two aggregation functions such as count(), sum(), min(), max() and etc. to them, because the join statement will cause some duplicate rows with the same name or same

id. Hence, we will get the wrong, multiplicated results. To avoid this issue, the aggregation functions will be applied separately inside the subqueries or CTEs, then at the final select statement these values will be retrieved.

- in this problem, we should calculate two things: total number of rentals and total amount spent on the rentals. The remaining required columns (first name and last name) can be easily fetched from the customer table. Therefore, main focus is performing the calculations accurately. Hence, two CTE will be used to calculate two results; Rental Calculations CTE will calculate the number of total rentals that the customer made (this problem is solved earlier) and the Payment Calculations CTE will find out total amount of payment spent by each customer. Calculation of total rentals and total payment amount is quite easy and straightforward, and has been explained in the previous solutions, hence it will not be repeated again.
- Briefly, main focus in this problem is to get two calculated results and each by separate CTE expression. Then in the final select statement the calculation results will be fetched to create the customer rental history. To log that query into the View, simply but "create view view_name as" statement above it.

```
CREATE VIEW customer_rental_history_CTE AS
WITH  Rental_Calculations AS (
      SELECT
      customer.customer_id,
      COUNT(rental.rental_id) AS Total_Rentals
      FROM customer
      JOIN rental
            ON customer.customer_id = rental.customer_id
      GROUP BY 1
),

Payment_Calculations AS (
      SELECT
      customer.customer_id,
      SUM(payment.amount) AS Total_Amount_Spent
      FROM customer
      JOIN payment
            ON customer.customer_id = payment.customer_id
      GROUP BY 1
)

SELECT customer.first_name,
      customer.last_name,
      Total_Rentals,
      Total_Amount_Spent
FROM customer
LEFT JOIN Rental_Calculations
      ON customer.customer_id = Rental_Calculations.customer_id
LEFT JOIN Payment_Calculations
      ON customer.customer_id = Payment_Calculations.customer_id
ORDER BY 4 DESC;

SELECT * FROM customer_rental_history_CTE;
```

Solution 34. Create a view that lists each film category along with the total number of films in that category and the average rental rate for films in that category.

- in this problem we should create a view which displays the categories of the film, the total number of films in each category and average rental rate for the films in each category. Category names will be taken from the category table directly, but the main consideration here is to accurately calculate two measurements: number_of_films in each category and avg_rental_rate for the films in each category. We will apply the aggregation functions on a single table – film table, we just need one CTE to perform calculations. Count() function will calculate the number of films and avg() function will calculate the average rental rates of each films and the results will be grouped based on the category name.
- At the final select statement ifnull() function can be applied to check wether the result is null or not. If the result is null, then that function will return the value inside the parantheses: ifnull(result, 0) – here if the result is null, then the function will return 0.

```
CREATE VIEW film_category_statistics AS
WITH Film_Calculation AS (
      SELECT
      category.category_id,
    COUNT(film.film_id) AS number_of_films,
    ROUND(AVG(film.rental_rate),2) AS average_rental_rate
    FROM category
      LEFT JOIN film_category
            ON film_category.category_id = category.category_id
      LEFT JOIN film
            ON film.film_id = film_category.film_id
      GROUP BY 1
)
SELECT category.name,
      number_of_films,
    average_rental_rate
FROM category
JOIN Film_Calculation
      ON category.category_id = Film_Calculation.category_id
ORDER BY number_of_films
;

SELECT * FROM film_category_statistics;


-- Simple Solution --
SELECT category.name,
    IFNULL(COUNT(film.film_id),0) AS number_of_films,
    IFNULL(AVG(film.rental_rate),0) AS average_rental_rate
    FROM category
      LEFT JOIN film_category
```

```
            ON film_category.category_id = category.category_id
      LEFT JOIN film
            ON film.film_id = film_category.film_id
      GROUP BY 1;
```

Solution 35. Create a view that provides a performance summary for each staff member. The view should include the staff member's first name, last name, total rentals processed, and the total revenue generated by that staff member.

- Two calculations required in the problem statement: total rentals and total revenue; one from rental table and another one from payment table. As is learned, in join statements, it must be avoided to perform aggregations on multiple tables. If the statement requires aggregated results from different tables, then each aggregation must be done separately, and then at final query the results must be called. Otherwise, obtained results will not be correct due to the duplicate rows.
- Therefore, in this problem it is required to perform calculations on rental table and payment tables separately, then retrieve the obtained results in the final select query. Rental_Calculation CTE will calculate how many total rentals processed by each staff by grouping the results based on the staff id.
- Revenue_Calculation CTE will calculate the total amount of payment earned by each staff by grouping the results based on staff id.
- At the final select query, the first name, last name and two calculation results – number_of_rentals and total_revenue will be retrieved. Here staff table and two CTEs must be joined by the common column staff_id.

```
CREATE VIEW staff_members_info AS
WITH Rental_Calculation AS (
      SELECT staff.staff_id,
    COUNT(rental.rental_id) AS number_of_rentals
    FROM staff
    JOIN rental
            ON rental.staff_id = staff.staff_id
      GROUP BY 1
),
-- Revenue_Calculation CTE will calculate the total_revenue earned BY
-- EACH staff
Revenue_Calculation AS (
      SELECT staff.staff_id,
    SUM(payment.amount) AS total_revenue
    FROM staff
    JOIN payment
            ON payment.staff_id = staff.staff_id
      GROUP BY 1
)
SELECT staff.first_name, staff.last_name,
    number_of_rentals,
    total_revenue
FROM staff
```

```
LEFT JOIN Rental_Calculation
      ON Rental_Calculation.staff_id = staff.staff_id
LEFT JOIN Revenue_Calculation
      ON Revenue_Calculation.staff_id = staff.staff_id;

SELECT * FROM staff_members_info;
```

