SQL BOOTCAMP -2025

Chaithra Pallavi
Numpy Ninja

Introduction

💡 What is PostgreSQL?

PostgreSQL is an **open-source, object-relational database system** known for its **reliability**, **stability**, and **advanced features**.

🌟 Key Features:

- ✅ **Open Source** — Free to use and backed by a strong community.
- 💾 **ACID Compliant** — Ensures reliable transactions.
- 🔥 **Supports Structured & Semi-Structured Data**(Tables, JSON, XML, Arrays)
- 🔗 **Extensible** — You can define your own data types, functions, and operators.
- 📈 **High Performance** — With indexing, query optimization, and concurrency control.

🔍 Where is PostgreSQL Used?

- 💼 Web Applications (Django, Rails, Spring Boot)
- 🧪 Data Warehousing & Analytics
- 🌐 Geographic Information Systems (GIS) (using PostGIS extension)

Why Learn PostgreSQL?

- Industry-standard SQL syntax.
- Strong community and documentation.
- Widely used by top companies: **Apple, Instagram, Spotify, Reddit, Cisco** and more!

✅ Fun Fact:

PostgreSQL's mascot is an elephant — representing its ability to handle large and heavy data loads!

Terminologies:

PostgreSQL Server: Software that Runs on a machine and can host multiple databases.

Database: An isolated unit inside the PostgreSQL server. Each database can have its own set of tables, indexes, functions, and users.

Schema: Inside a database, schemas help organize objects (like folders). The default is usually public.

```
PostgreSQL Server
 └── Database 1
     ├── Schema A
     │   └── Table
     └── Schema B
         └── Table
 └── Database 2
```

Connect to Database

Connect to PostgreSQL database server:

- **psql –** a terminal-based utility to connect to the PostgreSQL server.
- **pgAdmin –** a web-based tool to connect to the PostgreSQL server.

Creating and Altering a Database

A Postgre SQL server database can be created, altered and dropped:

1. Graphically using pgAdmin4
2. Using a Query

To create the database using query

```
Create Database Sample1
Create Database “Sample1”
```

To alter a database once it is created

```
ALTER DATABASE old_database_name RENAME TO new_database_name;
```

To delete/drop a database once it is created

```
DROP DATABASE sample1
DROP DATABASE “Sample1”
```

DATA TYPES

Category	Data Type	Description
Integer	`smallint`, `integer`, `bigint`	Whole numbers, 2-8 bytes, varies by size
Numeric	`real`, `double precision`, `numeric`	Decimal numbers, floating-point or exact
Character	`char(n)`, `varchar(n)`, `text`	Fixed / variable length strings
Boolean	`boolean`	`TRUE`, `FALSE`, `NULL`
Datetime	`date`, `time`, `timestamp`, `timestamptz`	Date, time, with/without timezone

Tables

```
CREATE TABLE IF NOT EXISTS table_name (

column1 datatype(length) column_constraint,

column2 datatype(length) column_constraint,

);
```

Insert values to table

```
INSERT INTO table_name (column1, column2, column3, ...)

VALUES (value1, value2, value3, ...);
```

Serial, Sequence, Identity

A sequence is a special kind of database object that generates a sequence of integers. A sequence is often used as the primary key column in a table.

When creating a new table, automatic number generation can be handled using SERIAL, SEQUENCE, or IDENTITY columns.

1️⃣SERIAL (creates a sequence automatically),

2️⃣SEQUENCE (manual creation and control),

3️⃣IDENTITY (SQL standard, handles auto-increment, control).

Primary Key

A **Primary Key** is a column (or a combination of columns) in a table that **uniquely identifies each row** in that table.

- **Uniqueness** — no duplicate records.
- **Indexing** — automatic index creation for faster lookup.
- **Data Integrity** — no NULL values allowed in primary key columns.

Foreign Key

A **Foreign Key** is a column (or a combination of columns) that creates a **relationship between two tables.**

- It refers to the **Primary Key** in another table.
- It enforces **Referential Integrity** — meaning, you cannot enter a value in the foreign key column if that value does not exist in the referenced primary key column.

Primary Key = Unique Identifier

Foreign Key = Table Relationship Enforcer

Check, Default, Unique, Not Null

1. A **CHECK** constraint ensures that values in a column or a group of columns meet a specific condition.
    a. CHECK(Age >0)
2. A **UNIQUE** constraint ensures that values stored in a column or a group of columns are unique across the whole table such as email addresses or usernames.
    a. email UNIQUE
3. To control whether a column can accept NULL, you use the **NOT NULL** constraint
    a. email UNIQUE NOT NULL
4. To define a default value for a column in the table using the **DEFAULT** constraint
    a. enrollment_status BOOLEAN DEFAULT TRUE

ER Diagram

An **Entity-Relationship Diagram (ER Diagram)** is a visual representation of the data and its relationships in a database.

It helps to **design, understand, and organize** the structure of databases by showing:

- Entities
- Attributes
- Relationships

Strong Relationships (typically when the child entity has its own primary key)

Weak Relationships (when the child entity depends on the parent entity for its identity)

Cardinality

- **One-to-One (1:1) : User & passport**
- **One-to-Many (1:N): Customer & Orders**
- **Many-to-Many (M:N): Students & Courses**

Alter Table

```
ALTER TABLE table_name action;

ALTER TABLE table_name
RENAME TO new_table_name;

ALTER TABLE table_name
ADD COLUMN column_name datatype column_constraint;

ALTER TABLE table_name
RENAME COLUMN column_name
TO new_column_name;
```

Alter/Drop Tables

```
ALTER TABLE table_name

DROP COLUMN col_name CASCADE;
```

```
ALTER TABLE books

DROP COLUMN column_name1,  DROP COLUMN column_name2;
```

```
DROP TABLE [IF EXISTS]

table_name [CASCADE | RESTRICT];
```

```
DROP TABLE table_name1, table_name2;
```

Generated Column, Temp Table, Truncate

A Generated column is a special type of column whose values are automatically calculated based on expressions or values from other columns.

