### 

#### **Normalization**

Normalization is the process of organizing a relational database in such a way that it reduces redundancy and dependency by dividing large tables into smaller, more manageable ones. The goal is to ensure that the data is stored in a way that prevents anomalies (insertion, update, or deletion anomalies) and maintains data integrity.

**Redundancy** in a database refers to the **unnecessary duplication of data** across one or more tables. When the same piece of information is repeated multiple times, it can lead to inefficiency and potential data inconsistency.

Normalization follows a set of **Normal Forms (NF)** that progressively reduce redundancy by breaking down large tables into smaller ones. Each **Normal Form** introduces stricter rules and guidelines to organize the data.

#### **Denormalization**

Denormalization is the opposite process, where normalized tables are combined back together to improve read performance, usually for reporting purposes or when the application is read-heavy. While it increases redundancy and can lead to update anomalies, denormalization can make querying faster by reducing the number of joins needed.

Denormalization is often used when:

* Query performance is critical.
* Data redundancy is acceptable.
* The database is read-intensive.

<https://www.simplilearn.com/tutorials/sql-tutorial/what-is-normalization-in-sql>

<https://en.wikipedia.org/wiki/Database_normalization>

To demonstrate **normalizing** and **denormalizing** data in PostgreSQL, we need to start by creating a sample database with a normalized structure.

### **Step 1: Create Normalized Tables**

Normalized tables separate data into different entities, removing redundancy. For this example, we’ll create tables for patients and their visits.

CREATE TABLE patient\_data (

patient\_id INT,

patient\_name VARCHAR(100),

date\_of\_birth VARCHAR(30),

date\_of\_visit DATE,

diagnosis\_code VARCHAR(10),

visit\_type VARCHAR(20),

age INT

);

INSERT INTO patient\_data (patient\_id, patient\_name, date\_of\_birth, date\_of\_visit, diagnosis\_code, visit\_type, age)

VALUES

(1, 'John Doe', '1980-05-15', '2024-03-10', 'A01', 'Checkup', 44),

(2, 'JANE Smith', '1990-08-22', '2024-03-11', 'B02', 'Follow-up', 34),

(3, 'Sam Brown', NULL, '2024-03-12', 'C03', 'Emergency', NULL), -- Missing date\_of\_birth and age

(4, 'John doe', '1980-05-15', '2024-03-10', 'A01', 'Checkup', 44), -- Duplicate row

(5, 'Alice White', '1985-07-30', '2024-03-10', 'D04', 'Checkup', 39), -- Missing date\_of\_visit

(6, 'Bob Green', '1978-12-01', '2024-03-14', 'E05', 'Follow-up', 46),

(7, 'Charlie Black', '1982-10-10', '2024-03-10', 'F06', 'Checkup', -42),

(8, 'Daniel Blue', '1995-01-20', '2024-03-15', 'G07', 'Follow-up', 29),

(9, 'Eve White', '2000-02-28', '2024-03-16', 'A01', 'Checkup', 24);

SELECT \*

FROM sample\_data

WHERE id IS NULL OR value is NULL;

SELECT COALESCE(value, 'Apple') FROM sample\_data;

UPDATE sample\_data

SET value = 'Apple'

WHERE value IS NULL;

select value, INITCAP(value)

from sample\_data

UPDATE sample\_data

SET value = INITCAP(value);

CREATE TABLE sample\_data (id INT, value VARCHAR(50));

INSERT INTO sample\_data (id, value) VALUES (1, 'apple'), (2, 'Apple'), (3, 'APPLE'), (4, NULL);

-- Detect NULL values

SELECT \* FROM sample\_data WHERE value IS NULL;

-- Replace NULL with 0

SELECT id, COALESCE(value, 'Orange') AS value FROM sample\_data;

UPDATE patient\_data

SET date\_of\_birth = 'Orange'

WHERE date\_of\_birth IS NULL;

UPDATE sample\_data

SET value = UPPER(value);

-- Detect NULL values

SELECT \* FROM sample\_data WHERE value IS NULL;

-- Replace NULL with 0

SELECT id, COALESCE(value, 0) AS value FROM sample\_data;

--1. Detecting NULL Values

--SELECT \* FROM your\_table WHERE your\_column IS NULL;

SELECT \*

FROM patient\_data

WHERE date\_of\_birth IS NULL OR date\_of\_visit is NULL OR diagnosis\_code IS NULL OR visit\_type IS NULL OR age is NULL;

