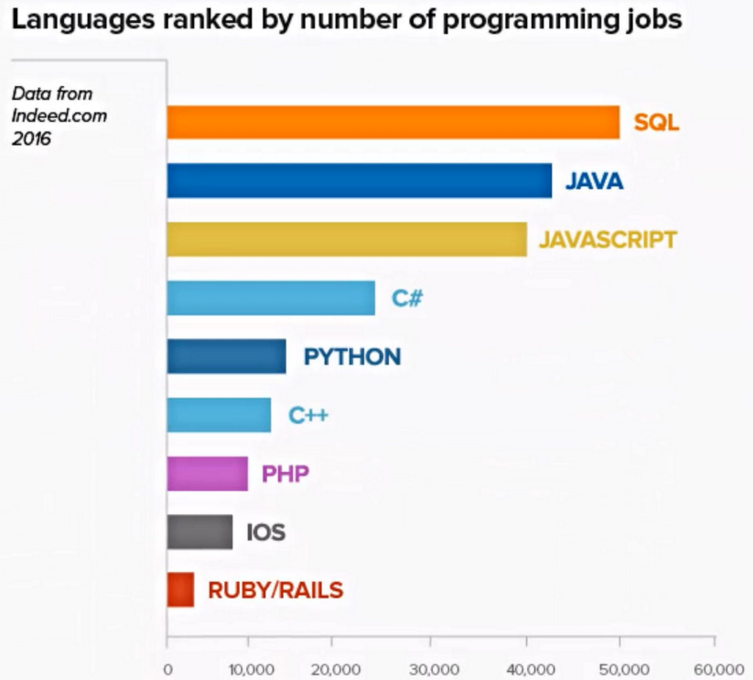
0. SQL

01 May 2024 22:01



- SQL is the programming language used to communicate with our Database

SQL Example :

```
SELECT customer_id, first_name, last_name
FROM sales
ORDER BY first_name;
```

SQL is useful for a lot of things!

MySQL
PostGreSQL
Oracle Databases
Microsoft Access
Amazon's Redshift
Looker
MemSQL
Periscope Data
Hive (Runs on top of Hadoop)
Google's BigQuery
Facebook's Presto

- PostgreSQL - SQL Engine that stores data and reads queries and returns information.
- PgAdmin - Graphical User Interface for connecting with PostgreSQL

1. Intro

21 April 2024 14:39

PostgreSQL is a popular *relational database management system* ***(RDBMS).***

ORDBMS [*Open-Source Object-Relational Database Management System*].

It supports both SQL (relational) and JSON (non-relational) querying and It is a stable database supported by more than 20 years of development by the open-source community.
PostgreSQL is not controlled by any corporation or other private entity and the source code is available free of charge.

- PostgreSQL is initially introduced *on 8th July 1996* at the *University of California*, Berkley (UCB)
- It is the first DBMS, which perform *MVCC* [*Multi-Version Concurrency Control*] feature, even before Oracle. (also known as *snapshot* isolation in Oracle).
- Written in C language
- PostgreSQL is *cross-platform* and runs on various operating systems such as *Microsoft Windows, UNIX, FreeBSD, Mac OS X, Solaris, HP-UX, LINUX*, and so on.
- PostgreSQL follow the ***transaction*** along with the ***ACID (Atomicity, Consistency, Isolation, and Durability)*** properties.

From <https://www.javatpoint.com/postgresql-tutorial>

-

RDBMS stands for Relational Database Management System. It's a type of software that lets you create, manage, and access data stored in a relational database.

- **Relational Database:** This is a collection of data organized into tables. Each table has rows and columns,
  - where rows represent individual records and columns represent specific attributes of that data.
  - For instance, a customer table might have rows for each customer and columns for their name, address, and email.
- **Relationships:** The relational part comes in how these tables are linked.
  - Tables can have connections defined based on shared data points.
  - This allows you to efficiently retrieve and analyze data from multiple tables at once.
  - Imagine a table for orders and another for customers. By relating them through a customer ID, you can easily see a customer's order history.
- **Management System:** This is the RDBMS software itself.
  - It provides tools to create and manage the database structure.
  - define relationships between tables.
  - store and update data
  - query the database to retrieve specific information.

Popular examples of RDBMS include MySQL, PostgreSQL, Microsoft SQL Server, and Oracle Database.

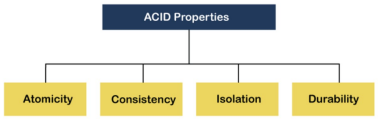
From <https://gemini.google.com/app/c_e337a9afc2e25276>

Transaction - a transaction is a single logical unit of work that performs operations on the database. It's like a mini-program that accesses and potentially modifies the data.