A Temporary table is a table that exists only during a database session. It is created and used within a single database session and is automatically dropped at the end of the session.

```
CREATE TEMP TABLE mytemp(column1_name  data type);
```

To remove all data from a table, you use the DELETE statement without a WHERE clause. However, when the table has numerous data, the DELETE statement is not efficient. In this case, you can use the TRUNCATE TABLE statement.

```
DELETE FROM table_name;
```

```
TRUNCATE TABLE table_name;
```

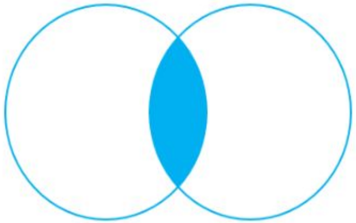
Cascading Referential Integrity

You can update a primary key **only if no other table references it**, or you use the right **CASCADE** options.

CASCADE Options (4 types):

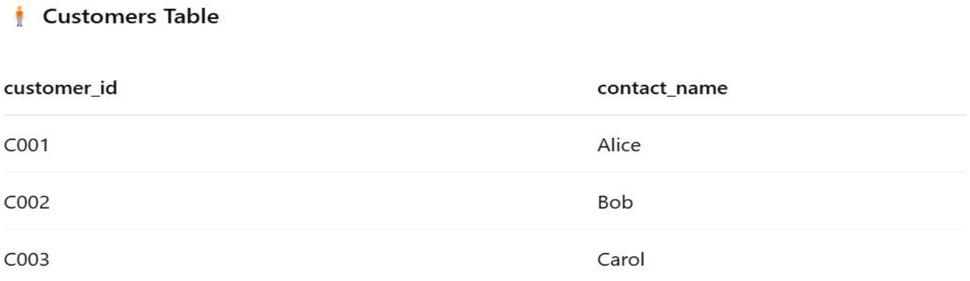
1. `ON UPDATE CASCADE`
2. `ON DELETE CASCADE`
3. `ON DELETE SET NULL`
4. `ON DELETE RESTRICT`

INNER JOIN



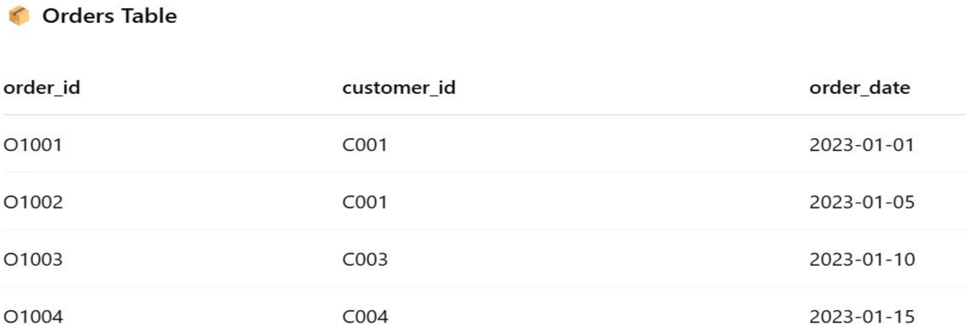
INNER JOIN

```
SELECT c.contact_name, o.order_id

FROM customers c

INNER JOIN orders o ON c.customer_id = o.customer_id;
```



🧍 Customers Table

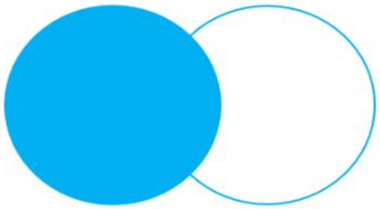
customer_id	contact_name
C001	Alice
C002	Bob
C003	Carol



📦 Orders Table

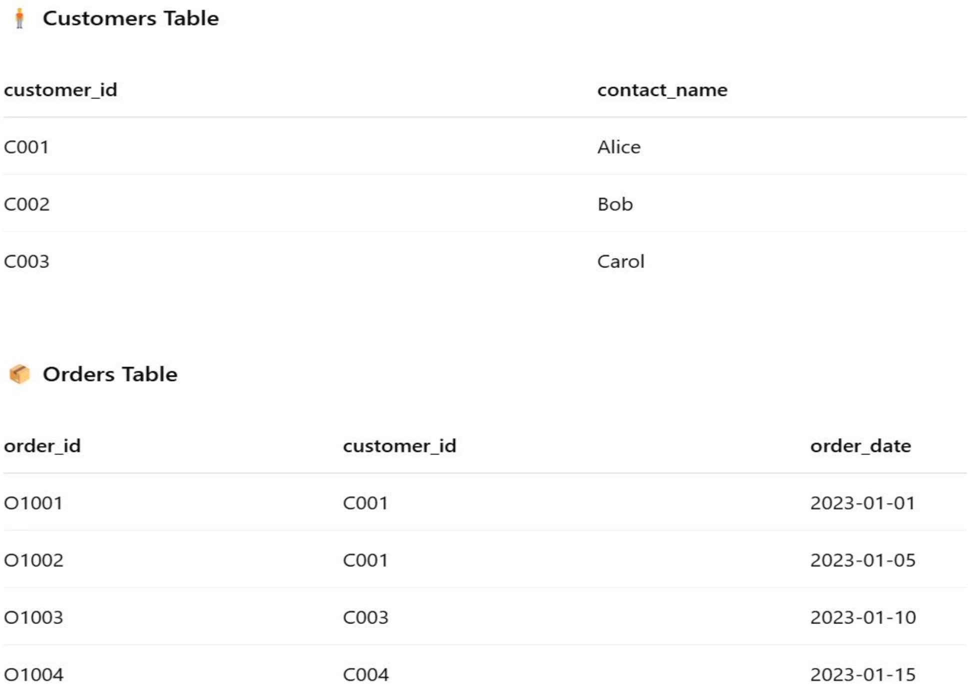
order_id	customer_id	order_date
O1001	C001	2023-01-01
O1002	C001	2023-01-05
O1003	C003	2023-01-10
O1004	C004	2023-01-15