Solution 36. Create a view that lists all films with a rental rate above a certain threshold (e.g., $3.99). The view should include the film title, rental rate, and release year.

- It is simple problem, Extracting the title, rental rate and release year from the film table with a condition of "rental_rate > 3.99" will solve the problem.

```
CREATE VIEW high_rental_rated_films AS
SELECT film.title, film.rental_rate, film.release_year
FROM film
      WHERE film.rental_rate > 3.99;
SELECT * FROM high_rental_rated_films;
```

Solution 37. Create a view that shows the current availability of films in the inventory. The view should include the film title, the total number of copies available, and the number of copies currently rented out.

- Two main calculations will be performed here: calculation of number_of_copies which represents the total number of all films, and calculation of the rented_copies which represents the copies that are currently rented and not returned (return_date is null)
- Calculation of total number of available films in the inventory will be performed on the inventory table, and calculation of the number of rented copies will be performed on the rental table. As we can not get the aggregation results from two tables in a single query, two CTEs are required here: Copies CTE will calculate the number of all films in the inventory, as multiple inventory ids store a single film. By grouping the number of inventories based on the film id, we can define how many films are available in the inventory.
- Rented CTE will calculate the number of films that are rented currently and not returned. It will simply count the rental id where the return_date is null, which means that the rented films are not returned
- At the final select query, film title, total number of available copies and rented copies will be retrieved and displayed.

```
CREATE VIEW available_films AS

WITH Copies_COUNT AS
(     SELECT
      film.film_id,

    COUNT(inventory_id) AS Total_Copies
    FROM film
    LEFT JOIN inventory
```

```
            ON inventory.film_id = film.film_id
      GROUP BY 1
),
Rented_COUNT AS
(     SELECT
      inventory.film_id,
      COUNT(rental_id) AS rented_copies
      FROM inventory
    LEFT JOIN rental
            ON inventory.inventory_id = rental.inventory_id
        AND rental.return_date IS NULL
    GROUP BY 1
)
SELECT film.film_id,
      IFNULL(Total_Copies,0) AS total_copies,
    IFNULL(rented_copies,0) AS rented_copies
FROM film
LEFT JOIN Copies_COUNT
      ON Copies_COUNT.film_id = film.film_id
LEFT JOIN Rented_COUNT
      ON Rented_COUNT.film_id = film.film_id
GROUP BY 1;
```

Solution 38. Create a view that calculates the lifetime value of each customer. The view should include the customer's first name, last name, total number of rentals, total amount spent, and the average amount spent per rental. Additionally, include the date of the customer's first rental and their most recent rental. This view will provide a comprehensive overview of customer value and behavior over time.

- In the problem statement 5 different aggregation (or calculations) are required: number of rentals, total amount spent, average amount spent per rental, first and last rental date of the customer. Three of these calculations will be performed on the rental table: finding out number_of_rentals, first_rental_date, last_rental_date; one will be performed on payment table: total_amount_spent; and the average_amount_spent_per_rental value will be derived in the final select query.
- Rental_Calculations CTE will calculate the number of rental_ids for each customer, also the first and last date of the customer's rental by using min() and max() operators on the rental_date column.
- Payment_Calculations CTE will find out the total amount spent by the customer. This has been explained in the previous questions.
- At the final select query, total_amount_spent value will be divided by the number_of_rentals value and that will give the average payment per rental value.
- One important note here is to check wether the calculation result is null or not. To check this don't forget to use ifnull() function, because handling the null values is quite important in data analysis.

```
CREATE VIEW customer_overview AS
WITH Rental_Calculations AS
(     SELECT
      customer.customer_id,
```

```
    COUNT(rental.rental_id) AS number_of_rentals,
    MIN(rental.rental_date) AS first_rental_date,
    MAX(rental.rental_date) AS last_rental_date
       FROM customer
    LEFT JOIN rental
              ON rental.customer_id = customer.customer_id
       GROUP BY 1
),
Payment_Calculations AS
(   SELECT
       customer.customer_id,
    SUM(payment.amount) AS total_amount_spent
    FROM customer
    LEFT JOIN payment
       ON payment.customer_id = customer.customer_id
    GROUP BY 1
)
SELECT
    customer.first_name,
    customer.last_name,
    number_of_rentals,
    IFNULL(total_amount_spent, 0) AS total_amount_spent,
    total_amount_spent/number_of_rentals AS avg_payment_per_rental,
    first_rental_date,
    last_rental_date
FROM customer
       LEFT JOIN Rental_Calculations
              ON customer.customer_id = Rental_Calculations.customer_id
       LEFT JOIN Payment_Calculations
              ON customer.customer_id = Payment_Calculations.customer_id
ORDER BY 4 DESC;

SELECT * FROM customer_overview;
```

Solution 39. Create a view that provides detailed statistics for each film, including the title, category, total number of times rented, total revenue generated, average rental duration, and the number of distinct customers who have rented the film. The view should also include the film's replacement cost and calculate the profitability of each film by subtracting the total rental revenue from the replacement cost. This view will offer insights into the performance and profitability of each film in the inventory.

