```
/* Q1  Create a table called employees with the following structure?
: emp_id (integer, should not be NULL and should be a primary key)Q
: emp_name (text, should not be NULL)Q
: age (integer, should have a check constraint to ensure the age is at
least 18)
: email (text, should be unique for each employee)Q
: salary (decimal, with a default value of 30,000).

Write the SQL query to create the above table with all constraints. */

create database pw;
use pw;
create table employee
(
emp_id  int  not null primary key ,
emp_name  varchar (40)  not null,
age  int check(age >=18),
email varchar(20) unique,
salary decimal default 30000
 );

 describe employee;

/* 2. Explain the purpose of constraints and how they help maintain data
integrity in a database. Provide
examples of common types of constraints.
Constraints are the rules that we can apply on the type of data in a
table. That is, we can specify the limit on the type of data that can be
stored in a particular column in a table using constraints.

The available constraints in SQL are:


NOT NULL: This constraint tells that we cannot store a null value in a
column. That is, if a column is specified as NOT NULL then we will not be
able to store null in this particular column any more.
UNIQUE: This constraint when specified with a column, tells that all the
values in the column must be unique. That is, the values in any row of a
column must not be repeated.
PRIMARY KEY: A primary key is a field which can uniquely identify each row
in a table. And this constraint is used to specify a field in a table as
primary key.
FOREIGN KEY: A Foreign key is a field which can uniquely identify each row
in a another table. And this constraint is used to specify a field as
Foreign key.
CHECK: This constraint helps to validate the values of a column to meet a
particular condition. That is, it helps to ensure that the value stored in
a column meets a specific condition.
DEFAULT: This constraint specifies a default value for the column when no
value is specified by the user.

example :
Not NULL
int(6) not null
```

```
UNIQUE
email varchar(20) UNIQUE,
Primary Key
emp_id  int  not null primary key,
CHECK:
age  int check(age >=18)
salary decimal default 30000

\* Q3.Why would you apply the NOT NULL constraint to a column? Can a
primary key contain NULL values? Justify
your answer.
Ans: -- The NOT NULL constraint ensures that a column cannot have a NULL
value. It is applied to enforce the presence of data in fields where
missing information would compromise the accuracy or integrity of the
database.
CREATE TABLE employees (
    emp_id INTEGER PRIMARY KEY, -- Cannot be NULL or duplicate
    emp_name varchar(40) NOT NULL      -- Ensures every employee id  has a
corresponding name
);

\*4  Explain the steps and SQL commands used to add or remove constraints
on an existing table. Provide an
example for both adding and removing a constraint.
Example of Adding Constraint

Step 1: Create Table
Step 2  Add Constraint by Using Alter Table
ALTER TABLE employees
ADD CONSTRAINT check_age CHECK (age >= 18);
Step 3 Delete/Remove Constraint by using Drop contraint
ALTER TABLE employees
drop CONSTRAINT check_age ;

Simmilarly other constaints can be added and drop from the table

\*5. Explain the consequences of attempting to insert, update, or delete
data in a way that violates constraints.
Provide an example of an error message that might occur when violating a
constraint
INSERT INTO employees (emp_id, emp_name) VALUES (1, NULL);
-- ERROR: null value in column "emp_name" violates not-null constraint

UPDATE employees SET check_age = 16 WHERE emp_id =20;
-- ERROR: new row for relation "employees" violates check constraint
"check_age"
DELETE FROM departments WHERE dept_id = 1;
-- ERROR: update or delete on table "departments" violates foreign key