LEFT JOIN



LEFT OUTER JOIN

```
SELECT c.contact_name, o.order_id

FROM customers c

LEFT JOIN orders o ON c.customer_id = o.customer_id;
```



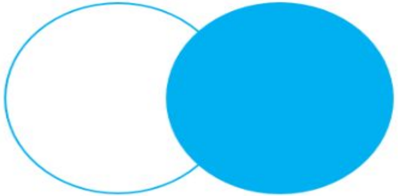
🧍 Customers Table

customer_id	contact_name
C001	Alice
C002	Bob
C003	Carol

📦 Orders Table

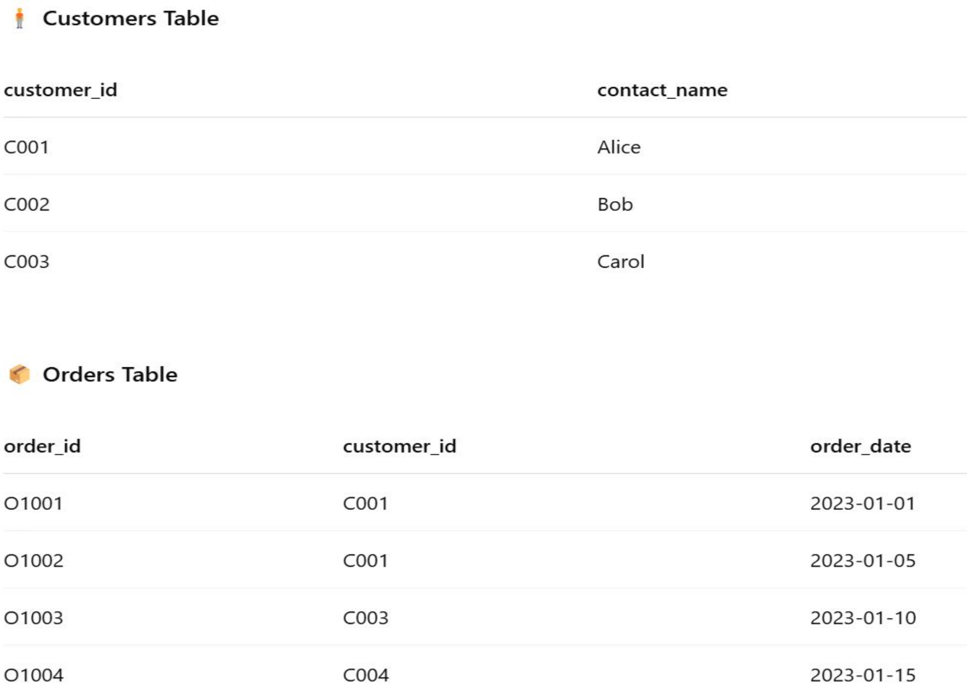
order_id	customer_id	order_date
O1001	C001	2023-01-01
O1002	C001	2023-01-05
O1003	C003	2023-01-10
O1004	C004	2023-01-15

RIGHT JOIN



RIGHT OUTER JOIN

```
SELECT c.contact_name, o.order_id

FROM customers c

RIGHT JOIN orders o ON c.customer_id = o.customer_id;
```



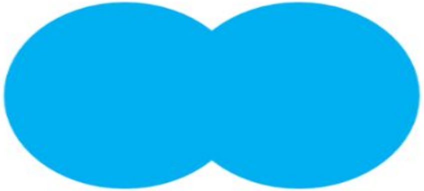
🧍 Customers Table

customer_id	contact_name
C001	Alice
C002	Bob
C003	Carol

📦 Orders Table

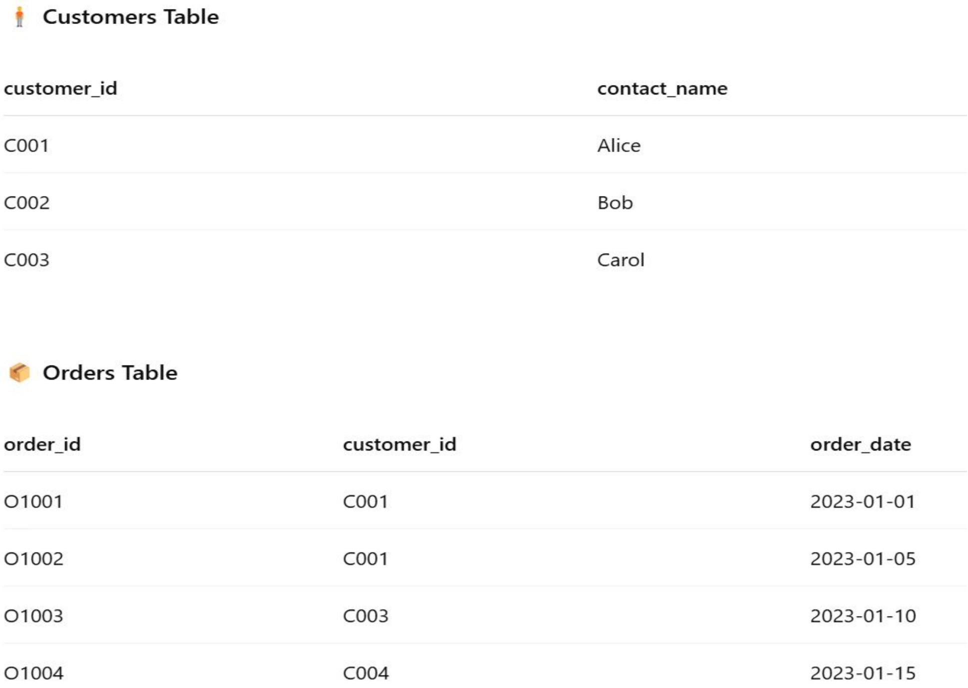
order_id	customer_id	order_date
O1001	C001	2023-01-01
O1002	C001	2023-01-05
O1003	C003	2023-01-10
O1004	C004	2023-01-15