A transaction typically consists of one or more database operations, such as reading, writing, updating, or deleting data. These operations are grouped together to ensure that they are treated as a single unit that either succeeds completely or fails completely.
Transactions ensure data integrity by following the ACID properties

- **Operations:** A transaction consists of a series of database operations, typically involving reads and writes. For instance, transferring funds between accounts might involve reading the current balances, subtracting from one account, and adding to another.
- **Logical Unit:** All the operations within a transaction are treated as a whole. Either all of them succeed, or none of them do. This prevents partially completed transactions from leaving the database in an inconsistent state

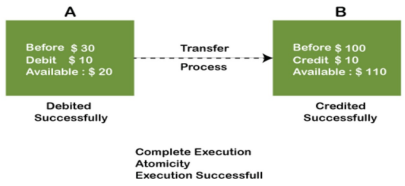
From <https://gemini.google.com/app/e337a9afc2e25276>



These are fundamental properties essential for ensuring the reliability and integrity of transactions in database systems.

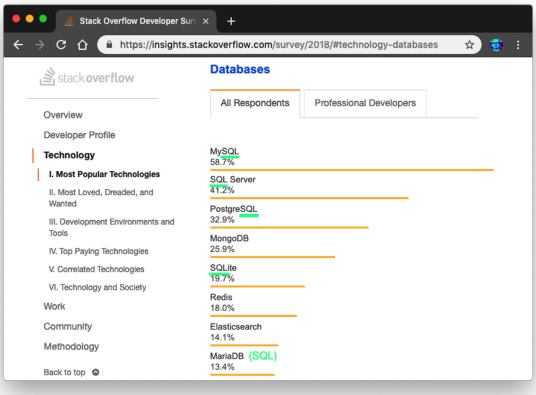
1. **Atomicity** -
   It means if any operation is performed on the data, either it should be performed or executed completely or should not be executed at all. (operation should not break in between or execute partially.)

   Example :

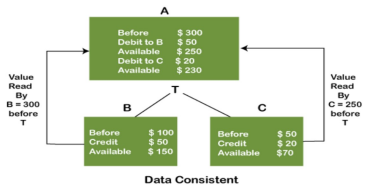


   Start of Transaction (starts with the intention to transfer $10 from account (A) to account (B).)
   |
   V
   Debit $10 from A (Successful) debit operation, taking $10 from account (A), executes successfully
   |
   V
   Credit $10 to B (Failed) the credit operation, adding $10 to account (B), fails due to some system error
   |
   V
   End of Transaction (Rollback) the transaction cannot be completed successfully, so it rolls back

   This demonstrates atomicity where the transaction is treated as a single unit, ensuring that either all operations within the transaction are completed successfully or none of them are applied, maintaining the consistency of the database.



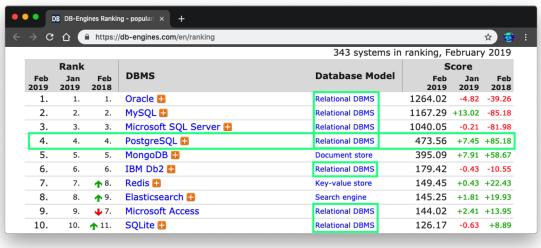
2. **Consistency** -
   ensures that a transaction transforms a database from one consistent state to another consistent state. It preserves the integrity constraints of the database. (transaction maintains the validity of the database according to pre-defined rules.)



   Example, a database might have a constraint that an account balance cannot be negative. Consistency ensures that any transaction updating account balances never results in a negative balance. It validates the data against these rules before committing the change.

   Transaction T from A to B and C: transaction starts with the intention to transfer $50 from account A to account B and $20 from account A to account C. Initially A = 300$, B= 100$, C=50$

   Start of Transaction
   |
   v
   Read $300 from A by B Before the transaction, account B reads the balance from account A as $300
   |
   v
   Debit $50 from A (Successful) debit operation takes $50 from account A.
   |
   v
   Credit $50 to B (Successful) credit operation adds $50 to account B
   |
   v
   Read $250 from A by C account C reads the balance from account A as $25
   |
   v
   Debit $20 from A (Successful) debit operation takes $20 from account A.
   |
   v
   Credit $20 to C (Successful) and the credit operation adds $20 to account C
   |
   v
   End of Transaction

