```sql
create table employees(
    emp_id int not null primary key,
    emp_name varchar(100) not null,
    age int check (age >= 18),
    email varchar(100) unique,
    salary decimal default 30000
);
```

2. Explain the purpose of constraints and how they help maintain data integrity in a database. Provide examples of common types of constraints.

Purpose of Constraints in a Database:

Constraints in a database are rules enforced on data columns in tables to ensure accuracy, reliability, and integrity of the data. They help maintain **data integrity** by restricting the type of data that can be stored, preventing invalid data entries, and maintaining consistent relationships between tables.

How Constraints Help Maintain Data Integrity:

1. **Preventing Invalid Data Entries:**
    - Constraints restrict what kind of data can be entered into a table (e.g., no negative age values).
2. **Enforcing Relationships:**
    - Constraints like foreign keys ensure valid relationships between tables (e.g., an order must belong to an existing customer).
3. **Ensuring Uniqueness:**
    - Constraints help avoid duplicate records where they shouldn't exist (e.g., two users with the same email).
4. **Maintaining Consistency:**

- By enforcing rules, constraints ensure the data remains consistent throughout operations like insertions, deletions, or updates.

Common Types of Constraints with Examples:

1. **NOT NULL**
    - Ensures that a column cannot have a NULL value.

Example:

```
 CREATE TABLE users (
  id INT NOT NULL,
  name VARCHAR(100) NOT NULL
);
```

    -
2. **UNIQUE**
    - Ensures all values in a column are unique.

Example:

```
 CREATE TABLE employees (
  email VARCHAR(100) UNIQUE
);
```

    -
3. **PRIMARY KEY**
    - Uniquely identifies each row in a table. It combines `NOT NULL` and `UNIQUE`.

Example:

```
 CREATE TABLE students (
  student_id INT PRIMARY KEY,
  name VARCHAR(100)
);
```

    -
4. **FOREIGN KEY**

- Ensures a value in one table corresponds to a value in another table.

Example:

```
CREATE TABLE orders (
 order_id INT PRIMARY KEY,
 customer_id INT,
 FOREIGN KEY (customer_id) REFERENCES customers(id)
);
```

5. **CHECK**
    - Ensures values in a column meet a specific condition.

Example:

```
CREATE TABLE products (
 price DECIMAL(10, 2) CHECK (price > 0)
);
```

6. **DEFAULT**
    - Assigns a default value if none is provided.

Example:

```
CREATE TABLE accounts (
 balance DECIMAL(10, 2) DEFAULT 0.00
);
```

Conclusion:

Constraints are essential for enforcing business rules at the database level. They provide a reliable way to maintain the quality and consistency of the data by preventing errors and invalid data entries.

3.Why would you apply the NOT NULL constraint to a column? Can a primary key contain NULL values? Justify your answer.

Sure! Here's a more natural, human-like answer:

Why would you use the NOT NULL constraint on a column?

You'd use the `NOT NULL` constraint when you want to **make sure a column always has a value**. For example, if you're creating a table for users, you'd probably want the **username or email** to never be empty. It helps make sure your data is complete and reliable.

Imagine someone signing up on a website without entering their name or email — that wouldn't make much sense, right? That's exactly what `NOT NULL` helps prevent.

Can a primary key contain NULL values?

No, a primary key can't have NULL values.

That's because a **primary key is meant to uniquely identify each row** in a table. If it had NULL (which means "unknown" or "empty"), the database wouldn't know how to identify that row properly. Every row needs a value in the primary key column — and that value must be unique and not empty.

So even if you don't write `NOT NULL` with a primary key, the database automatically treats it as `NOT NULL`.

Simple Example:

Let's say you have a `students` table:

```
CREATE TABLE students (
  student_id INT PRIMARY KEY,
  name VARCHAR(100) NOT NULL
);
```

- `student_id` is the **primary key**, so it **can't be empty** and must be **unique**.
- `name` is marked `NOT NULL`, so every student must have a name.

Let me know if you'd like more examples or a simple analogy to remember this!

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

Sure! Here's a clear and human-style explanation of how to **add or remove constraints** on an existing SQL table, with examples.

✅ Steps to Add or Remove Constraints in SQL

When working with an existing table, you can use the `ALTER TABLE` command to **add** or **remove** constraints.

🟢 1. Adding a Constraint

✅ General Syntax:

```
ALTER TABLE table_name
ADD CONSTRAINT constraint_name constraint_type (column_name);
```

📌 Example: Add a UNIQUE Constraint

Suppose you have a table called `employees` and you want to make sure the `email` column is unique.

```
ALTER TABLE employees
ADD CONSTRAINT unique_email UNIQUE (email);
```

🔍 This makes sure no two employees can have the same email.