--2.Techniques to Handle Missing Data

--SELECT COALESCE(your\_column, 'default\_value') FROM your\_table;

SELECT COALESCE(date\_of\_birth, '2000-01-01') FROM patient\_data;

UPDATE patient\_data

SET date\_of\_birth = '2024-01-01'

WHERE date\_of\_birth IS NULL;

-- UPDATE your\_table

-- SET your\_column = (SELECT AVG(your\_column) FROM your\_table)

-- WHERE your\_column IS NULL;

UPDATE patient\_data

SET age = (SELECT AVG(AGE) FROM patient\_data)

WHERE age IS NULL;

UPDATE patient\_data

SET age = (SELECT AVG(AGE) FROM patient\_data)

WHERE age IS NULL;

-- Removing Duplicates

-- 1. Identifying Duplicates

-- SELECT your\_column, COUNT(\*)

-- FROM your\_table

-- GROUP BY your\_column

-- HAVING COUNT(\*) > 1;

INSERT INTO patient\_data (patient\_id, patient\_name, date\_of\_birth, date\_of\_visit, diagnosis\_code, visit\_type, age)

VALUES

(1, 'John Doe', '1980-05-15', '2024-12-10', 'A01', 'Checkup', 44),

(1, 'John Doe', '1980-05-15', '2024-12-12', 'A01', 'Checkup', 44)

;

SELECT patient\_name, date\_of\_birth, COUNT(\*)

FROM patient\_data

GROUP BY patient\_name, date\_of\_birth

HAVING COUNT(\*) > 1;

SELECT \*

FROM patient\_data

WHERE patient\_name = 'John Doe' and date\_of\_birth = '1980-05-15'

--2. Deleting Duplicates

-- WITH CTE AS (

-- SELECT your\_column, ROW\_NUMBER() OVER (PARTITION BY your\_column ORDER BY your\_column) AS rn

-- FROM your\_table

-- )

-- DELETE FROM CTE WHERE rn > 1;

WITH CTE AS (

SELECT patient\_id, date\_of\_visit,

ROW\_NUMBER() OVER (PARTITION BY patient\_name, date\_of\_birth ORDER BY date\_of\_visit) AS row\_num

FROM patient\_data

)

DELETE FROM patient\_data

WHERE patient\_id IN (SELECT patient\_id FROM CTE WHERE row\_num > 1);

-- Correcting Inconsistent Data

-- 1. Standardizing Data Formats

-- Convert date formats

--SELECT CAST(your\_date\_column AS DATE) FROM your\_table;

-- Convert strings to uppercase

-- SELECT UPPER(your\_string\_column) FROM your\_table;

-- SELECT CONVERT(target\_data\_type, expression, [style])

UPDATE patient\_data

SET date\_of\_visit = date\_of\_visit::DATE;

select patient\_name, INITCAP(patient\_name), UPPER(patient\_name) as Upper\_

from patient\_data

UPDATE patient\_data

SET patient\_name = INITCAP(patient\_name);

----------------------------------------------------

sql

Копировать код

-- Create Patients Table (1st Normal Form)

CREATE TABLE patients (

patient\_id SERIAL PRIMARY KEY,

first\_name VARCHAR(50),

last\_name VARCHAR(50),

date\_of\_birth DATE

);

-- Create Visits Table (2nd Normal Form, normalized from the patient data)

CREATE TABLE visits (

visit\_id SERIAL PRIMARY KEY,

patient\_id INT REFERENCES patients(patient\_id),

visit\_date DATE,

diagnosis VARCHAR(100)

);

Here, we have two tables:

1. patients table stores patient details.
2. visits table stores patient visits with a foreign key reference to the patients table.

### **Step 2: Insert Sample Data**

Now, let's insert some sample data into these tables.

sql

Копировать код

-- Insert data into patients table

INSERT INTO patients (first\_name, last\_name, date\_of\_birth)

VALUES

('John', 'Doe', '1985-02-20'),

('Jane', 'Smith', '1990-05-15'),

('Emily', 'Johnson', '1982-08-25');

-- Insert data into visits table

INSERT INTO visits (patient\_id, visit\_date, diagnosis)

VALUES

(1, '2024-01-10', 'Flu'),

(1, '2024-02-20', 'Cold'),

(2, '2024-03-05', 'Headache'),

(3, '2024-04-10', 'Diabetes');

### **Step 3: Denormalizing Data**

Denormalization involves combining data from multiple tables into one, which may introduce redundancy but can improve performance in some cases, especially when we query large datasets frequently.