- In the problem 4 extractions and 4 calculations are required: film title, category name, average rental duration and replacement cost columns should be extracted (retrieved) from the film and category tables; Then number of rentals, total revenue generated, number of distinct customers and profitability score for each film are asked to be calculated. The structure of the query will be in the following way:
- Rental_Calculations CTE will calculate the number of rentals for each film by using count() functions on rental_id.
- Payment_Calculations CTE will calculate the total amount of revenue generated by each film and the number of distinct customers who have rented the corresponding film by applying sum() function on amount column and count() function on customer_id column.

- In the final select query, the extractions are performed from the table by joining the multiple tables and the calculated results are retrieved from the CTEs. Then the profitability score is calculated on this query by subtracting the extracted replacement_cost for the corresponding film from the total_revenue generated by that film.
- Don't forget to join the CTEs to the tables by using the common columns.

```
CREATE VIEW films_detailed_statistics AS
WITH Rental_Calculations AS
(      SELECT
       film.film_id,
       COUNT(rental.rental_id) AS total_rentals
    FROM film
    LEFT JOIN inventory
              ON film.film_id = inventory.film_id
       LEFT JOIN rental
              ON inventory.inventory_id = rental.inventory_id
       GROUP BY 1
),


Payment_Calculations AS (
    SELECT
       film.film_id,
       SUM(payment.amount) AS total_revenue,
       COUNT(distinct payment.customer_id) AS number_of_customers
    FROM film
    LEFT JOIN inventory
              ON film.film_id = inventory.film_id
       LEFT JOIN rental
              ON rental.inventory_id = inventory.inventory_id
       LEFT JOIN payment
              ON rental.rental_id = payment.rental_id
       GROUP BY 1
)
SELECT
    film.film_id,
    film.title,
    category.name,
    IFNULL(Rental_Calculations.total_rentals,0) AS total_rentals,
    IFNULL(Payment_Calculations.total_revenue,0) AS total_revenue,
    film.rental_duration AS AVG_rental_duration,
    IFNULL(Payment_Calculations.number_of_customers,0) AS
number_of_customers,
    film.replacement_cost,
    IFNULL(total_revenue - film.replacement_cost,0) AS profitability
FROM film
LEFT JOIN film_category
       ON film.film_id = film_category.film_id
LEFT JOIN category
       ON film_category.category_id = category.category_id
LEFT JOIN Rental_Calculations
       ON Rental_Calculations.film_id = film.film_id
LEFT JOIN Payment_Calculations
       ON Payment_Calculations.film_id = film.film_id
GROUP BY 1,2,3,4,5,6,7,8,9;

SELECT * FROM films_detailed_statistics;
```

Solution 40. Create a view that compares the performance of each store. The view should include the store's address, total number of rentals, total revenue, the average revenue per rental, the number of distinct customers served, and the top 3 most rented films at each store. This view will allow for a side-by-side comparison of store performance and help identify trends or patterns in rental activity across different locations.