OUTER JOIN



FULL OUTER JOIN

```
SELECT c.contact_name, o.order_id

FROM customers c

FULL OUTER JOIN orders o ON c.customer_id = o.customer_id;
```



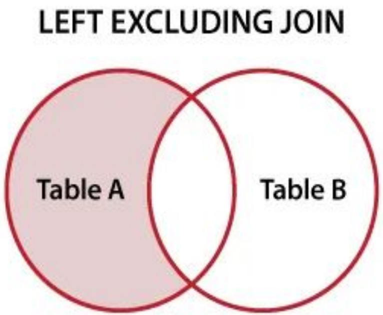
🧍 Customers Table

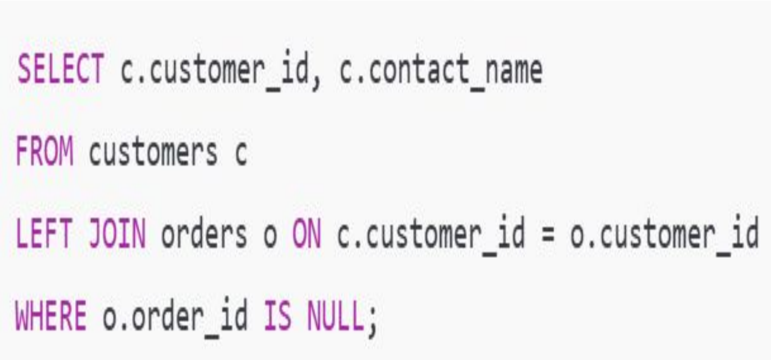
customer_id	contact_name
C001	Alice
C002	Bob
C003	Carol

📦 Orders Table

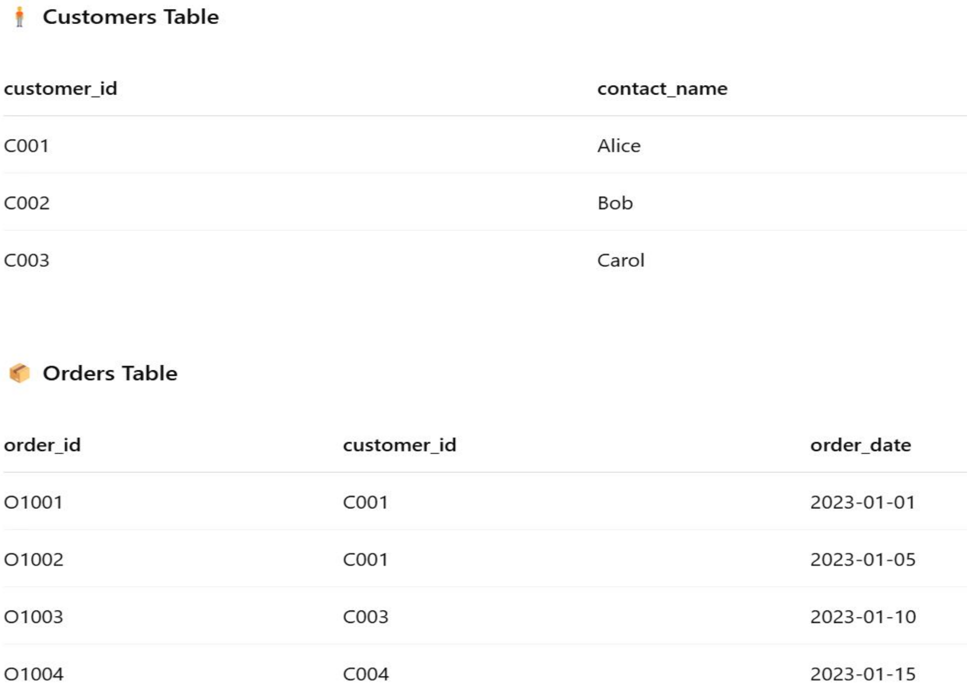
order_id	customer_id	order_date
O1001	C001	2023-01-01
O1002	C001	2023-01-05
O1003	C003	2023-01-10
O1004	C004	2023-01-15

Left Only, Left Anti Join





```sql
SELECT c.customer_id, c.contact_name
FROM customers c
LEFT JOIN orders o ON c.customer_id = o.customer_id
WHERE o.order_id IS NULL;
```



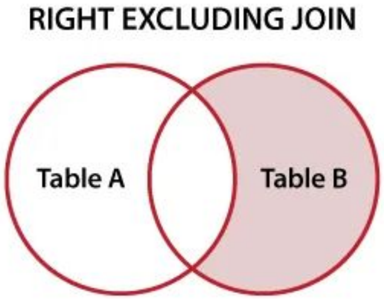
🧍 Customers Table

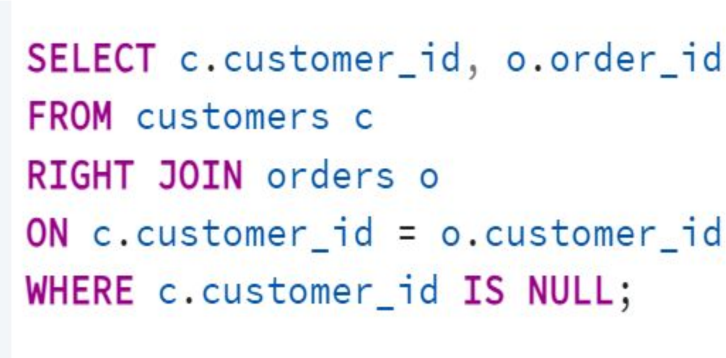
customer_id	contact_name
C001	Alice
C002	Bob
C003	Carol