constraint "fk_dept_id" on table "employees"

Q6. You created a products table without constraints as follows:

CREATE TABLE products (
    product_id INT,
```

```
    product_name VARCHAR(50),
    price DECIMAL(10, 2));
Now, you realise that?
: The product_id should be a primary keyQ
: The price should have a default value of 50.00
*/
CREATE TABLE products (
    product_id INT,
    product_name VARCHAR(50),
    price DECIMAL(10, 2)
);
describe products;
alter table products
ADD CONSTRAINT primary key(product_id);
SET SQL_SAFE_UPDATES = 0;
update products set price=50.00
where price is null;


-- Q7 Write the query to fetch the student Name and Class name using Inner
Join --
CREATE TABLE Students (
    student_id INT PRIMARY KEY,
    student_name VARCHAR(50) NOT NULL,
    class_id INT
);
INSERT INTO Students (student_id, student_name, class_id)
VALUES
    (1, 'Alice', 101),
    (2, 'Bob', 102),
    (3, 'Charlie', 101);
    select * from Students;
    CREATE TABLE Classes (
    class_id INT PRIMARY KEY,
    class_name VARCHAR(50) NOT NULL
);

-- Inserting sample data
INSERT INTO Classes (class_id, class_name)
VALUES
(101, 'Math'),
(102, 'Science'),
(103, 'History');
 # usING iNNER jOIN  TO get student Name and class Name
SELECT
    s.student_name,
    c.class_name
FROM
    Students s
INNER JOIN
    Classes c ON s.class_id = c.class_id;


-- Q8
```

```sql
CREATE TABLE Orders (
    order_id INT PRIMARY KEY,
    order_date DATE,
    customer_id INT
);

INSERT INTO Orders (order_id, order_date, customer_id)
VALUES
(1, '2024-01-01', 101),
(2, '2024-01-03', 102);
CREATE TABLE Customers (
    customer_id INT PRIMARY KEY,
    customer_name VARCHAR(50) NOT NULL
);

INSERT INTO Customers (customer_id, customer_name)
VALUES
(101, 'Alice'),
(102, 'Bob');
CREATE TABLE Products1 (
    product_id INT PRIMARY KEY,
    product_name VARCHAR(50) NOT NULL,
    order_id INT)
;
INSERT INTO Products1 (product_id, product_name, order_id)
VALUES
(1, 'Laptop', 1),
(2, 'Phone', NULL);
use pw;
SELECT o.order_id,c.customer_name,p.product_name
FROM  Customers c
INNER JOIN
    Orders o ON c.customer_id = o.customer_id
LEFT JOIN
    Products1 p ON o.order_id = p.order_id
UNION
SELECT order_id,c.customer_name,p.product_name
FROM     Customers c
LEFT JOIN
    Products1 p ON c.customer_id = p.order_id;


-- Q-9
CREATE TABLE Product (
    product_id INT PRIMARY KEY,
    product_name VARCHAR(50) NOT NULL
);
INSERT INTO Product (product_id, product_name)
VALUES
(101, 'Laptop'),
(102, 'Phone');

CREATE TABLE Sales (
    sale_id INT PRIMARY KEY,
```

```
    product_id INT,
    amount DECIMAL(10, 2)
);
INSERT INTO Sales (sale_id, product_id, amount)
VALUES
(1, 101, 500),
(2, 102, 300),
(3, 101, 700);
select * from product;

SELECT product_name,sum(s.amount) as amount
FROM
    Sales s
INNER JOIN
    Product p ON s.product_id = p.product_id
group by product_name
order by amount ;

   -- Q10 --
   CREATE TABLE Orders1 (
    order_id INT PRIMARY KEY,
    order_date DATE,
    customer_id INT,
    FOREIGN KEY (customer_id) REFERENCES Customers1(customer_id)
);

-- Creating the Customers table
CREATE TABLE Customers1 (
    customer_id INT PRIMARY KEY,
    customer_name VARCHAR(50) NOT NULL
);
 CREATE TABLE Order_Details (
    order_id INT,
    product_id INT,
    quantity INT,
    PRIMARY KEY (order_id, product_id),
    FOREIGN KEY (order_id) REFERENCES Orders(order_id)
);

INSERT INTO Customers1 (customer_id, customer_name)
VALUES
(1, 'Alice'),
(2, 'Bob');

-- Inserting data into the Orders table
INSERT INTO Orders1 (order_id, order_date, customer_id)
VALUES
(1, '2024-01-02', 1),
(2, '2024-01-05', 2);
INSERT INTO Order_Details (order_id, product_id, quantity)
VALUES
(1, 101, 2),
(1, 102, 1),
(2, 101, 3);
```

```
SELECT
    o.order_id,
    c.customer_name,