To denormalize the patients and visits data, we can create a single table that stores both patient and visit information in one place:

sql

Копировать код

-- Create Denormalized Table (Combining Patients and Visits)

CREATE TABLE denormalized\_patient\_visits AS

SELECT

p.patient\_id,

p.first\_name,

p.last\_name,

p.date\_of\_birth,

v.visit\_id,

v.visit\_date,

v.diagnosis

FROM

patients p

JOIN

visits v ON p.patient\_id = v.patient\_id;

This table contains both patient information and their visit history combined into one table.

### **Step 4: Query Normalized vs. Denormalized Data**

Now that we have both normalized and denormalized data, we can query them to see the difference.

**Query the Normalized Data:**

sql

Копировать код

-- Query Normalized Data

SELECT \* FROM patients;

SELECT \* FROM visits;

**Query the Denormalized Data:**

sql

Копировать код

-- Query Denormalized Data

SELECT \* FROM denormalized\_patient\_visits;

**Normalized Data**: The data is split across two tables (patients and visits), avoiding redundancy.

**Denormalized Data**: The data from both tables is merged into one, which makes it easier to access all information in a single query. However, this increases redundancy (e.g., the same patient information is repeated for every visit).

#### **Normalization**

Normalization is the process of organizing a relational database in such a way that it reduces redundancy and dependency by dividing large tables into smaller, more manageable ones. The goal is to ensure that the data is stored in a way that prevents anomalies (insertion, update, or deletion anomalies) and maintains data integrity.

Normalization follows a set of **Normal Forms (NF)** that progressively reduce redundancy by breaking down large tables into smaller ones. Each **Normal Form** introduces stricter rules and guidelines to organize the data.

#### **Denormalization**

Denormalization is the opposite process, where normalized tables are combined back together to improve read performance, usually for reporting purposes or when the application is read-heavy. While it increases redundancy and can lead to update anomalies, denormalization can make querying faster by reducing the number of joins needed.

Denormalization is often used when:

* Query performance is critical.
* Data redundancy is acceptable.
* The database is read-intensive.

<https://www.simplilearn.com/tutorials/sql-tutorial/what-is-normalization-in-sql>

<https://en.wikipedia.org/wiki/Database_normalization>

The PIVOT() function in SQL Server is used to transform or rotate data from a **long format (row-based)** to a **wide format (column-based)**. Pivoting is often useful when you want to summarize or aggregate your data in a more readable or report-friendly format. This function is especially handy when you need to perform operations such as summarizing sales by category, showing average performance by region, or comparing values across different time periods.

**Pivoting**: The PIVOT operation transforms rows into columns, often using aggregate functions like SUM(), AVG(), etc.

**Unpivoting**: The UNPIVOT operation reverses pivoting by transforming columns back into rows.

**T-SQL Syntax** (SQL Server): PIVOT and UNPIVOT are built-in functions used for these operations. In databases like PostgreSQL, these operations are typically done using CASE expressions or UNION ALL.

<https://regexone.com/>

### **1. PRIMARY KEY Constraint**

A **PRIMARY KEY** is a column (or a combination of columns) in a database table that uniquely identifies each row in the table. It ensures that the values in the primary key column(s) are **unique** and **not null**.

#### **Why We Use PRIMARY KEY:**

* **Uniqueness**: Ensures that every row in the table has a unique identifier.
* **Consistency**: Helps maintain consistency within the database by preventing duplicate entries.
* **Indexing**: The primary key automatically creates a unique index on the column(s), speeding up query performance.
* **Referential Integrity**: It is often used in combination with foreign keys to link tables together.

#### **Example:**

sql

Копировать код

-- Adding a primary key to the customers table

ALTER TABLE customers

ADD PRIMARY KEY (customer\_id);

* In this example, customer\_id is designated as the primary key, ensuring that each customer record has a unique identifier.

### **2. FOREIGN KEY Constraint**

A **FOREIGN KEY** is a column (or a set of columns) in one table that establishes a link between the data in two tables. The foreign key in the child table points to the primary key in the parent table. This relationship ensures that the data in the child table corresponds to valid data in the parent table.

#### **Why We Use FOREIGN KEY:**

* **Referential Integrity**: Ensures that a record in the child table cannot exist without a corresponding record in the parent table.
* **Data Validation**: Prevents inserting or updating records with invalid references.
* **Cascading Updates/Deletes**: Can be configured to automatically update or delete related records when the parent record changes, helping maintain consistency.