📦 Orders Table

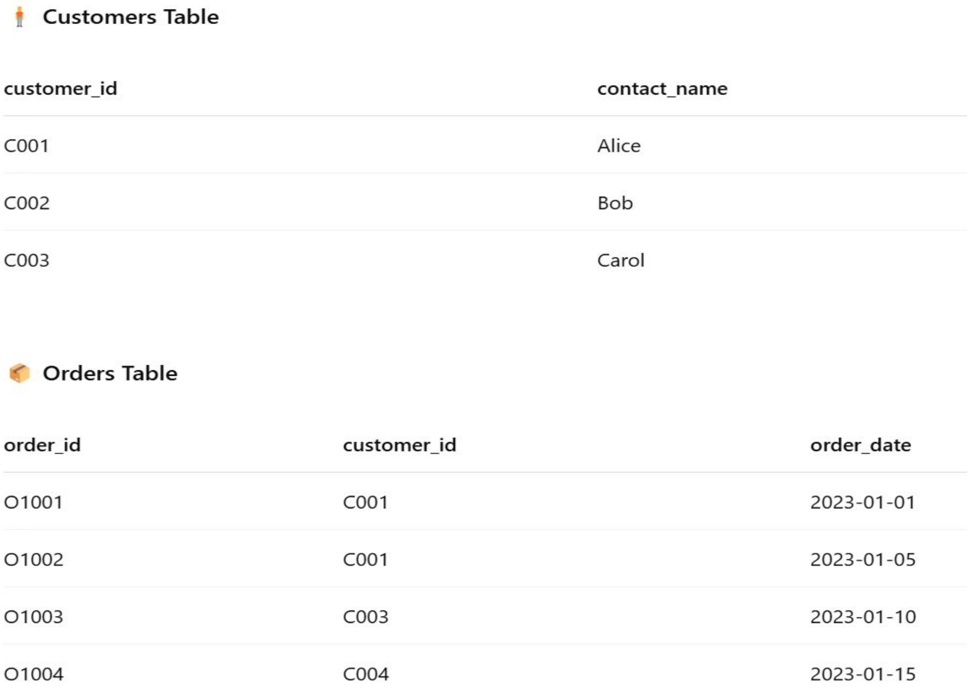
order_id	customer_id	order_date
O1001	C001	2023-01-01
O1002	C001	2023-01-05
O1003	C003	2023-01-10
O1004	C004	2023-01-15

Right Only, Right Anti join





```
SELECT c.customer_id, o.order_id
FROM customers c
RIGHT JOIN orders o
ON c.customer_id = o.customer_id
WHERE c.customer_id IS NULL;
```



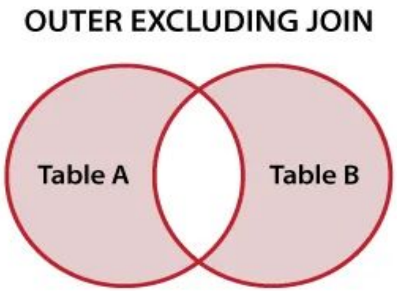
🧍 Customers Table

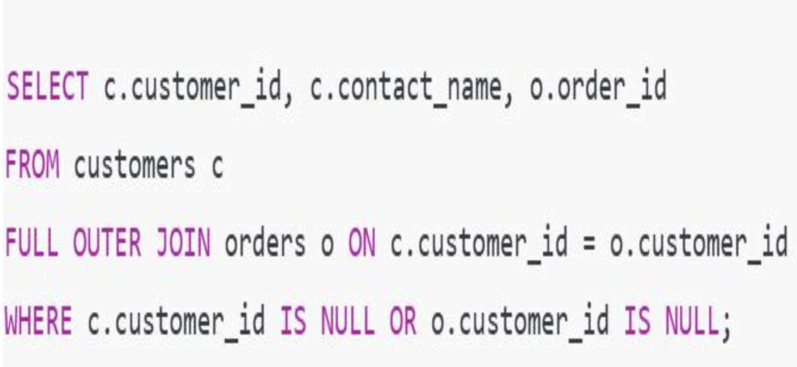
customer_id	contact_name
C001	Alice
C002	Bob
C003	Carol

📦 Orders Table

order_id	customer_id	order_date
O1001	C001	2023-01-01
O1002	C001	2023-01-05
O1003	C003	2023-01-10
O1004	C004	2023-01-15

Outer Only , Full Anti JOIN





```sql
SELECT c.customer_id, c.contact_name, o.order_id
FROM customers c
FULL OUTER JOIN orders o ON c.customer_id = o.customer_id
WHERE c.customer_id IS NULL OR o.customer_id IS NULL;
```