    od.quantity
FROM
    Orders o
INNER JOIN
    Customers c ON o.customer_id = c.customer_id
INNER JOIN
    Order_Details od ON o.order_id = od.order_id;

-- Sql Commands --
-- Q1
 -- Table Actor    Primary Key  actor_id
-- Actor_award     Primary Key  actor_id     Foreign Key




-- Q2--

select * from actor;

-- Q3 --
show tables;
select * from customer;

-- Q4 --
select country from country;

-- q5 --
select active from customer;

-- q6 --
use sakila;
select rental_id
from rental
where customer_id =1 ;

-- q7 --
SELECT *
FROM rental
WHERE DATEDIFF(return_date, rental_date) > 5;

-- q8
select count(*)  from film
WHERE replacement_cost > 15 AND replacement_cost < 20;

-- Q9 --
select distinct first_name
from actor;

-- q 10 --
```

```
select * from customer
limit 10;

-- q11 --

select * from customer;

select  * from customer where
 first_name like 'b%'
 limit 3;
-- Q12 --
select * from film
where rating = 'G'
limit 5;

-- Q13 --
use sakila;
select  * from customer where
 first_name like 'a%';

-- Q14 --

select  * from customer where
first_name like '%a';

-- Q 15 --

select * from city where city like 'a%' and city like '%a';

-- Q 16 --
select  * from customer where
first_name like '%NI%';

-- Q 17 --

select  * from customer where
first_name like '_r%';

-- Q 18 --
select  * from customer where
first_name like 'a____';

-- Q 19--
select  * from customer where
first_name like 'a%_%O';

-- Q 20 --
select * from film
where rating IN ('pg','pg-13');

-- Q 21 --
select * from film
where length between 50 and 100;
```

```sql
-- Q 22 --

select * from actor
limit 50;

-- Q 23 --

select distinct (film_id)
from inventory;

-- Functions
-- Q 1 --
select  count(rental_id)
from rental;

select * from rental;
-- Q 2  --
select Avg(datediff(return_date,rental_date)) as rental_duration
from rental;

-- Q 3 --
select * from customer;
select UPPER(first_name) as First_Name , upper(last_name) as Last_Nmae
from customer;

--  Q 4 --
select month(rental_date) from rental;

-- q 5 --
select customer_id, count(*)  as count_rental from rental
group by customer_id;

-- Q 6 --
select store_id,sum(rental_rate) as total_revenue
from film f
join inventory i on i.film_id=f.film_id
group by store_id ;

select * from  film;
select * from  film_category;
select * from rental ;
-- Q 7 --

SELECT fc.category_id, c.name AS category_name, COUNT(r.rental_id) AS
total_rentals
FROM film_category fc
JOIN film f ON fc.film_id = f.film_id
JOIN inventory i ON f.film_id = i.film_id
JOIN rental r ON i.inventory_id = r.inventory_id
JOIN category c ON fc.category_id = c.category_id
GROUP BY fc.category_id, c.name
ORDER BY total_rentals DESC;

-- Q 8 --
```

```
select * from language;
select * from film;

select name,avg(rental_rate) as avg_rental
from film f
join language l on l.language_id=f.language_id
group by name;

-- Q 9 --
select * from  film;
select title as movie_title,first_name,last_name
from rental r
join inventory i on i.inventory_id=r.inventory_id
join film f on i.film_id=f.film_id
join customer c on c.customer_id=r.customer_id;

-- Q 10 --
select  concat(first_name,Last_name) as Name
from actor a
join film_actor fa on a.actor_id=fa.actor_id
join film f on f.film_id=fa.film_id
where f.title='Gone with the Wind';

-- Q11 --
SELECT
    c.customer_id,
    CONCAT(c.first_name, ' ', c.last_name) AS customer_name,
    SUM(p.amount) AS total_amount_spent
FROM
    customer c
JOIN
    payment p ON c.customer_id = p.customer_id
JOIN
    rental r ON p.rental_id = r.rental_id
GROUP BY
    c.customer_id, customer_name
ORDER BY
    total_amount_spent DESC;


-- Q 12--
use sakila;
select first_name, last_name, title as movie_title
from customer c
join address a on c.address_id = a.address_id
join city ci on a.city_id = ci.city_id
join rental r on c.customer_id = r.customer_id