🔴 2. Removing a Constraint

✅ General Syntax:

ALTER TABLE table_name
DROP CONSTRAINT constraint_name;

⚠️ You **must know the name** of the constraint to drop it.

📌 Example: Remove the UNIQUE Constraint

Let's remove the `unique_email` constraint we added earlier:

ALTER TABLE employees
DROP CONSTRAINT unique_email;

🔍 How to Find Constraint Names (if you forget them)

In systems like **PostgreSQL** or **SQL Server**, you can run:

SELECT * FROM information_schema.table_constraints
WHERE table_name = 'employees';

Or in **MySQL**, use:

SHOW CREATE TABLE employees;

✅ Summary

Action	SQL Command Example
Add constraint	`ALTER TABLE employees ADD CONSTRAINT unique_email UNIQUE (email);`
Drop constraint	`ALTER TABLE employees DROP CONSTRAINT unique_email;`

Let me know which database you're using (MySQL, PostgreSQL, SQL Server, etc.), and I can tailor the syntax exactly for that!

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.

Sure! Here's a clear and simple explanation:

❌ What Happens If You Break a Constraint?

When you try to **insert**, **update**, or **delete** data that goes against a constraint (like `NOT NULL`, `UNIQUE`, `PRIMARY KEY`, etc.), the **database will block the action** and give you an **error message**.

🎯 Why?

Because constraints are rules designed to **protect your data** and **keep it accurate**.
Violating them means you're trying to enter data that doesn't follow those rules.

🧨 Consequences by Action Type:

1. Inserting Invalid Data

If you try to insert data that breaks a rule, the insert will fail.

📌 Example:

INSERT INTO students (student_id, name) VALUES (NULL, 'Asha');

🔴 This will fail if `student_id` is a `PRIMARY KEY` — because it **can't be NULL**.

💥 **Error Message:**

ERROR: null value in column "student_id" violates not-null constraint

---

2. Updating Data to Invalid Value

If you try to update a column in a way that breaks a constraint, it will also fail.

📌 Example:

UPDATE students SET student_id = 101 WHERE student_id = 102;

🔴 If `101` already exists, and `student_id` is a `PRIMARY KEY`, this will break the **UNIQUE** rule.

💥 **Error Message:**

ERROR: duplicate key value violates unique constraint "students_pkey"

---

3. Deleting Data That Is Referenced

If a row is linked by a **foreign key**, and you try to delete it, it will fail unless handled properly.

📌 Example:
You have two tables:

- `students` (with `student_id` as primary key)
- `marks` (with `student_id` as foreign key)

DELETE FROM students WHERE student_id = 101;

🔴 If that student has marks recorded in the `marks` table, this delete will fail.

💥 **Error Message:**

ERROR: update or delete on table "students" violates foreign key constraint on table "marks"

✅ Summary Table

Action	Constraint Violated	What Happens	Example Error Message
Insert	NOT NULL	Insert fails	`null value violates not-null constraint`
Insert	UNIQUE	Duplicate value blocked	`duplicate key value violates unique constraint`
Update	PRIMARY/UNIQUE	Value already exists	`duplicate key value violates unique constraint`
Delete	FOREIGN KEY	Can't delete if data is linked	`violates foreign key constraint`

Let me know if you want me to show this with a live example in code or diagram!

6. 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 key The price should have a default value of 50.00

A. ALTER TABLE products
ADD CONSTRAINT pk_product_id PRIMARY KEY (product_id);

ALTER TABLE products
ALTER COLUMN price SET DEFAULT 50.00;

7. You have two tables:
Write a query to fetch the student_name and class_name for each student using an INNER JOIN

A. select students.student_name,classes.class_name
From students
Inner join classes on
Classes.class_id = students.class_id;

8. Consider the following three tables:
Write a query that shows all order_id, customer_name, and product_name, ensuring that all products are listed even if they are not associated with an order Hint: (use INNER JOIN and LEFT JOIN)

A.select sales.order_id,customers.customer_name,order_details.quantity
From sales
left join order_details on
Order_details.order_id = sales.order_id
Inner join products on
Products.order_id = sales.order_id;

9.Write a query to find the total sales amount for each product using an INNER JOIN and the SUM() function.