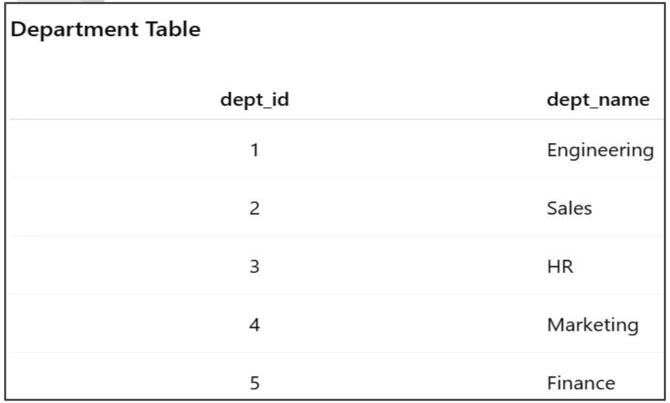
🧍 Customers Table

customer_id	contact_name
C001	Alice
C002	Bob
C003	Carol

📦 Orders Table

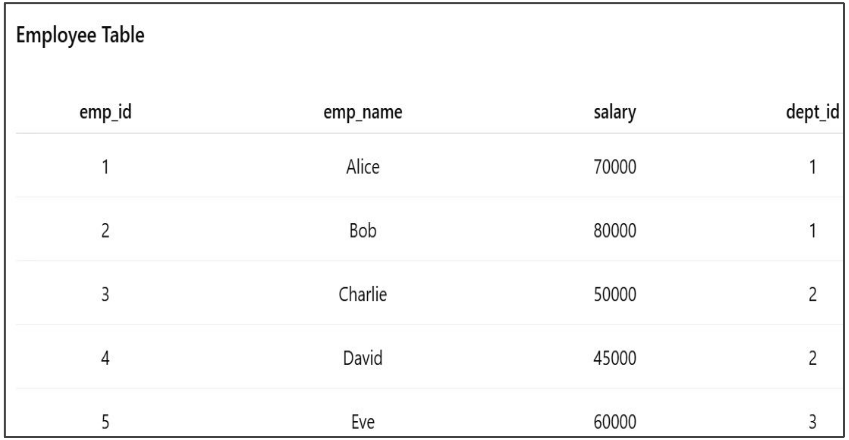
order_id	customer_id	order_date
O1001	C001	2023-01-01
O1002	C001	2023-01-05
O1003	C003	2023-01-10
O1004	C004	2023-01-15

Group BY



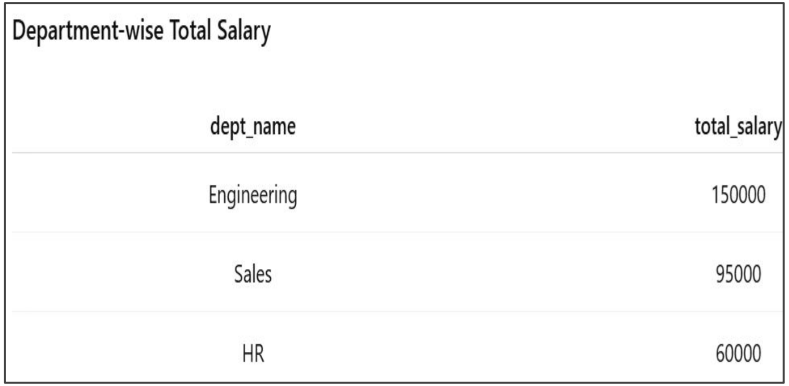
Department Table

dept_id	dept_name
1	Engineering
2	Sales
3	HR
4	Marketing
5	Finance



Employee Table

emp_id	emp_name	salary	dept_id
1	Alice	70000	1
2	Bob	80000	1
3	Charlie	50000	2
4	David	45000	2
5	Eve	60000	3



Department-wise Total Salary

dept_name	total_salary
Engineering	150000
Sales	95000
HR	60000

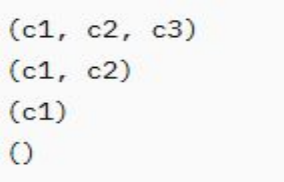
WHERE Vs HAVING

WHERE	HAVING
Can be used with Select,Insert & Update Statements	Only with Select Statement
Filter rows before aggregation	Filter rows after aggregation
Aggregate Fns cannot be used in WHERE unless its in subquery contained in HAVING clause	Aggregate Fns can be used in HAVING

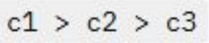
GROUPING

Grouping Sets: To generate multiple grouping sets in a query.

Roll Up: The `ROLLUP` assumes a hierarchy among the input columns and generates all grouping sets that make sense considering the hierarchy.

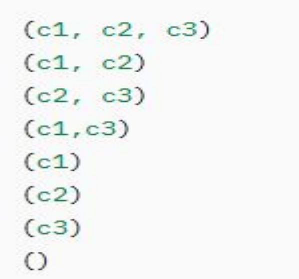


```
(c1, c2, c3)
(c1, c2)
(c1)
()
```



```
c1 > c2 > c3
```

Cube: To generate all possible grouping sets based on the specified columns



```
(c1, c2, c3)
(c1, c2)
(c2, c3)
(c1,c3)
(c1)
(c2)
(c3)
()
```

Set Operators

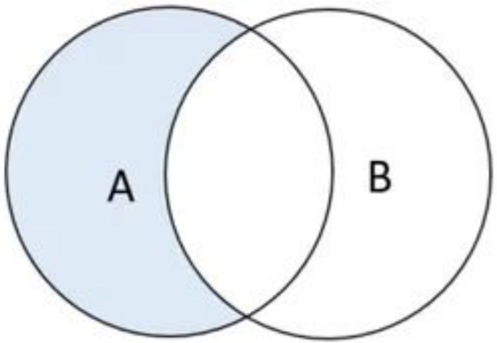
UNION: The `UNION` operator allows you to combine the result sets of two or more `SELECT` statements into a single result set. The `UNION` operator removes all duplicate rows from the combined data set.

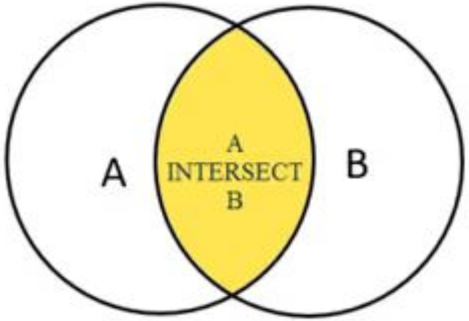
UNION ALL: To retain the duplicate rows, you use the `UNION ALL`

Set Operators

EXCEPT: The `EXCEPT` operator returns distinct rows from the first (left) query that are not in the second (right) query.

INTERSECT: Retrieves the common records from both left and right query of the intersect operator





Intersect Vs Inner Join

- Intersect filter the duplicates and return only distinct rows that are common between Left and Right query, Inner Join does not filter the duplicates.
- To make Inner Join behave as Intersect use Distinct operator
- Inner Join treats 2 NULL’s as different values

CASE, NULLIF,COALESCE, CAST

CASE: PostgreSQL CASE conditional expression is used to form conditional queries.