join inventory i on r.inventory_id = i.inventory_id
join film f on i.film_id = f.film_id
where ci.city = 'London'
Group by first_name, last_name, title;

-- Q-12-
```

```
select first_name, last_name, title as movie_title
from customer c
join address a on c.address_id = a.address_id
join city ci on a.city_id = ci.city_id
join rental r on c.customer_id = r.customer_id
join inventory i on r.inventory_id = i.inventory_id
join film f on i.film_id = f.film_id
where ci.city = 'London'
order by first_name, last_name, title;

-- Q 13 --
use sakila;
select f.film_id,title as movie_title, COUNT(r.rental_id) as rental_count
from film f
join inventory i on f.film_id = i.film_id
join rental r on i.inventory_id = r.inventory_id
group by f.film_id, title
order by rental_count desc
limit 5;

-- Q-14-
select c.customer_id,first_name,last_name
from customer c
join rental r on c.customer_id = r.customer_id
join inventory i on r.inventory_id = i.inventory_id
join store s on i.store_id = s.store_id
group by customer_id,first_name,last_name
having count(distinct(s.store_id=1 and s.store_id=2));

-- Q1 --
SELECT
    c.customer_id,
    c.first_name,
    c.last_name,
    SUM(p.amount) AS total_spent,
    RANK() OVER (ORDER BY SUM(p.amount) DESC) AS rank_
FROM customer c
JOIN payment p ON c.customer_id = p.customer_id
GROUP BY c.customer_id, c.first_name, c.last_name
ORDER BY rank_;

-- Q-2-
SELECT title as film_title,p.payment_date,SUM(p.amount) over (partition by
f.film_id order by p.payment_date) as cumulative_revenue
from payment p
join rental r on p.rental_id = r.rental_id
join inventory i on r.inventory_id = i.inventory_id
join film f on i.film_id = f.film_id
order by title, p.payment_date;

-- Q-3-
select f.film_id,title as film_title,floor(f.length/10)*10 as
length_range,avg(datediff(r.return_date,r.rental_date)) as rental_duration
from film f
```

```
join inventory i on f.film_id = i.film_id
join rental r on i.inventory_id = r.inventory_id
group by f.film_id,title
order by length_range,rental_duration desc ;

-- Q4 --
WITH FilmRentalCounts AS (
    SELECT
        f.film_id,
        f.title AS film_title,
        c.category_id,
        c.name AS category_name,
        COUNT(r.rental_id) AS rental_count
    FROM rental r
    JOIN inventory i ON r.inventory_id = i.inventory_id
    JOIN film f ON i.film_id = f.film_id
    JOIN film_category fc ON f.film_id = fc.film_id
    JOIN category c ON fc.category_id = c.category_id
    GROUP BY f.film_id, f.title, c.category_id, c.name
),
RankedFilms AS (
    SELECT
        film_id,
        film_title,
        category_name,
        rental_count,
        ROW_NUMBER() OVER (PARTITION BY category_name ORDER BY
rental_count DESC) AS rank_
    FROM FilmRentalCounts
)
SELECT
    film_title,
    category_name,
    rental_count
FROM RankedFilms
WHERE rank_ <= 3
ORDER BY category_name, rank_;

with CustomerRentals as (select c.customer_id,concat(c.first_name, ' ',
c.last_name) as customer_name,
        count(r.rental_id) as total_rentals
    from customer c
    join rental r on c.customer_id = r.customer_id
    group by c.customer_id, c.first_name, c.last_name
),AverageRentals as (select avg(total_rentals) as avg_rentals
    from CustomerRentals
)
select cr.customer_id,cr.customer_name,cr.total_rentals,ar.avg_rentals,
    (cr.total_rentals - ar.avg_rentals) as rental_difference
from CustomerRentals cr
cross join AverageRentals ar
order by rental_difference desc;
```

```
-- Q-6-
select date_format(payment_date,'%Y-%m') as revenue_month, SUM(amount) AS
total_revenue
from payment
group by revenue_month
order by revenue_month;

-- Q-7-
with CustomerSpending as (select c.customer_id,concat(c.first_name, ' ',
c.last_name) as customer_name,sum(p.amount) as total_spending
    from customer c
    join payment p on c.customer_id = p.customer_id