A.select sum(sales.amount) as total_sales,products.product_name
from sales;
inner join products on
products.product_id = sales.product_id
group by products.product_name

10. Write a query to display the order_id, customer_name, and the quantity of products ordered by each customer using an INNER JOIN between all three tables. Note - The above-mentioned questions don't require any dataset.

A. SELECT o.order_id, c.customer_name, p.product_name
FROM Orders o
INNER JOIN Customers c
    ON o.customer_id = c.customer_id
INNER JOIN Products p
    ON o.product_id = p.product_id;

SQL commands

1-Identify the primary keys and foreign keys in maven movies db. Discuss the differences.

A.SELECT TABLE_NAME, COLUMN_NAME
FROM INFORMATION_SCHEMA.KEY_COLUMN_USAGE
WHERE TABLE_SCHEMA = 'mavenmovies'
  AND CONSTRAINT_NAME = 'PRIMARY';

SELECT TABLE_NAME, COLUMN_NAME, CONSTRAINT_NAME,
REFERENCED_TABLE_NAME, REFERENCED_COLUMN_NAME
FROM INFORMATION_SCHEMA.KEY_COLUMN_USAGE
WHERE TABLE_SCHEMA = 'mavenmovies'
  AND REFERENCED_TABLE_NAME IS NOT NULL;

Primary Key (PK)

- A **Primary Key** is a column (or set of columns) that **uniquely identifies each row** in a table.
- It **cannot be NULL**.
- Each table can have **only one primary key** (but it can be made up of multiple columns = composite key).
- Example in Maven Movies:
    - In the **`actor`** table, `actor_id` is the **primary key**. Every actor has a unique `actor_id`.

Foreign Key (FK)

- A **Foreign Key** is a column (or set of columns) that **creates a link between two tables**.
- It refers to the **primary key in another table**.
- It can **contain duplicate values** and **can be NULL** (depending on design).
- Example in Maven Movies:
    - In the **`film_actor`** table, `actor_id` is a **foreign key** that references `actor(actor_id)`.
    - This means: the `film_actor` table can only use actor IDs that exist in the `actor` table.

2- List all details of actors

```
SELECT *
FROM actor;
```

3.-List all customer information from DB.

```
A.SELECT *
FROM customer;
```

4.-List different countries.

```
A.SELECT *
FROM country;
```

5.Display all active customers.

```
A.SELECT *
FROM customer
WHERE active = 1;
```

6 -List of all rental IDs for customer with ID 1.

```
A.SELECT rental_id
FROM rental
WHERE customer_id = 1;
```

7 - Display all the films whose rental duration is greater than 5.

A.SELECT *
FROM film
WHERE rental_duration > 5;

8 - List the total number of films whose replacement cost is greater than $15 and less than $20.

A.SELECT COUNT(*) AS total_films
FROM film
WHERE replacement_cost > 15 AND replacement_cost < 20;

9 - Display the count of unique first names of actors.

SELECT COUNT(DISTINCT first_name) AS unique_first_names
FROM actor;

10- Display the first 10 records from the customer table .

A.SELECT *
FROM customer
LIMIT 10;

11 - Display the first 3 records from the customer table whose first name starts with ‘b’.

A.SELECT *
FROM customer
WHERE first_name LIKE 'B%'
LIMIT 3;

12 -Display the names of the first 5 movies which are rated as ‘G’.

A.SELECT title
FROM film
WHERE rating = 'G'
LIMIT 5;

13-Find all customers whose first name starts with "a".

A.SELECT *
FROM customer
WHERE first_name LIKE 'A%';

14- Find all customers whose first name ends with "a".

A.SELECT *
FROM customer
WHERE first_name LIKE '%a';

15- Display the list of first 4 cities which start and end with ‘a’ .

A.SELECT *
FROM city
WHERE city LIKE 'A%'
  AND city LIKE '%a'
LIMIT 4;

16- Find all customers whose first name have "NI" in any position.

A.SELECT *
FROM customer
WHERE first_name LIKE '%NI%';

17- Find all customers whose first name have "r" in the second position .

A.SELECT *
FROM customer
WHERE first_name LIKE '_r%';

18 - Find all customers whose first name starts with "a" and are at least 5 characters in length.

A.SELECT *
FROM customer
WHERE first_name LIKE 'A%'
  AND LENGTH(first_name) >= 5;

19- Find all customers whose first name starts with "a" and ends with "o".

A.SELECT *
FROM customer
WHERE first_name LIKE 'A%'
  AND first_name LIKE '%o';

20 - Get the films with pg and pg-13 rating using IN operator.