COALESCE(): The COALESCE() function accepts multiple arguments and returns the first argument that is not null. If all arguments are null, the COALESCE() function will return null.

```
COALESCE (NULL, 2 , 1); --Return 2
```

NULLIF(): The NULLIF function returns a null value if argument_1 equals to argument_2, otherwise, it returns argument_1.

```
NULLIF (1, 1); -- return NULL
```

CAST: to convert a value of one type into another. PostgreSQL offers the `CAST()` function and cast operator (`::`) to do this.

SubQuery & Correlated SubQuery

- A subquery is a query nested within another query. A subquery is also known as an inner query or nested query.
- A subquery can be useful for retrieving data that will be used by the main query as a condition for further data selection.
- A correlated subquery is a subquery that references the columns from the outer query.
- Unlike a regular subquery, PostgreSQL evaluates the correlated subquery once for each row processed by the outer query.

ANY, ALL, EXISTS

1. The PostgreSQL `ANY` operator compares a value with a set of values returned by a subquery

2. The PostgreSQL `ALL` operator allows you to compare a value with all values in a set returned by a subquery.

3. The `EXISTS` operator is a boolean operator that checks the existence of rows in a subquery.

Window Functions

OVER clause combined with partition by is used to break the data into partitions.

dense_rank : is a function used to assign a rank to each distinct row within a result set. It provides a non-gapped ranking of values when there are ties

The rank function assigns a unique rank to each distinct row in the result set and leaves gaps in the ranking sequence when there are ties

The row_number() function assigns a unique, sequential integer to each row within the partition of a result set. Unlike rank() and dense_rank(), it does not handle ties.

Window Functions

The lag() function in Postgres is a window function that allows you to access values from previous rows in a result set without the need for a self-join.

```
lag(value any [, offset integer [, default any ]]) over (...)
```

The lead() function in Postgres is a window function that allows you to access values from subsequent rows in a result set without the need for a self-join.

```
lead(value any [, offset integer [, default any ]]) over (...)
```

CTE

A common table expression (CTE) allows you to create a temporary result set within a query.

A CTE helps you enhance the readability of a complex query by breaking it down into smaller and more reusable parts

- Improve the readability of complex queries. You use CTEs to organize complex queries in a more organized and readable manner.
- Ability to create recursive queries, which are queries that reference themselves. The recursive queries come in handy when you want to query hierarchical data such as organization charts.
- Use in conjunction with window functions. You can use CTEs in conjunction with window functions to create an initial result set and use another select statement to further process this result set.

SYNTAX

```
WITH cte_name (column1, column2, ...) AS (
    -- CTE query
    SELECT ...
)
-- Main query using the CTE
SELECT ...
FROM cte_name;
```

Recursive

A recursive CTE allows you to perform recursion within a query using the WITH RECURSIVE syntax.

A recursive CTE is often referred to as a recursive query.

```
WITH RECURSIVE cte_name (column1, column2, ...)
AS(
    -- anchor member
            SELECT select_list FROM table1 WHERE condition

            UNION [ALL]

    -- recursive term
            SELECT select_list FROM cte_name WHERE recursive_condition
)
SELECT * FROM cte_name;
```

Transaction

A database transaction is a single unit of work that consists of one or more operations.

- Atomicity guarantees that the transaction is completed in an all-or-nothing manner.
- Consistency ensures that changes to data written to the database are valid and adhere to predefined rules.
- Isolation determines how the integrity of a transaction is visible to other transactions.
- Durability ensures that transactions that have been committed are permanently stored in the database.

- Use the **BEGIN** statement to explicitly start a transaction
- Use the **COMMIT** statement to apply the changes permanently to the database.
- Use the **ROLLBACK** statement to undo the changes made to the database during the transaction.

Views

A view is a named query stored in the PostgreSQL database server. A view is defined based on one or more tables which are known as base tables, and the query that defines the view is referred to as a defining query.

When you query a view in PostgreSQL, PostgreSQL doesn't store data inside the view — instead, it rewrites your query to run against the original base tables that the view is built on.

Advantages

- Simplifying complex queries
- Security and access control
- Logical data independence

Updatable Views

In PostgreSQL, a view is a named query stored in the database server. A view can be updatable if it meets certain conditions. This means that you can insert, update, or delete data from the underlying tables via the view.

An updatable view may contain both updatable and non-updatable columns. If you attempt to modify a non-updatable column, PostgreSQL will raise an error.

When you execute a modification statement such as INSERT, UPDATE, or DELETE to an updatable view, PostgreSQL will convert this statement into the corresponding statement of the underlying table.

If you have a `WHERE` condition in the defining query of a view, you still can update or delete the rows that are not visible through the view. However, if you want to avoid this, you can use the `WITH CHECK OPTION` to define the view.

Materialized views

Materialized views store the result of a query physically and refresh the data from base tables periodically.

```
REFRESH MATERIALIZED VIEW view_name;
```

While refreshing no other session can read from the view until the refresh is done.

```
REFRESH MATERIALIZED VIEW CONCURRENTLY view_name;
```

Recursive View

A recursive view can be useful in performing hierarchical or recursive queries on hierarchical data structures stored in the database.

Triggers

A PostgreSQL trigger is a function invoked automatically whenever an event associated with a table occurs. An event could be any of the following: INSERT, UPDATE, DELETE or TRUNCATE.

A trigger is a special user-defined function associated with a table. To create a new trigger, you define a trigger function first, and then bind this trigger function to a table.

PostgreSQL provides two main types of triggers:

- Row-level triggers
- Statement-level triggers.