    group by c.customer_id, c.first_name, c.last_name
),RankedSpending as (select customer_id,
customer_name,total_spending,ntile(5) over (order by total_spending desc)
as spending_percentile
 from CustomerSpending)
select  customer_id,customer_name,total_spending
from RankedSpending
where spending_percentile = 1
order by total_spending desc;

-- Q-8-
with category_rental as (select cat.category_id,cat.name as category_name,
COUNT(r.rental_id) as total_rentals
from rental r
join inventory i on r.inventory_id = i.inventory_id
join film f on i.film_id = f.film_id
join film_category fc on f.film_id = fc.film_id
join category cat on fc.category_id = cat.category_id
group by cat.category_id,cat.name)select
category_name,total_rentals,sum(total_rentals) over (order by
total_rentals desc) as running_total
from category_rental
order by total_rentals desc;

-- Q-9-

with CategoryAverageRentals as (select c.category_id,c.name as
category_name,avg(rental_count) as avg_rentals
    from (select c.category_id,count(r.rental_id) as rental_count
        from rental r
        join inventory i on r.inventory_id = i.inventory_id
        join film f on i.film_id = f.film_id
        join film_category fc on f.film_id = fc.film_id
        join category c on fc.category_id = c.category_id
        group by c.category_id, f.film_id
    ) CategoryRentals  group by c.category_id, c.name
),
FilmRentals as (select f.film_id,f.title as
film_title,c.category_id,count(r.rental_id) as film_rental_count
    from rental r
    join inventory i on r.inventory_id = i.inventory_id
```

```
    join film f on i.film_id = f.film_id
    join film_category fc on f.film_id = fc.film_id
    join category c on fc.category_id = c.category_id
    group by f.film_id, f.title, c.category_id
)select fr.film_title,ca.category_name,fr.film_rental_count,ca.avg_rentals
from FilmRentals fr
join CategoryAverageRentals ca on fr.category_id = ca.category_id
where fr.film_rental_count < ca.avg_rentals
order by ca.category_name, fr.film_rental_count;

-- Q-10-
select date_format(payment_date,'%Y-%m') as revenue_month, SUM(amount) AS
total_revenue
from payment
group by revenue_month
order by revenue_month
limit 5;
-- Normalization and CTE --


/* Q1   First Normal Form (1NF): Identify a table in the Sakila database
that violates 1NF. Explain how you would normalize it to achieve 1NF.

Ans 1)
Explanation of 1NF:
1NF requires that:
Each column in a table must contain only atomic (indivisible) values.
Each column must contain unique data (no repeating groups or arrays).
Each row must be unique.
Example of 1NF Violation:
In the Sakila database, a common example of a table that might violate 1NF
could be one where there are columns containing multiple values in a
single cell, such as an actor's full name stored in one column ("John,
Smith"), or a column that holds a list of genres like "Action, Drama".
How to Normalize to 1NF:
Separate Atomic Values:
Split any columns that contain multiple values into separate columns. For
example, split a column FullName into FirstName and LastName.
Create Additional Tables if Necessary:
If a table has columns containing repeating groups (e.g., multiple
genres), create a related table with foreign keys to establish one-to-many
relationships.

A typical example in the Sakila database where 1NF may be violated is the
film table. In some cases, you might find columns that could potentially
store multiple values, such as a list of actors or categories in a single
column.

Scenario: film Table with Potential 1NF Violation
Imagine the film table had a column called actors_list containing data
like "Tom Hanks, Julia Roberts, Brad Pitt" for one film. This would
violate 1NF because:

The actors_list column contains multiple values in a single cell.
```