A.SELECT *
FROM film
WHERE rating IN ('PG', 'PG-13');

21 - Get the films with length between 50 to 100 using between operator.

A.SELECT *
FROM film
WHERE length BETWEEN 50 AND 100;

22 - Get the top 50 actors using limit operator.

A.SELECT *
FROM actor
LIMIT 50;

23 - Get the distinct film ids from inventory table.

A.SELECT DISTINCT film_id
FROM inventory;

Basic Aggregate Functions:

1.Retrieve the total number of rentals made in the Sakila database.

A.SELECT COUNT(*) AS total_rentals
FROM rental;

2. Find the average rental duration (in days) of movies rented from the Sakila database. Hint: Utilize the AVG() function.

A.SELECT AVG(rental_duration) AS agrent
FROM film;

3.Display the first name and last name of customers in uppercase.

A.SELECT UPPER(first_name), UPPER(last_name)
FROM customer;

4.Extract the month from the rental date and display it alongside the rental ID.

A.SELECT MONTH(rental_date) AS rental_month, rental_id
FROM rental;

5.Retrieve the count of rentals for each customer (display customer ID and the count of rentals). Hint: Use COUNT () in conjunction with GROUP BY.

A.SELECT customer_id, COUNT(rental_id) AS total_rentals
FROM rental
GROUP BY customer_id;

6. Find the total revenue generated by each store. Hint: Combine SUM() and GROUP BY

A.SELECT s.store_id, SUM(p.amount) AS total_revenue
FROM payment p
JOIN staff st ON p.staff_id = st.staff_id
JOIN store s ON st.store_id = s.store_id
GROUP BY s.store_id;

7. Determine the total number of rentals for each category of movies. Hint: JOIN film_category, film, and rental tables, then use cOUNT () and GROUP BY.

A.SELECT c.name AS category, COUNT(r.rental_id) AS total_rentals
FROM category c
JOIN film_category fc ON c.category_id = fc.category_id
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
GROUP BY c.name;

8. Find the average rental rate of movies in each language. Hint: JOIN film and language tables, then use AVG () and GROUP BY.

A.SELECT language.name AS language, AVG(film.rental_rate) AS avg_rental_rate
FROM film
JOIN language ON film.language_id = language.language_id
GROUP BY language.name;

9.Display the title of the movie, customer s first name, and last name who rented it. Hint: Use JOIN between the film, inventory, rental, and customer tables.

A.SELECT f.title AS movie_title,
    c.first_name,
    c.last_name
FROM film f
JOIN inventory i ON f.film_id = i.film_id
JOIN rental r ON i.inventory_id = r.inventory_id
JOIN customer c ON r.customer_id = c.customer_id;

10. Retrieve the names of all actors who have appeared in the film "Gone with the Wind." Hint: Use JOIN between the film actor, film, and actor tables.

A.SELECT a.first_name, a.last_name
FROM actor a
JOIN film_actor fa ON a.actor_id = fa.actor_id
JOIN film f ON fa.film_id = f.film_id
WHERE f.title = 'GONE WITH THE WIND';

11. Retrieve the customer names along with the total amount they've spent on rentals. Hint: JOIN customer, payment, and rental tables, then use SUM() and GROUP BY.

```
a.SELECT c.first_name,
      c.last_name,
      SUM(p.amount) AS total_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, c.first_name, c.last_name;
```

12. List the titles of movies rented by each customer in a particular city (e.g., 'London'). Hint: JOIN customer, address, city, rental, inventory, and film tables, then use GROUP BY

```
A.SELECT c.first_name,
      c.last_name,
      ci.city,
      f.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 c.first_name, c.last_name, ci.city, f.title;
```

13.Display the top 5 rented movies along with the number of times they've been rented. Hint: JOIN film, inventory, and rental tables, then use COUNT () and GROUP BY, and limit the results.

```
A.SELECT f.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.title
ORDER BY rental_count DESC
LIMIT 5;
```

14. Determine the customers who have rented movies from both stores (store ID 1 and store ID 2). Hint: Use JOINS with rental, inventory, and customer tables and consider COUNT() and GROUP BY.

```
A.SELECT c.customer_id,
      c.first_name,
```

```
    c.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
WHERE i.store_id IN (1, 2)
GROUP BY c.customer_id, c.first_name, c.last_name
HAVING COUNT(DISTINCT i.store_id) = 2;
```

1. Rank the customers based on the total amount they've spent on rentals.

```
A.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_position
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 total_spent DESC;
```