Types

DML triggers:

BEFORE/AFTER INSERT, BEFORE/AFTER UPDATE, BEFORE/AFTER DELETE, BEFORE TRUNCATE,INSTEADOF,

DDL Triggers(Event Triggers):

ddl_command_start,ddl_command_end,table_rewrite,sql_drop

```
CREATE FUNCTION trigger_function()
      RETURNS TRIGGER
      LANGUAGE PLPGSQL
AS $$
BEGIN
-- trigger logic
END;
$$
```

Binding

```
CREATE TRIGGER trigger_name
   {BEFORE | AFTER} { event }
   ON table_name   [
   FOR [EACH] { ROW | STATEMENT }]
      EXECUTE PROCEDURE trigger_function
```

INSTEAD OF

INSTEAD OF triggers are a special type of triggers that intercept insert, update, and delete operations on views.

It means that when you execute an INSERT, UPDATE, or DELETE statement on a view, PostgreSQL does not directly execute the statement. Instead, it executes the statements defined in the INSTEAD OF trigger.

Enable/Disable Trigger

```
ALTER TABLE table_name

DISABLE TRIGGER trigger_name | ALL
```

```
ALTER TABLE table_name

ENABLE TRIGGER trigger_name |  ALL;
```

Disadvantages

Hidden logic

- Triggers run in the background, so it’s easy to forget they're affecting data.

Performance issues

- Too many triggers slow down INSERT, UPDATE, or DELETE operations.

Debugging difficulties

- If something goes wrong, it’s harder to trace the problem when triggers are involved.

Business logic

- If rules are complex, it's better to put them in your application code, not inside the database

Recursive behavior

- Triggers can unintentionally trigger other triggers and create loops if not handled carefully.

Stored procedures

A stored procedure is a **predefined set of SQL statements** that are **saved in the database** and can be **executed on demand**.

Think of it like a **reusable function** in programming — you define it once, and then call it whenever you need that logic.

```
create [or replace] procedure procedure_name(parameter_list)

language plpgsql

as $$declare-- variable declaration

begin--

stored procedure body

end; $$
```

Call sp_name()

```
drop procedure [if exists] procedure_name (argument_list)[

cascade | restrict]
```

Advantages

Benefit	Description
✅ Reusability	Write once, call many times
✅ Performance	Precompiled and optimized by the database
✅ Security	Control what users can do through access to procedures only
✅ Easier Maintenance	Centralized logic in the database
✅ Reduced Network Traffic	Only one CALL needed instead of many SQL queries
✅ Better Error Handling	Built-in exception blocks in most databases (like BEGIN...EXCEPTION...END)
✅ Supports Transactions	Can manage commits and rollbacks within the procedure

Functions

PostgreSQL allows developers to create **user-defined functions** to **encapsulate reusable logic**, making database operations more **efficient** and **modular.**

These functions can accept **parameters**, **perform operations**, and return values. Functions are especially useful for simplifying **complex queries** and **centralizing logi**c that can be executed multiple times without rewriting the code.

```
CREATE [OR REPLACE] FUNCTION function_name(param_list)
RETURNS return_type
LANGUAGE plpgsql
AS
$$
DECLARE
-- variable declaration
BEGIN
-- logic
END;
$$;
```

Function Overloading

We can create multiple functions with the same name, provided that each function has different arguments.

This feature, known as function overloading, allows you to define functions that perform similar operations but handle different types or numbers of inputs. PostgreSQL determines which function to execute based on the provided arguments.

Cursor

In PostgreSQL, a cursor is a database object that allows you to traverse the result set of a query one row at a time.

Cursors can be useful when you deal with large result sets or when you need to process rows sequentially.

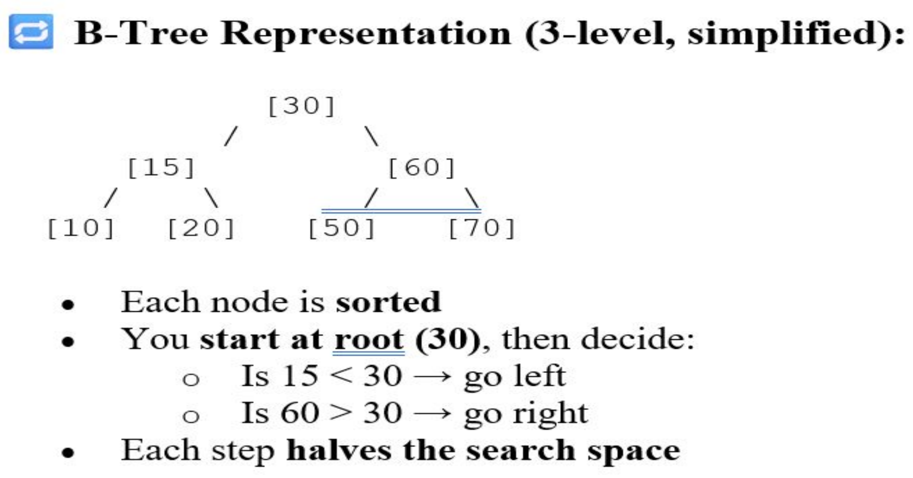
1. First, declare a cursor.
2. Next, open the cursor.
3. Then, fetch rows from the result set into a record or a variable list.
4. After that, process the fetched row and exit the loop if there is no more row to fetch.
5. Finally, close the cursor.

FETCH NEXT: fetches the next row from the cursor.
FETCH PRIOR: fetches the previous row from the cursor.
FETCH FIRST: fetches the first row from the cursor.
FETCH LAST: fetches the last row from the cursor.
FETCH ALL: fetches all rows from the cursor.

Indexes

Indexes are used by the queries to find the data from the tables quickly.

```
CREATE INDEX [IF NOT EXISTS] index_nameON table_name(column1, column2, ...);
```



🔁 B-Tree Representation (3-level, simplified):

```
            [30]
          /      \
     [15]          [60]
    /    \        /    \
[10]    [20]   [50]     [70]
```

- Each node is **sorted**
- You **start at root (30)**, then decide:
    - Is 15 < 30 → go left
    - Is 60 > 30 → go right
- Each step **halves the search space**

Indexes- Performance Considerations

- Index Maintenance Overhead
- Index Selectivity
- Multiple Indexes