- Rental Calculations CTE will calculate the number of rentals and the number of distinct customers that purchases the rentals for each store.
- Address Finder CTE will concatenate the address, city and countries and make the address location.
- Payment Calculation CTE will calculate the revenue made by each store.
- To find the most rented 3 films row_number() window function will be used. It will rank the films based on the count of rentals.

```
CREATE VIEW store_detailed_statistics AS
WITH Rental_Calculations AS
(      SELECT
       store.store_id,
    COUNT(rental.rental_id) AS total_rentals,
    COUNT(DISTINCT rental.customer_id) AS number_of_customers
    FROM store
    LEFT JOIN inventory
        ON inventory.store_id = store.store_id
    LEFT JOIN rental
        ON rental.inventory_id = inventory.inventory_id
       GROUP BY 1
),

-- Address Finder returns: whole address
Address_Finder AS (
       SELECT
    store.store_id,
    CONCAT(country.country, ", ", address.district, ", ", city.city,
                    ", ", address.address, ", ", address.postal_code) AS
store_address
       FROM store
    JOIN address
              ON store.address_id = address.address_id
       JOIN city
              ON address.city_id = city.city_id
       JOIN country
              ON city.country_id = country.country_id
       GROUP BY 1
 ),

-- Payment_Calculation returns:  total_revenue
Payment_Calculation AS (
       SELECT
       store.store_id,
    SUM(payment.amount) AS total_revenue
    FROM store
    LEFT JOIN inventory
```

```
            ON inventory.store_id = store.store_id
      LEFT JOIN rental
            ON rental.inventory_id = inventory.inventory_id
      LEFT JOIN payment
            ON payment.rental_id = rental.rental_id
      GROUP BY 1
),

Most_Rented_3_Films AS
(     SELECT
      store.store_id,
    film.title,
    COUNT(rental.rental_id) AS rental_COUNT,
    ROW_NUMBER() OVER (PARTITION BY store.store_id ORDER BY
COUNT(rental.rental_id)  DESC) AS rank_of_films
    FROM film
    LEFT JOIN inventory ON inventory.film_id = film.film_id
      LEFT JOIN rental ON rental.inventory_id = inventory.inventory_id
      LEFT JOIN store ON store.store_id = inventory.store_id
      GROUP BY 1,2
)

SELECT store.store_id,
         af.store_address,
       IFNULL(rc.total_rentals,0) AS total_rentals,
       IFNULL(pc.total_revenue,0) total_revenue,
       IFNULL(pc.total_revenue/rc.total_rentals,0) AS
AVG_revenue_per_rental,
       IFNULL(rc.number_of_customers,0) AS number_of_customers,
       GROUP_CONCAT(mf.title ORDER BY mf.rank_of_films DESC separator ', ')
AS three_most_rented_films
FROM store
      LEFT JOIN Address_Finder af ON af.store_id = store.store_id
      LEFT JOIN Rental_Calculations rc ON rc.store_id = store.store_id
      LEFT JOIN Payment_Calculation pc  ON pc.store_id = store.store_id
      LEFT JOIN Most_Rented_3_Films mf  ON mf.store_id = store.store_id
                AND mf.rank_of_films <= 3

GROUP BY 1,2,3,4,5,6;

SELECT * FROM store_detailed_statistics;
```

4. Triggers

Solution 41. Create a trigger that automatically sets a default value for the active column in the customer table to 1 (active) whenever a new customer record is inserted, if the value is not provided. This ensures that all new customers are marked as active by default.

- In this problem, we should check the values which are intended to be inserted into the customer table before it is actually inserted, because if the trigger will fired after the insert, in that case the corresponding active column must be updated by update statement. However, there is a trigger restriction which does not allow to update the table which starts the trigger. Meaning that the insert command on the customer_copy table will fire the trigger, and hence the trigger will not have a authority to update the

table that fired it, in that case it is customer_copy table. Hence, the values that are going to be inserted into the table must be modified before they actually are inserted into the table. Therefore, “before insert on” statement must be used here. In that case, the active column will be set to 1 automatically by the trigger just before it is inserted into the table.

```
DELIMITER //
CREATE TRIGGER set_customer_active
BEFORE INSERT ON customer_copy
FOR EACH ROW
BEGIN
      IF NEW.is_active IS NULL THEN
            SET NEW.is_active = 1;
      END IF;
END //
DELIMITER ;
```

Solution 42. Create a trigger that automatically capitalizes the first name of a customer before it is inserted into the customer table. This ensures that all first names follow a consistent format.

- This trigger also will use “before insert on” statement first because it will not be able to update the customer_copy table once the values are inserted, and the trigger is fired. Hence, before the insertion operation is done on customer_copy table, the first letter of the first name and last name will be uppercased, and the remaining substring will be lowercased, then the results are concatenated by concat() function.

```
DELIMITER //
CREATE TRIGGER capitalize_first_name
BEFORE INSERT ON customer_copy
FOR EACH ROW
BEGIN
      SET NEW.first_name = CONCAT(UPPER(LEFT(NEW.first_name,1)),
LOWER(SUBSTRING(NEW.first_name,2))),
      NEW.last_name = CONCAT(UPPER(LEFT(NEW.last_name,1)),
LOWER(SUBSTRING(NEW.last_name,2)));
END //
DELIMITER ;
```

Solution 43. Create a trigger that logs every new customer added to the customer table into a customer_log table. The log should record the customer ID, first name, last name, and the date when the record was inserted

- Before creating the trigger, customer_log table will be created with appropriate data types.
- In this case, “after insert on” can be used, because the insertion operation will be done on the customer_copy table, but the fired trigger will not do any operation on that table, instead it will insert the customer log info into another table with the name of customer_log. Hence, the “after insert on” will be used here.

```
CREATE TABLE customer_log(
      log_id INT AUTO_INCREMENT PRIMARY KEY,
    customer_id INT,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    created_at DATETIME DEFAULT CURRENT_TIMESTAMP
);

DELIMITER //
CREATE TRIGGER customer_log_info
AFTER INSERT ON customer_copy
FOR EACH ROW
BEGIN
      INSERT INTO customer_log (customer_id,first_name,last_name)
    VALUES (NEW.customer_id, NEW.first_name, NEW.last_name) ;
END //
DELIMITER ;
```

Solution 44. Create a trigger that logs every time a film's rental rate is increased in the film table. The trigger should store the film id, old rental rate, new rental rate, and the date of the change into a rental_rate_log table.