#### **Example:**

sql

Копировать код

-- Adding a foreign key to the orders table

ALTER TABLE orders

ADD FOREIGN KEY (customer\_id) REFERENCES customers(customer\_id);

* In this example, customer\_id in the orders table refers to the customer\_id in the customers table, ensuring that each order is linked to an existing customer.

### **3. UNIQUE Constraint**

A **UNIQUE** constraint ensures that all values in a column are different from each other. It allows **NULL** values but ensures that if there are any non-NULL values, they must be distinct across the table.

#### **Why We Use UNIQUE:**

* **Avoid Duplicates**: Prevents duplicate entries in a column where uniqueness is required (e.g., email addresses, usernames).
* **Data Quality**: Ensures that key attributes such as email, social security numbers, or any other identifiers remain unique.
* **Flexible Validation**: Unlike the primary key, it allows for the presence of NULL values, but guarantees that other values are unique.

#### **Example:**

sql

Копировать код

-- Adding a unique constraint to the email column in the users table

ALTER TABLE users

ADD CONSTRAINT unique\_email UNIQUE (email);

* This constraint ensures that no two users can have the same email address, improving data quality and reducing the possibility of conflicts.

### **4. CHECK Constraint**

A **CHECK** constraint is used to ensure that the values in a column satisfy a specific condition or rule. The condition can be a simple or complex expression that must evaluate to true for every row in the table.

#### **Why We Use CHECK:**

* **Data Validation**: Ensures that data inserted into the table meets certain conditions (e.g., prices cannot be negative, ages must be greater than zero).
* **Business Logic Enforcement**: Helps enforce rules directly in the database that reflect business logic (e.g., ensuring that a product's price is greater than zero).
* **Consistency**: Ensures that only valid data is entered, improving the overall data integrity.

#### **Example:**

sql

Копировать код

-- Adding a check constraint to ensure price is greater than 0

ALTER TABLE products

ADD CONSTRAINT check\_price CHECK (price > 0);

* In this case, the constraint ensures that no product can have a price less than or equal to zero, enforcing business logic and improving the reliability of the data.

### **Summary of Why to Use Each Data Integrity Constraint**

| **Constraint** | **Purpose** | **Benefits** |
| --- | --- | --- |
| **PRIMARY KEY** | Ensures unique identification for each row. | Prevents duplicate rows, ensures uniqueness, and improves query performance. |
| **FOREIGN KEY** | Ensures referential integrity between related tables. | Prevents orphaned records, ensures that data remains consistent across related tables. |
| **UNIQUE** | Ensures that all values in a column are unique. | Prevents duplicate values in key attributes such as email, ID numbers, etc. |
| **CHECK** | Ensures that the values in a column meet specific conditions. | Enforces business rules and ensures that invalid data cannot be inserted. |

-- Create Patients Table (1st Normal Form)

CREATE TABLE patients (

patient\_id SERIAL PRIMARY KEY,

first\_name VARCHAR(50),

last\_name VARCHAR(50),

date\_of\_birth DATE

);

-- Create Visits Table (2nd Normal Form, normalized from the patient data)

CREATE TABLE visits (

visit\_id SERIAL PRIMARY KEY,

patient\_id INT REFERENCES patients(patient\_id), --FOREIGN KEY

visit\_date DATE,

diagnosis VARCHAR(100)

);

-- Insert data into patients table

INSERT INTO patients (first\_name, last\_name, date\_of\_birth)

VALUES

('John', 'Doe', '1985-02-20'),

('Jane', 'Smith', '1990-05-15'),

('Emily', 'Johnson', '1982-08-25');

-- Insert data into visits table

INSERT INTO visits (patient\_id, visit\_date, diagnosis)

VALUES

(1, '2024-01-10', 'Flu'),

(1, '2024-02-20', 'Cold'),

(2, '2024-03-05', 'Headache'),

(3, '2024-04-10', 'Diabetes');

-- Query Normalized Data

SELECT \* FROM patients;

SELECT \* FROM visits;

----------------------

-- Create Denormalized Table (Combining Patients and Visits)

CREATE TABLE denormalized\_patient\_visits AS

SELECT

p.patient\_id,

p.first\_name,

p.last\_name,

p.date\_of\_birth,

v.visit\_id,

v.visit\_date,

v.diagnosis

FROM

patients p

JOIN

visits v ON p.patient\_id = v.patient\_id;