It is not atomic, as it combines multiple actor names in one field.
How to Normalize to 1NF:
To normalize the film table to meet 1NF, you need to:

Remove Repeating Groups:

Create a new table, e.g., film_actor, that relates film_id and actor_id to ensure each actor associated with a film is stored as a separate row.
Define Relationships:

Create a one-to-many relationship between the film table and the film_actor table, where:
film_id is the foreign key from the film table.
actor_id is the foreign key from an actor table that stores individual actor records.

Q2 Second Normal Form (2NF): Choose a table in Sakila and describe how you would determine whether it is in 2NF.
If it violates 2NF, steps to normalize it.

Ans 2
A table is in Second Normal Form (2NF) if:

It is in First Normal Form (1NF) (i.e., it has no repeating groups, and all columns have atomic values).
All non-prime attributes (attributes not part of a candidate key) are fully functionally dependent on the entire primary key, not just part of it.
Violation of 2NF usually occurs when a table has a composite primary key, and non-prime attributes depend on only part of that key (partial dependency).

Example: film_actor Table in Sakila Database
The film_actor table has a composite primary key (film_id, actor_id):

Primary Key: (film_id, actor_id)

How to Determine if It Violates 2NF:
Check if there are non-prime attributes in the table.
Determine if any non-prime attribute depends only on part of the composite primary key, not the whole key.
In this case, last_update is related to the entire film_actor record (both film_id and actor_id), so it does not violate 2NF.
Another Example: Hypothetical Table
Consider a hypothetical table that violates 2NF:

Table: rental_info
Columns: (rental_id, film_id, rental_date, film_title)

If rental_id is a primary key and film_title only depends on film_id, this would violate 2NF because film_title is partially dependent on film_id, not rental_id.

Steps to Normalize to 2NF:
Identify Partial Dependencies:

Find non-prime attributes that depend on part of the composite primary key.
Create New Tables:

Split the table into two or more tables to remove partial dependencies.
For the rental_info example, create:
A rental table with (rental_id, rental_date, film_id).
A film table with (film_id, film_title).
Establish Relationships:

Use foreign keys to maintain relationships between the tables.

Q3) Third Normal Form (3NF): a. Identify a table in Sakila that violates 3NF. Describe the transitive dependencies
present and outline the steps to normalize the table to 3NF.

Ans) xplanation of Third Normal Form (3NF):
A table is in Third Normal Form (3NF) if:

It is in Second Normal Form (2NF).
It has no transitive dependencies, meaning non-prime attributes should not depend on other non-prime attributes.
Transitive dependency occurs when a non-prime attribute depends on another non-prime attribute instead of directly depending on the primary key.