- Before creating the trigger, a rental_rate_log table will be created with the given column names and appropriate data types.
- The problem states that “every time a film’s rental rate is increased” which means each time the rental rate in the film is updated the trigger should be fired. Hence the statement of “after update on” will be used here.
- After each update on the rental rate in the film table, the trigger will take the old rental rate value and new rental rate value by using new and old keywords, and also the date of change value, then it will log these values into the newly created rental_rate_log table.

```
CREATE TABLE rental_rate_log (
      rental_log_id INT AUTO_INCREMENT PRIMARY KEY,
      film_id INT,
    old_rental_rate DECIMAL(4,2),
    new_rental_rate DECIMAL(4,2),
    date_of_change DATETIME DEFAULT CURRENT_TIMESTAMP
);

DELIMITER //
CREATE TRIGGER store_rental_rate_changes
AFTER UPDATE ON film_copy
FOR EACH ROW
BEGIN
      INSERT INTO rental_rate_log (film_id, old_rental_rate,new_rental_rate)
    VALUES(NEW.film_id, OLD.rental_rate, NEW.rental_rate);
END //
DELIMITER ;
```

Solution 45. Create a trigger that automatically updates the last_update column in the customer table every time a customer's record is updated. This ensures that the last_update field always reflects the most recent change to the customer's information.

- This time the problem asks that whenever there is an update operation performed on the customer table, the trigger should be started and change the last_update value in the customer table. As is mentioned earlier, the trigger can not do any update on the table which starts that trigger. Hence, the trigger should update the new values before they are actually inserted into the table which fires the trigger. As a resul "before insert on" command is applied here, and the last_update value will be changed automatically when there is an update operation performed on the customer table.s

```
DELIMITER //
CREATE TRIGGER monitor_last_UPDATE
BEFORE UPDATE ON customer_copy
FOR EACH ROW
BEGIN
    SET NEW.last_UPDATE = NOW();
END //
DELIMITER ;
```

Solution 46. Create a trigger that logs deletions from the rental table. When a record is deleted, the trigger should insert a record into a rental_deletions_log table with details such as the rental id, deletion date, and the staff id who performed the deletion.

- Before creating the trigger as the problem states, a rental_deletions_log table will be created with the given column names and appropriate data types.
- The problem states that "when a record is deleted", it means after the deletion operation is detected on the rental table, the trigger will be started. Therefore, "after delete on" command will be written here. The trigger will log the old rental id and staff id into the rental_deletions_log table.
- There is no need to insert the deletion_date value manually into the rental_deletions_log table, because it will automatically detects the time of deletion and will store itself.

```
CREATE TABLE rental_deletions_log (
      deletion_id INT AUTO_INCREMENT PRIMARY KEY,
    rental_id INT,
    deletion_date DATETIME DEFAULT CURRENT_TIMESTAMP,
    staff_id INT
);


delimiter //
CREATE TRIGGER store_deletions
AFTER DELETE ON rental_copy
```

```
FOR EACH ROW
BEGIN
      INSERT INTO rental_deletions_log (rental_id, staff_id)
    VALUES (OLD.rental_id, OLD.staff_id);
END //
DELIMITER ;
```

Solution 47. Create a trigger that automatically updates the return_date in the rental table when a payment is recorded in the payment table for a specific rental. This trigger ensures that when a customer makes a payment, the corresponding rental is marked as returned, using the current date as the return date.

- This is simple problem which focuses on the effective use of join commands. The trigger will check the payment table and whenever a customer pays the rental payment, the return_date column for that corresponding customer will be updated and set equal to the date of payment.
- Because the trigger must update the return table after the payment amount is paid, “after insert on” command should be used here.

```
DELIMITER //
CREATE TRIGGER UPDATE_return_rate
AFTER INSERT ON payment_copy
FOR EACH ROW
BEGIN
    UPDATE rental_copy
    JOIN payment_copy
            ON rental_copy.rental_id = payment_copy.rental_id
    SET return_date = curdate()
    WHERE rental_copy.rental_id = NEW.rental_id;
END //
DELIMITER ;
```

Solution 48. Create a trigger that logs any new film inserted into the film table with a rental rate above a certain threshold (e.g., $4.99). The trigger should insert a record into a high_rated_films_log table with details such as the film title, rental rate, and insertion date whenever a film with a high rental rate is added.

- high_rated_films_log table should be created with the given column names and appropriate data types.
- problem states that “whenever a film with a high rental date is added” which means after the insertion into the film table, the condition will check the new rental rate wether

it is greater than 4.99. If yes the corresponding recently added film will be logged into the high_rated_films_log table.

```
CREATE TABLE high_rated_films_log(
      high_rated_films_log_id INT AUTO_INCREMENT PRIMARY KEY,
    film_title VARCHAR(100),
    rental_rate DECIMAL(4,2),
    insertion_date DATETIME DEFAULT CURRENT_TIMESTAMP
);

DELIMITER //
CREATE TRIGGER store_high_rated_films
AFTER INSERT ON film_copy
FOR EACH ROW
BEGIN
      IF NEW.rental_rate > 4.99 THEN
            INSERT INTO high_rated_films_log (film_title,rental_rate)
        VALUES (NEW.title, NEW.rental_rate);
      END IF;
END //
DELIMITER ;
```

Solution 49. Create a trigger that logs any changes to a customer's email address in the customer table. The trigger should capture the old email, the new email, the customer id, and the date of the change, and store this information in a customer_email_change_log table.