-- Query Denormalized Data

SELECT \* FROM denormalized\_patient\_visits;

-- -- Drop the denormalized table

-- DROP TABLE denormalized\_patient\_visits;

-- -- Drop the patients and visits tables

-- DROP TABLE visits;

-- DROP TABLE patients;

---------------------------------------------------------

CREATE TABLE sales\_data (

product\_id INT,

category VARCHAR(50),

sales INT

);

-- Insert sample data into the sales\_data table

INSERT INTO sales\_data (product\_id, category, sales) VALUES

(1, 'Electronics', 500),

(1, 'Clothing', 300),

(1, 'Groceries', 150),

(2, 'Electronics', 1000),

(2, 'Clothing', 500),

(2, 'Groceries', 200),

(3, 'Electronics', 800),

(3, 'Clothing', 450),

(3, 'Groceries', 300);

SELECT product\_id, category, SUM(sales)

FROM sales\_data

GROUP BY product\_id, category

ORDER BY product\_id, category

SELECT DISTINCT category

FROM sales\_data

SELECT \*

FROM (

SELECT product\_id, category, sales

FROM sales\_data

) AS SourceTable

PIVOT (

SUM(sales)

FOR category IN ([Electronics], [Clothing], [Groceries])

) AS PivotTable;

-------------------------

CREATE TABLE users (

id INT PRIMARY KEY,

status VARCHAR(10)

);

INSERT INTO users (id, status)

VALUES

(1, 'A'),

(2, 'I'),

(3, 'X2'), -- Invalid status

(4, 'I'),

(5, Null), -- Invalid status

(6, 'Z'); -- Invalid status

SELECT

id,

CASE

WHEN status = 'A' THEN 'Active'

WHEN status = 'I' THEN 'Inactive'

ELSE 'Unknown'

END AS status

FROM users;

UPDATE users

SET status = CASE

WHEN status = 'A' THEN 'Active'

WHEN status = 'I' THEN 'Inactive'

ELSE 'Unknown'

END;

SELECT \* FROM users;

------------------------------

CREATE TABLE exercise\_data (

id INT,

date VARCHAR(50),

value INT,

status VARCHAR(1)

);

INSERT INTO exercise\_data (id, date, value, status) VALUES

(1, '2021-01-01', 10, 'A'),

(2, '2021-01-02', NULL, 'I'),

(3, '2021-01-03', -5, 'A'),

(4, '2021-01-04', 20, 'X');

--Task 1

-- Use CASE for conditional cleaning

SELECT

id,

CASE

WHEN status = 'A' THEN 'Active'

WHEN status = 'I' THEN 'Inactive'

ELSE 'Unknown'

END AS status

FROM exercise\_data;

--Task 2

-- Use REGEXP to replace patterns

--REGEXP\_REPLACE(source, pattern, replacement, [flags])

UPDATE exercise\_data

SET date = REGEXP\_REPLACE(date, '-', '/');