Example of 3NF Violation in the Sakila Database:
Consider a hypothetical violation in the rental table:

Table: rental
Columns: rental_id, customer_id, store_id, customer_address, rental_date
In this example:

rental_id is the primary key.
customer_id is a foreign key that references a customer.
customer_address depends on customer_id, not directly on rental_id. This creates a transitive dependency because customer_address should not be part of the rental table.
Steps to Normalize the rental Table to 3NF:
Identify Transitive Dependencies:

Find non-prime attributes (e.g., customer_address) that depend on other non-prime attributes (e.g., customer_id).
Create Separate Tables:

Split the rental table to eliminate transitive dependencies. Create a customer table with customer_id and customer_address, and keep the rental table with only directly relevant columns.
New Table Structures:

rental table:
rental_id (Primary Key)
customer_id (Foreign Key)
store_id
rental_date
customer table:
customer_id (Primary Key)
customer_address
Establish Relationships:

Use foreign keys to connect the rental table to the customer table based on customer_id.
Explanation of Why This Is Now in 3NF:
After splitting the tables, all non-prime attributes depend only on the primary key of their respective tables, removing transitive dependencies and achieving 3NF.

Q4) 4. Normalization Process:
a. Take a specific table in Sakila and guide through the process of normalizing it from the initial unnormalized form up to at least 2NF

Ans)
normalization process step-by-step using the rental table from the Sakila database as an example. We'll guide it from an unnormalized state to Third Normal Form (3NF).

Step 1: Unnormalized Form (UNF)
Imagine the rental table initially has redundant and nested data:

rental_id	customer_id	customer_name	film_id	film_title	rental_date	return_date	store_id	store_address
1	101	John Smith	2001	Toy Story	2024-11-01	2024-11-05	5	123 Main St
2	102	Alice Brown	2002	The Matrix	2024-11-02	2024-11-06	5	123 Main St

Step 2: Normalize to First Normal Form (1NF)
Goal: Ensure that each column contains only atomic (indivisible) values and remove any repeating groups.

Changes:

Ensure that columns like customer_name and film_title only store one piece of data per cell.
Revised 1NF Table Structure:

rental_id	customer_id	film_id	rental_date	return_date	store_id
1	101	2001	2024-11-01	2024-11-05	5

2 102 2002 2024-11-02 2024-11-06 5
Create separate tables:

customer table: customer_id, customer_name
film table: film_id, film_title
store table: store_id, store_address
Step 3: Normalize to Second Normal Form (2NF)
Goal: Remove partial dependencies. All non-prime attributes should be fully dependent on the entire primary key.

Identifying Partial Dependencies:

In the rental table, store_id is dependent on rental_id, which is the primary key.
Attributes like customer_name in the customer table are fully dependent on customer_id, and film_title in the film table is fully dependent on film_id.
Revised 2NF Table Structures:

rental table:

rental_id (Primary Key)
customer_id (Foreign Key)
film_id (Foreign Key)
rental_date
return_date
store_id (Foreign Key)
customer table:

customer_id (Primary Key)
customer_name
film table:

film_id (Primary Key)
film_title
store table:

store_id (Primary Key)
store_address
Step 4: Normalize to Third Normal Form (3NF)
Goal: Remove transitive dependencies. Non-prime attributes should not depend on other non-prime attributes.

Identifying Transitive Dependencies:

If store_address is included in the rental table, it creates a transitive dependency through store_id.
Revised 3NF Table Structures:

rental table:

rental_id (Primary Key)
customer_id (Foreign Key)
film_id (Foreign Key)

rental_date
return_date
store_id (Foreign Key)
customer table:

customer_id (Primary Key)
customer_name
film table:

film_id (Primary Key)
film_title
store table:

store_id (Primary Key)
store_address
Final Normalized Form:
The rental table now only contains columns that directly depend on its primary key, rental_id.
The store table holds store_id and store_address, removing any transitive dependency from the rental table.
This step-by-step normalization ensures that the rental table in the Sakila database is in Third Normal Form (3NF). Would you like to see SQL commands for creating these normalized tables?