- customer_email_change_log table will be created.
- When the email is changed in the customer table, that change will be logged into that table. Hence, “after update on” command will be used.

```
CREATE TABLE customer_email_change_log (
      email_log_id INT AUTO_INCREMENT PRIMARY KEY,
    old_email VARCHAR(50),
    new_email VARCHAR(50),
    customer_id INT,
    date_of_change DATETIME DEFAULT CURRENT_TIMESTAMP
);
SELECT * FROM customer_email_change_log;

DELIMITER //
CREATE TRIGGER store_email_changes
AFTER UPDATE ON customer_copy
FOR EACH ROW
BEGIN
      IF OLD.email != NEW.email THEN
            INSERT INTO customer_email_change_log
        (old_email, new_email, customer_id)
        VALUES (OLD.email, NEW.email, NEW.customer_id);
      END IF;
END //
DELIMITER ;
```

5. Stored Procedures

Solution 50. Create a stored procedure that takes a customer's last name as an input parameter and returns all the details of customers with that last name. This procedure should allow easy retrieval of customer information by their last name.

- This procedure has one input which is the last name of the customer. Based on this input, the procedure will return all details of the customers including the customer first name, last name, address, total rentals he/she rented, total payment he/she spent on rentals.
- In the queries, views the calculations have been easily performed by the help of CTEs. In this case, we can use another method to perform the calculations and that method is quite effective inside the procedures. We can declare the variables inside the procedure, then assign values to these variables and finally retrieve them in final select statement.
- In this problem, we can calculate the total rentals made by each customer, total payment spent by each customer and derive the exact location of the customers from multiple tables. Hence, for simplicity and readibility, three variables are declared with the appropriate data types: total_rentals, total_payment and customer_address.
- we can assign the values to these variables by using "select aggregation_function() into variable" command. As in the following example, the count of rental_ids is selected from the rental table where the last name of the customer is equal to the input last_nm and selected (assigned) into the variable total_rentals.
- Then the total payment is calculated using the sum() function over the amount column in payment table where the last name of customer is equal to the input last_nm, then group by the results based on that customer, and assign into the total_payment variable.
- Finally, the exact location of the customer including the address, city, district and country is derived from multiple tables and assigned into the customer_address variable. This time also the operation is performed only for the customer whose last name is equal to the given input last_nm value.
- After getting the values for the variables, the first name, last name and other informations will be retrieved from the customer table.
- Important note 1: Don't forget to filter each calculation, each extraction and each derivation based on the input value, and filter out those that are not equal to the given input
- Important note 2: As is stated earlier, all calculation results must be checked if they are null or not. Ifnull() function helps to replace the null values with 0. Handling null values effectively is important, because there may be some customers that do not rented any films but because we do not handle those they will not appear in the results.

```
DELIMITER //
CREATE PROCEDURE customer_details (IN last_nm VARCHAR(25))
BEGIN
```

```
      DECLARE total_rentals INT;
    DECLARE total_payment INT;
    DECLARE customer_address TEXT;

    SELECT COUNT(rental.rental_id)
      FROM rental
    RIGHT JOIN customer
            ON customer.customer_id = rental.customer_id
      WHERE customer.last_name = last_nm
    INTO total_rentals;


    SELECT SUM(payment.amount)
    FROM payment
    RIGHT JOIN customer
            ON customer.customer_id = payment.customer_id
      WHERE customer.last_name = last_nm
    INTO total_payment;

    SELECT CONCAT(country.country, ", " , address.district,", ", city.city,
                          ", " , address.address,", " ,
address.postal_code)
      FROM customer
    JOIN address
            ON customer.address_id = address.address_id
      JOIN city
            ON city.city_id = address.city_id
      JOIN country
            ON country.country_id = city.country_id
      WHERE customer.last_name = last_nm
    INTO customer_address;

      SELECT
      first_name,
    last_name,
    customer_address,
    IFNULL(total_rentals,0) AS total_rentals,
    IFNULL(total_payment,0) AS total_payment,
    active
    FROM customer
      WHERE customer.last_name = last_nm;

END //
DELIMITER ;
```

Solution 51. Create a stored procedure that takes a film category name as an input parameter and returns a list of all films in that category. The procedure should include the film ti-tle, description, and rental rate.

- this procedure takes the name of the category as an input parameter.
- the list of all films which are in the given category must be returned. To do that, the films that fall in the given input category name will be filtered and selected.
- Then film, film_category and category tables must be joined to add the condition that the category name is equal to the input.

```
DELIMITER //
CREATE PROCEDURE films_BY_category (IN category_name VARCHAR(25))
BEGIN
    SELECT film.title,
    film.description,
    film.rental_rate
    FROM film
    JOIN film_category
            ON film.film_id = film_category.film_id
      JOIN category
            ON category.category_id = film_category.category_id
    WHERE category.name = category_name;

END //
DELIMITER ;

CALL films_BY_category("Action");
```

Solution 52. Create a stored procedure that takes a customer id and a new email address as input parameters and updates the email address of the specified customer. This procedure should help in easily updating customer contact information.