--Task 3

select \* from exercise\_data

UPDATE exercise\_data

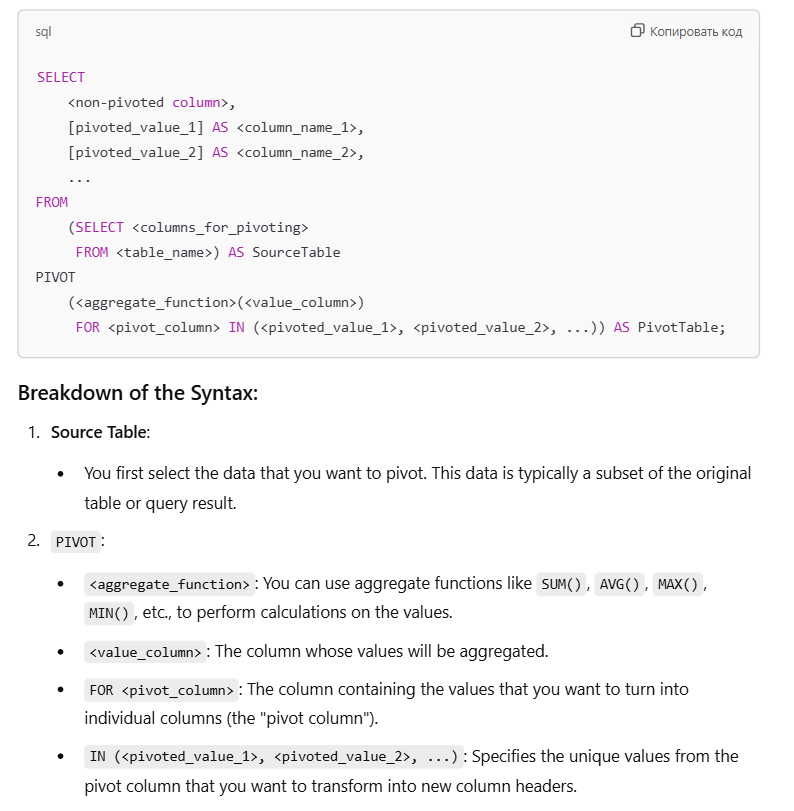
SET value = value \* (-1) + 1

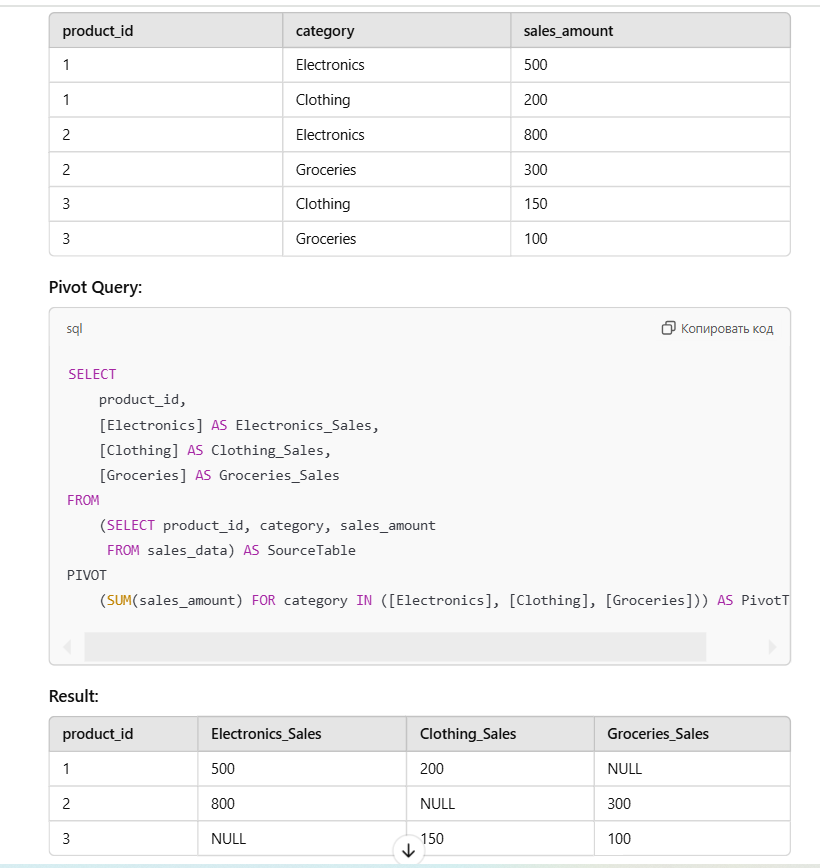
WHERE value < 0 OR value is ;

