

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);
```

sql

 Копировать код

```
SELECT
    <non-pivoted column>,
    [pivoted_value_1] AS <column_name_1>,
    [pivoted_value_2] AS <column_name_2>,
    ...
FROM
    (SELECT <columns_for_pivoting>
     FROM <table_name>) AS SourceTable
PIVOT
    (<aggregate_function>(<value_column>)
     FOR <pivot_column> IN (<pivoted_value_1>, <pivoted_value_2>, ...)) AS PivotTable;
```

Breakdown of the Syntax:

1. Source Table:

- You first select the data that you want to pivot. This data is typically a subset of the original table or query result.

2. PIVOT :

- `<aggregate_function>` : You can use aggregate functions like `SUM()` , `AVG()` , `MAX()` , `MIN()` , etc., to perform calculations on the values.
- `<value_column>` : The column whose values will be aggregated.
- `FOR <pivot_column>` : The column containing the values that you want to turn into individual columns (the "pivot column").
- `IN (<pivoted_value_1>, <pivoted_value_2>, ...)` : Specifies the unique values from the pivot column that you want to transform into new column headers.

product_id	category	sales_amount
1	Electronics	500
1	Clothing	200
2	Electronics	800
2	Groceries	300
3	Clothing	150
3	Groceries	100

Pivot Query:

sql

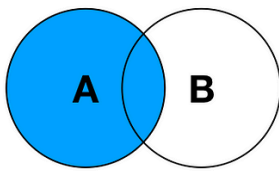
Копировать код

```
SELECT
    product_id,
    [Electronics] AS Electronics_Sales,
    [Clothing] AS Clothing_Sales,
    [Groceries] AS Groceries_Sales
FROM
    (SELECT product_id, category, sales_amount
     FROM sales_data) AS SourceTable
PIVOT
    (SUM(sales_amount) FOR category IN ([Electronics], [Clothing], [Groceries])) AS PivotT
```

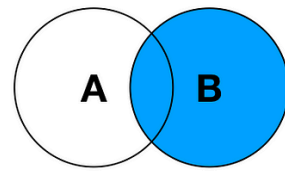
Result:

product_id	Electronics_Sales	Clothing_Sales	Groceries_Sales
1	500	200	NULL
2	800	NULL	300
3	NULL	150	100

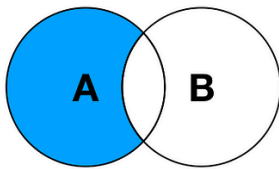
SQL JOINS



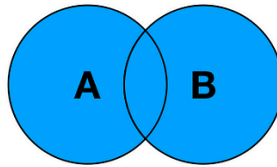
LEFT JOIN



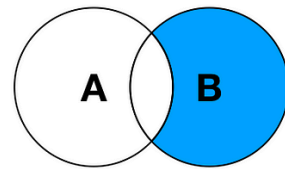
RIGHT JOIN



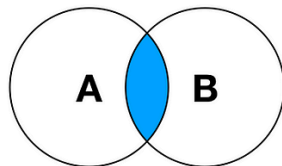
LEFT JOIN EXCLUDING
INNER JOIN



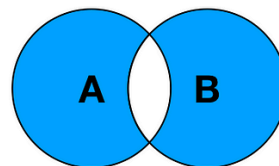
FULL OUTER JOIN



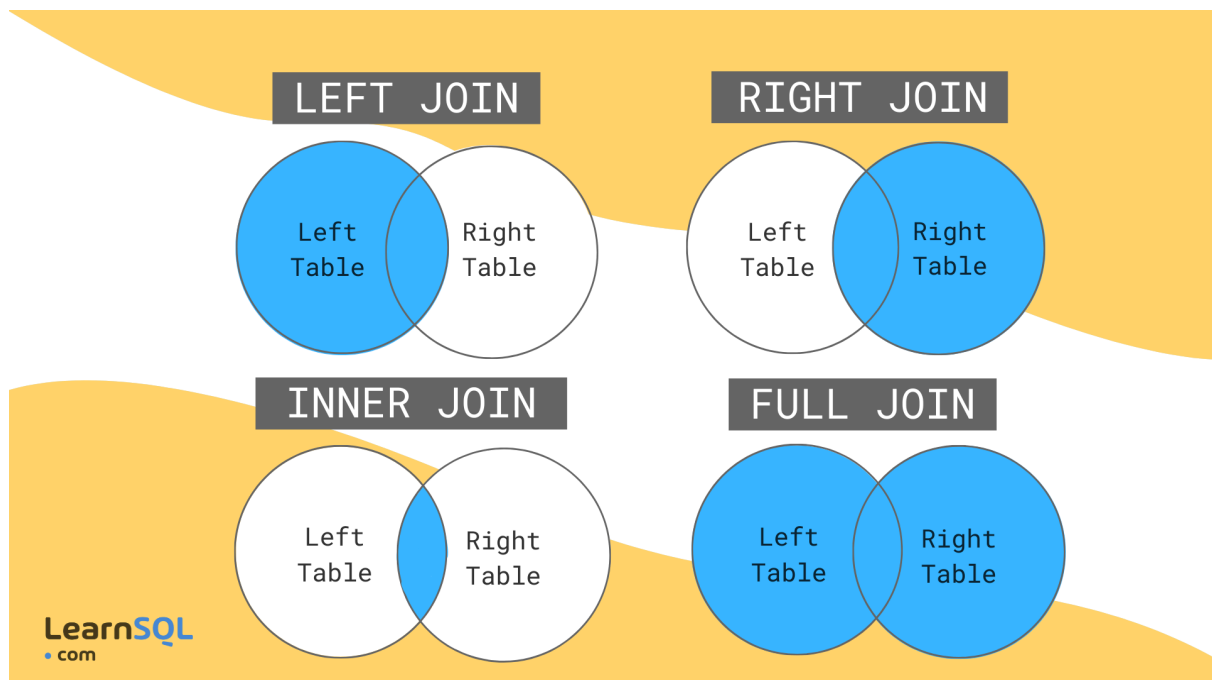
RIGHT JOIN EXCLUDING
INNER JOIN



INNER JOIN



FULL OUTER JOIN EXCLUDING
INNER JOIN



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
```