- The procedure takes two input parameters: customer id and new email.
- Before updating the email column for the given customer id, we should check whether there is any customer that have the same email which we want to insert as an input.
- We can do it by declaring the existing_email variable which looks through the customer table and finds the number of customers that have the email equal to the input new_email.
- If the existing_email is equal to 0 at the end, it means there is not any customer who has the same email that we want to use. Hence, we can update the email of the corresponding customer by setting it equal to the input new_email.

```
DELIMITER //
CREATE PROCEDURE UPDATE_customer_contact(IN cust_id INT, IN new_email
VARCHAR(50))
BEGIN
    DECLARE existing_email INT;

    SELECT COUNT(customer_id) FROM
    customer
    WHERE email = new_email
    INTO existing_email;

    IF existing_email > 0 THEN
        SIGNAL SQLSTATE "45000"
        SET MESSAGE_TEXT = "Email already exists";
      ELSE
            UPDATE customer
        SET email = new_email
        WHERE customer_id = cust_id;
```

```
      END IF;

END //
DELIMITER ;
```

Solution 53. Create a stored procedure that takes a store id as an input parameter and returns the total number of rentals processed by that store. This procedure will provide quick access to the rental count for any store.

- this procedure will take a store id as an input and will calculate the number of rentals processed by the store.
- It is simple join statement together with the where condition. The number of rental_id must be calculated from the rental table, and then it should be joined with inventory and store tables. The condition must be given that the store_id of the store table must be equal to the input store_id.

```
DELIMITER //
CREATE PROCEDURE Rental_COUNT_Calculations (IN store_id INT)
BEGIN
    SELECT
    store.store_id,
    COUNT(rental.rental_id) AS total_rentals
    FROM store
    JOIN inventory
       ON inventory.store_id = store.store_id
    JOIN rental
       ON rental.inventory_id = inventory.inventory_id
    WHERE store.store_id = store_id
    GROUP BY 1;
END //
DELIMITER ;
```

Solution 54. Create a stored procedure that takes a language id as an input parameter and returns a list of films available in that language. The procedure should include the film title and description.

- this procedure will have one input as a language id and will return all the films which are in this language,
- This problem requires simple join and filtering operation. Language and film tables will be joined and the films which have the language_id equal to the input will be listed.

```
DELIMITER //
CREATE PROCEDURE films_BY_language (IN lang_id INT)
BEGIN
      SELECT
    language.name,
```

```
    film.title
    FROM film
    JOIN language
            ON film.language_id = language.language_id
      WHERE language.language_id = lang_id;
END //
DELIMITER ;
```

Solution 55. Create a stored procedure that takes a city name as an input parameter and returns the total number of customers living in that city. This procedure can help you quickly find out how many customers are located in a particular area.

- the procedure takes the city id as an input parameter, and find the city corresponding to this id and then calculate the number of citizens that live in this city by applying the count() function to the customer_id column.

```
DELIMITER //
CREATE PROCEDURE city_citizens (IN city_idd INT)
BEGIN
      SELECT city.city,
    COUNT(customer_id) AS number_of_citizens
    FROM city
    JOIN address
            ON city.city_id = address.city_id
      JOIN customer
            ON customer.address_id = address.address_id
      WHERE city.city_id = city_idd
    GROUP BY 1;
END //
DELIMITER ;
```

Solution 56. Create a stored procedure that takes a month and a year as input parameters and returns the total rental revenue for that specific month across all stores. The procedure should aggregate payments made during the specified period.

- this procedure takes two inputs: month and year. It must show the total revenue earned on that month and year.
- To find the payment year and payment month, the year() and month() functions will extract the year and month from the payment_date column in the payment table. The filtering will be done based on that year and month. If the year and at the same time month of the payment is equal to the input year and month, respectively, then the sum of payment amount will be calculated for that month and year, and then aggregated. For simplictiy, let's explain step by step:
- the statement asks to return the total revenue generated within the given month in the given year. (i.e March, 2005). To filter the results based on these input year and month, we should know the years and months of the payments; that is we can derive those whose year and month are equal to the input year and month parameters.

- the total amount is calculated by using sum function on the amount column. If there are 3 payments in, for example, March, 2005; those three results will be summed and grouped as [the revenue earned in month = March, year = 2005].
- We can find the asked result by setting the year and month equal to the input year and month, respectively.

```
DELIMITER //
CREATE PROCEDURE total_revenue_BY_month (IN monthh INT, IN yearr INT
)
BEGIN
    SELECT
    YEAR(payment_date) AS given_year,
    MONTH(payment_date) AS given_month,
    SUM(amount) AS total_payment
    FROM payment
    WHERE YEAR(payment_date) = yearr AND
          MONTH(payment_date) = monthh
    GROUP BY 1,2;
END //
DELIMITER ;
```

Solution 57. Create a stored procedure that takes a customer id as an input parameter and checks the total number of rentals made by that customer. If the customer has rented more than a certain number of films (e.g., 50), update their status to a "vip" customer in a custom status column.