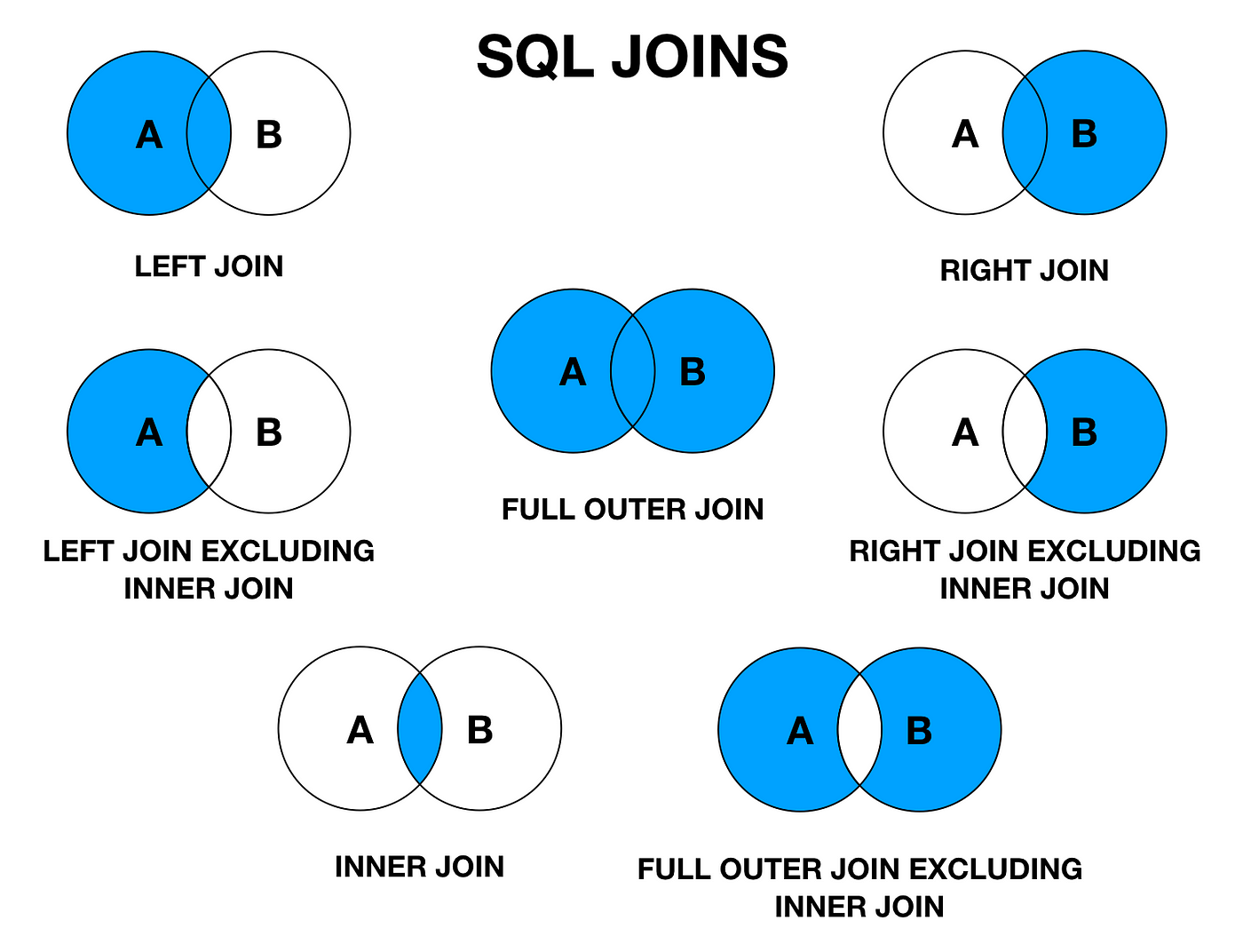
-- Add CHECK constraint

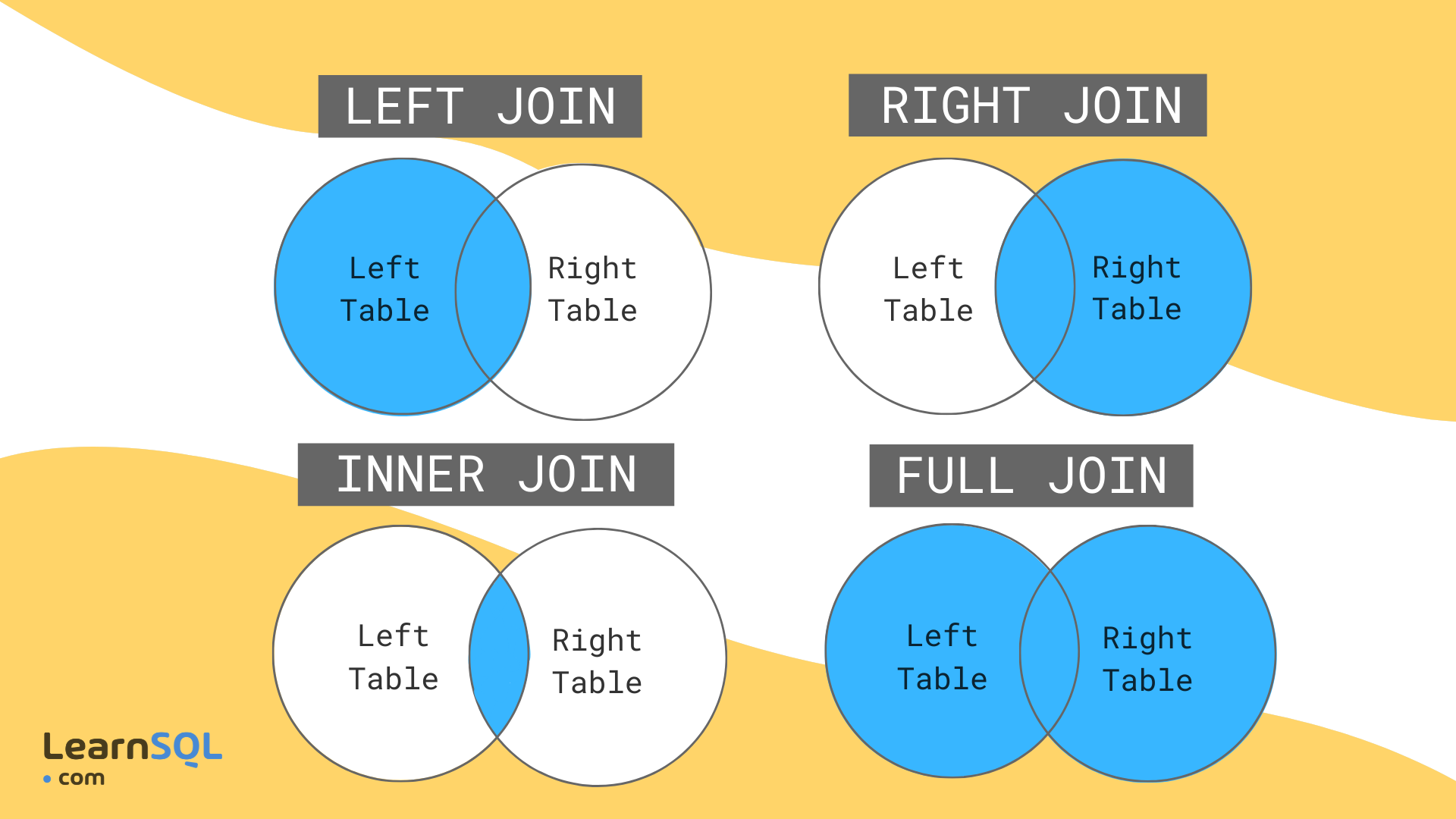
ALTER TABLE exercise\_data

ADD CONSTRAINT check\_value CHECK (value > 0);









**LEFT JOIN**: Use when you need to include all records from the left table and matching records from the right table.

* Example: List all customers with their orders, even those who haven't placed any orders.

**RIGHT JOIN**: Use when you need to include all records from the right table and matching records from the left table.

* Example: List all products with their suppliers, including those with no suppliers.

**FULL JOIN**: Use when you need to combine all records from both tables, showing unmatched rows as NULL in either table.

* Example: List all employees and all departments, including employees without departments and departments without employees.

A **CROSS JOIN** creates a Cartesian product of two tables, meaning it pairs each row from the first table with every row from the second table. It's useful when you need all possible combinations of rows from two tables.

CREATE TABLE employees (

employee\_id INT PRIMARY KEY,

name VARCHAR(50),

department\_id INT

);

CREATE TABLE departments (

department\_id INT PRIMARY KEY,