_____
_____
_____
*/

```
-- Ans 5
WITH ActorFilmCount AS (
    SELECT
        a.  actor_id,
        a.  first_name,
        a.  last_name,
        COUNT(fa.film_id) AS num_films
    FROM actor a
    JOIN film_actor fa ON a.actor_id = fa.actor_id
    GROUP BY a.actor_id, a.first_name, a.last_name
)
SELECT
    first_name,
    last_name,
    num_films
FROM ActorFilmCount
ORDER BY num_films DESC;

-- Ans 6 --
WITH FilmLanguageInfo AS (SELECT f.title AS film_title,l.name AS
language_name,f.rental_rate
    FROM film f
```

```
    JOIN language l ON f.language_id = l.language_id
)SELECT film_title,language_name,rental_rate
FROM FilmLanguageInfo
ORDER BY language_name, film_title;


Ans 7--

WITH CustomerRevenue AS (SELECT c.customer_id,CONCAT(c.first_name, ' ',
c.last_name) AS customer_name,SUM(p.amount) AS total_revenue
    FROM customer c
    JOIN payment p ON c.customer_id = p.customer_id
    GROUP BY c.customer_id, c.first_name, c.last_name
)SELECT customer_name, total_revenue
FROM CustomerRevenue
ORDER BY total_revenue DESC;

-- Ans 8 --
WITH FilmRentalRank AS (SELECT f.film_id,f.title AS
film_title,f.rental_duration,
      ROW_NUMBER() OVER (ORDER BY f.rental_duration DESC) AS rank_
    FROM film f
)SELECT film_title,rental_duration,rank_
FROM FilmRentalRank
ORDER BY rank_;
-- Q9 --
WITH CustomerRentalCount AS (SELECT c.customer_id,COUNT(r.rental_id) AS
rental_count
    FROM customer c
    JOIN rental r ON c.customer_id = r.customer_id
    GROUP BY c.customer_id
    HAVING COUNT(r.rental_id) > 2
)SELECT
c.customer_id,c.first_name,c.last_name,c.email,c.address_id,crc.rental_cou
nt
FROM CustomerRentalCount crc
JOIN customer c ON crc.customer_id = c.customer_id
ORDER BY crc.rental_count DESC;

-- Q 10 --
WITH MonthlyRentalsCTE AS (
    SELECT
        DATE_FORMAT(rental_date, '%Y-%m') AS rental_month,
        COUNT(*) AS total_rentals
    FROM rental
    GROUP BY rental_month
)
SELECT
    rental_month,
    total_rentals
FROM MonthlyRentalsCTE
ORDER BY rental_month;

-- Q11
```

```
WITH ActorPairs AS (
    SELECT
        fa1.actor_id AS actor_1_id,
        fa2.actor_id AS actor_2_id,
        f.title AS film_title
    FROM film_actor fa1
    JOIN film_actor fa2 ON fa1.film_id = fa2.film_id
    JOIN film f ON fa1.film_id = f.film_id
    WHERE fa1.actor_id < fa2.actor_id  -- Ensures each pair is only listed
once
)
SELECT
    a1.first_name AS actor_1_first_name,
    a1.last_name AS actor_1_last_name,
    a2.first_name AS actor_2_first_name,
    a2.last_name AS actor_2_last_name,
    ap.film_title
FROM ActorPairs ap
JOIN actor a1 ON ap.actor_1_id = a1.actor_id
JOIN actor a2 ON ap.actor_2_id = a2.actor_id
ORDER BY ap.film_title, actor_1_last_name, actor_2_last_name;


WITH RECURSIVE EmployeeHierarchy AS (
    -- Start with the specific manager
    SELECT
        staff_id,
        first_name,
        last_name,
        manager_id
    FROM staff
    WHERE staff_id = <manager_id>  -- Replace <manager_id> with the actual
manager's staff_id

    UNION ALL

    -- Recursively find employees who report to the current staff member
    SELECT
        s.staff_id,
        s.first_name,
        s.last_name,
        s.manager_id
    FROM staff s
    JOIN EmployeeHierarchy eh ON s.manager_id = eh.staff_id
)
SELECT
    staff_id,
    first_name,
    last_name,
    manager_id
FROM EmployeeHierarchy
ORDER BY staff_id;
```