- This problem takes one input argument – customer id, and one output argument – customer status. If the number of rentals for the given customer id is more than threshold value, the procedure will set the output value customer_status to “VIP”, otherwise to “Standard”.
- It is required to calculate the number of total rentals made by the customer corresponding to the given customer id. Hence, total_rentals variable is declared, and assigned to the count of rental_id from the rental table where the customer id is equal to input cust_id.
- After assigning the value to the total_rentals variable, if-else conditions check whether the value is greater than threshold or not. If yes, then the output parameter will be set to “VIP” ; if no then “Standard”

```
DELIMITER //
CREATE PROCEDURE  categorize_the_customer (IN cust_id INT, OUT
customer_status VARCHAR(20))
BEGIN
    DECLARE total_rentals INT;

    SELECT COUNT(rental.rental_id) INTO total_rentals
    FROM rental
```

```
    JOIN customer
            ON customer.customer_id = rental.customer_id
      WHERE customer.customer_id = cust_id;

    IF total_rentals >= 25 THEN
        SET customer_status = "VIP";
      ELSE
            SET customer_status = "Standard" ;
      END IF;

END //
DELIMITER ;

CALL categorize_the_customer (1, @ customer_status);
SELECT @customer_status;
```

Solution 58. Create a stored procedure that takes a film id as an input parameter and returns the number of available copies of that film across all stores. The procedure should also return a message indicating whether the film is available or out of stock.

- the procedure takes one input – film id, and one output argument – is_available which checks the availability of the film
- To check the availability of the film, it is required to know how many films are there in the inventory and how many of them are currently rented and not returned. Therefore two variables are declared: all_films and rented_films.
- Calculation of the all_films in the inventory and alse the rented_films is explained in the solution of the previous questions. sdfkndsfdsfndsfn
- After assigning the values to these variables, the following condition must be checked:
- If the all_films is more than the rented_films, it means there are still corresponding film copies in the inventory, and they are available to be rented. In that case, algorithm must set the output argument is_available to “Available: N copies”. N here is calculated by subtracting the rented_films from all_films.
- If all_films is equal to rented_films, then it means all copies of the corresponding input film are currently rented and not returned. Therefore, there is not any avaiilable copy of the input film_id in the inventory. In that case, the is_available output will be set to “Out of stock” text.

```
DELIMITER //
CREATE PROCEDURE check_available_films (IN flm_id INT, OUT
is_available VARCHAR(25))
BEGIN
      DECLARE all_films INT;
    DECLARE rented_films INT;

    SELECT
```

```
    COUNT(inventory_id)
    FROM inventory
    WHERE inventory.film_id = flm_id
    INTO all_films ;

    SELECT
    COUNT(rental.rental_id)
    FROM rental
    LEFT JOIN inventory
            ON rental.inventory_id = inventory.inventory_id
        AND rental.return_date IS NULL
    WHERE inventory.film_id = flm_id
    INTO rented_films ;

    IF all_films - rented_films > 0 THEN
       SET is_available = CONCAT("Available: ",all_films -
rented_films, " copies") ;
    ELSE
            SET is_available = "Out of stock" ;
    END IF;
END //

DELIMITER ;

CALL check_available_films (1, @is_available);
SELECT @is_available;
```

Solution 59. Create a procedure that takes a customer id as an input parameter and returns the full name of the customer in the format "first name last name". This function should concatenate the first and last names of the customer.

- This procedure takes one input argument customer id and one output argument full_name. The output must be the concatenation of the first and last name of the customer whose customer id is equal to the id given in the input.
- The variable fullname is declared and the first name and last name of the customer who have the customer id equal to the input customer id are retrieved and concatenated. The result of the concatenation is assigned to the variable.

```
DELIMITER //
CREATE PROCEDURE customer_full_name (IN cust_id INT, OUT full_name
VARCHAR(100))
BEGIN
    DECLARE fullname VARCHAR(100);

    SELECT
    CONCAT(customer.first_name, " ", customer.last_name)
```

```
    INTO fullname
    FROM customer
    WHERE customer_id = cust_id;

    SET full_name = fullname;
END //
DELIMITER ;

CALL customer_full_name(1, @fullname);
SELECT @fullname;
```

Solution 60. Create a function that takes a category id as an input parameter and returns the total number of films available in that category. This function will provide a quick count of films within a specific category.

- this is a function and the functions are like the stored procedures but they return some values with the “returns ---” statement.
- In this problem, the function takes the category id as an input, and calculate the count of films from the film_category table whose category_id value is equal to the input categ_id value.
- Then the function assigns the result into the returned variable by using set statement.
- At the end of function, the result must be returned by using “return result” statemen

```
DELIMITER //
CREATE FUNCTION count_of_films_in_category (categ_id INT)
RETURNS VARCHAR(100)
READS SQL DATA
BEGIN
    DECLARE nm_of_films INT;
    DECLARE result VARCHAR(100);

    -- COUNT the number of films in the given category
    SELECT COUNT(film_id) INTO nm_of_films
    FROM film_category
    WHERE category_id = categ_id;

    -- CREATE the result string
    SET result = CONCAT(nm_of_films, ' available films in category_id
= ', categ_id);

    -- Return the result
    RETURN result;
END //
DELIMITER ;

SELECT count_of_films_in_category(1);
```