department\_name VARCHAR(50)

);

INSERT INTO employees (employee\_id, name, department\_id)

VALUES

(1, 'Alice', 1),

(2, 'Bob', 2),

(3, 'Charlie', 5),

(4, 'David', 3),

(5, 'Sam', 1),

(6, 'David', 3);

INSERT INTO departments (department\_id, department\_name)

VALUES

(4, 'Marketing');

SELECT e.employee\_id, e.name, d.department\_name

FROM employees e

INNER JOIN departments d

ON e.department\_id = d.department\_id;

SELECT e.employee\_id, e.name, d.department\_name

FROM employees e

LEFT JOIN departments d

ON e.department\_id = d.department\_id;

SELECT e.employee\_id, e.name, d.department\_name

FROM departments d

LEFT JOIN employees e

ON e.department\_id = d.department\_id;

SELECT e.employee\_id, e.name, d.department\_name

FROM employees e

RIGHT JOIN departments d

ON e.department\_id = d.department\_id;

SELECT e.employee\_id, e.name, d.department\_name

FROM departments d

RIGHT JOIN employees e

ON e.department\_id = d.department\_id;

SELECT e.employee\_id, e.name, d.department\_name

FROM employees e

FULL OUTER JOIN departments d

ON e.department\_id = d.department\_id;

SELECT e.employee\_id, e.name, d.department\_name

FROM employees e

CROSS JOIN departments d

order by e.employee\_id, e.name, d.department\_name