2.Calculate the cumulative revenue generated by each film over time.

```
A.SELECT f.title,
    r.rental_date,
    SUM(p.amount) OVER (
        PARTITION BY f.film_id
        ORDER BY r.rental_date
        ROWS UNBOUNDED PRECEDING
    ) AS cumulative_revenue
FROM film f
JOIN inventory i ON f.film_id = i.film_id
JOIN rental r ON i.inventory_id = r.inventory_id
JOIN payment p ON r.rental_id = p.rental_id
ORDER BY f.title, r.rental_date;
```

3. Determine the average rental duration for each film, considering films with similar lengths.

```
A.SELECT f.length,
    AVG(f.rental_duration) AS avg_rental_duration,
    COUNT(f.film_id) AS films_count
FROM film f
GROUP BY f.length
ORDER BY f.length;
```

4. Identify the top 3 films in each category based on their rental counts.

```
A.SELECT c.name AS category,
```

```
    f.title AS film_title,
    COUNT(r.rental_id) AS rental_count,
    RANK() OVER (PARTITION BY c.name ORDER BY COUNT(r.rental_id) DESC) AS
rank_in_category
FROM category c
JOIN film_category fc ON c.category_id = fc.category_id
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
GROUP BY c.name, f.title
QUALIFY RANK() OVER (PARTITION BY c.name ORDER BY COUNT(r.rental_id) DESC)
<= 3;
```

5.Calculate the difference in rental counts between each customer's total rentals and the average rentals across all customers.

```
A.SELECT c.customer_id,
    c.first_name,
    c.last_name,
    COUNT(r.rental_id) AS total_rentals,
    (COUNT(r.rental_id) - avg_data.avg_rentals) AS diff_from_avg
FROM customer c
JOIN rental r ON c.customer_id = r.customer_id
JOIN (
   SELECT AVG(rental_count) AS avg_rentals
   FROM (
      SELECT customer_id, COUNT(rental_id) AS rental_count
      FROM rental
      GROUP BY customer_id
   ) sub
) avg_data
GROUP BY c.customer_id, c.first_name, c.last_name, avg_data.avg_rentals
ORDER BY diff_from_avg DESC;
```

6. Find the monthly revenue trend for the entire rental store over time.

```
A.SELECT YEAR(p.payment_date) AS year,
    MONTH(p.payment_date) AS month,
    SUM(p.amount) AS total_revenue
FROM payment p
GROUP BY YEAR(p.payment_date), MONTH(p.payment_date)
ORDER BY year, month;
```

7. Identify the customers whose total spending on rentals falls within the top 20% of all customers.

```
A.SELECT customer_id,
```

```
    first_name,
    last_name,
    total_spent
FROM (
  SELECT c.customer_id,
      c.first_name,
      c.last_name,
      SUM(p.amount) AS total_spent,
      NTILE(5) OVER (ORDER BY SUM(p.amount) DESC) AS spending_percentile
  FROM customer c
  JOIN payment p ON c.customer_id = p.customer_id
  GROUP BY c.customer_id, c.first_name, c.last_name
) ranked
WHERE spending_percentile = 1
ORDER BY total_spent DESC;
```

8. Calculate the running total of rentals per category, ordered by rental count.

```
A.SELECT c.name AS category,
    COUNT(r.rental_id) AS rental_count
FROM category c
JOIN film_category fc ON c.category_id = fc.category_id
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
GROUP BY c.name
ORDER BY rental_count DESC;
```

9. Find the films that have been rented less than the average rental count for their respective categories.

```
A.SELECT film_data.title,
    film_data.category,
    film_data.film_rentals,
    cat_avg.avg_rentals AS category_avg
FROM (
  SELECT f.film_id,
      f.title,
      c.name AS category,
      COUNT(r.rental_id) AS film_rentals
  FROM film f
  JOIN film_category fc ON f.film_id = fc.film_id
  JOIN category c ON fc.category_id = c.category_id
  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, f.title, c.name
) film_data
```

```
JOIN (
    SELECT c.category_id,
           AVG(film_count) AS avg_rentals
    FROM (
        SELECT f.film_id,
               fc.category_id,
               COUNT(r.rental_id) AS film_count
        FROM film f
        JOIN film_category fc ON f.film_id = fc.film_id
        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, fc.category_id
    ) sub
    JOIN category c ON sub.category_id = c.category_id
    GROUP BY c.category_id
) cat_avg
ON film_data.category = (SELECT name FROM category WHERE category_id =
cat_avg.category_id)
WHERE film_data.film_rentals < cat_avg.avg_rentals;
```