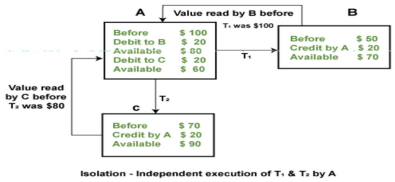


Rank Feb 2019	Rank Jan 2019	Rank Feb 2018	DBMS	Database Model	Score Feb 2019	Score Jan 2019	Score Feb 2018
1.	1.	1.	Oracle	Relational DBMS	1264.02	-4.82	-39.26
2.	2.	2.	MySQL	Relational DBMS	1167.29	+13.02	-85.18
3.	3.	3.	Microsoft SQL Server	Relational DBMS	1040.05	-0.21	-81.98
4.	4.	4.	PostgreSQL	Relational DBMS	473.56	+7.45	+85.18
5.	5.	5.	MongoDB	Document store	395.09	+7.91	+58.67
6.	6.	6.	IBM Db2	Relational DBMS	179.42	-0.43	-10.55
7.	7.	8.	Redis	Key-value store	149.45	+0.43	+22.43
8.	8.	9.	Elasticsearch	Search engine	145.25	+1.81	+19.93
9.	9.	7.	Microsoft Access	Relational DBMS	144.02	+2.41	+13.95
10.	10.	11.	SQLite	Relational DBMS	126.17	-0.63	+8.89

The balances in accounts A, B, and C are consistent with the transactions performed.
This demonstrates consistency, where the database remains in a consistent state throughout the transaction. All operations within the transaction are executed successfully, and the data remains consistent across all involved accounts.

3. **Isolation**

This property deals with concurrent transactions, meaning multiple transactions happening at the same time. Isolation ensures that each transaction appears to run independently, even if they're accessing the same data



Isolation - Independent execution of T_1 & T_2 by A

Example :
account A is making T1 and T2 transactions to account B and C, but both are executing independently without affecting each other. It is known as Isolation.

Transaction T1 from A to B: T1 starts with the intention to transfer $50 from account A to account B

```
Start of Transaction T1
        |
        v
Debit $50 from A       (Successful)
        |
        v
Credit $50 to B        (Successful)
        |
        v
End of Transaction T1
```

Transaction T2 from A to C: Independently, transaction T2 starts with the intention to transfer $30 from account A to account C.

```
Start of Transaction T2
        |
        v
Debit $30 from A       (Successful)
        |
        v
Credit $30 to C        (Successful)
        |
        v
End of Transaction T2
```

T1 executes its debit and credit operations without being affected by the execution of transaction T2.
Similarly, transaction T2 executes its debit and credit operations without being affected by the execution of transaction T1.
- Both transactions execute independently, and their operations do not interfere with each other's execution.
- Each transaction maintains its own isolated view of the database, ensuring that the operations of one transaction do not affect the outcome of the other transaction.

This demonstrates isolation, where transactions execute independently without interfering with each other, preserving the integrity and consistency of the database.

4. Durability -
durability ensures that the changes made by the transaction are permanently stored in the database and will not be lost even if the database system crashes immediately after the commit.

```
Start of Transaction
  |
  v
Database Operations
  |
  v
Commit Transaction   (Successful)
  |
  v
End of Transaction
  |
  v
System Failure / Crash
  |
  v
Recovery Process
  |
  v
Database Operations Replay
```

- The transaction begins with the execution of database operations, such as reading, writing, updating, or deleting data.
- Once the transaction completes its operations, it is committed, indicating that all changes made by the transaction should be permanently stored in the database.
- After the transaction is committed, but before the end of the transaction, a system failure or crash occurs. This could be due to power loss, hardware failure, or software errors.
- Upon system recovery, a recovery process is initiated to restore the database to a consistent state.
- During the recovery process, the durability property ensures that the changes made by the committed transaction are replayed or restored from durable storage to bring the database back to its state before the crash.
- Once the recovery process is complete, the database operations of the committed transaction are replayed or restored, ensuring that the effects of the transaction persist despite the system failure.

This demonstrates durability, where committed changes made by a transaction are durable and survive system failures, ensuring the reliability and persistence of data in the database.

From <https://chat.openai.com/c/bea2b950-c2ef-4f0f-bce5-562311cf3741>

2. Fundamentals

02 May 2024 00:13

1. **SELECT**
   a. SQL SELECT statement is the workhorse for retrieving data from relational databases(one or more tables) .
   b. It allows you to specify which columns you want to retrieve, as well as any conditions for filtering the data

```
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

- SELECT: This keyword specifies the columns that you want to retrieve from the database.
- column1, column2, ...: These are the names of the columns you want to retrieve.we can select all columns using * or list specific column names separated by commas
- FROM: This keyword specifies the table or tables from which you want to retrieve the data.
- table_name: This is the name of the table from which you want to retrieve the data.
- WHERE (optional): used to specify conditions that the retrieved data must meet. Only rows that satisfy the specified conditions will be returned. allows you to filter the results based on a specific condition. You can use comparison operators like =, >, <, and logical operators like AND, OR to refine your selection.
- Condition (optional): This is the condition that must be met for a row to be included in the result set. It can include comparisons, logical operators, and other SQL expressions.

  - In general it is not good practice to use an asterisk (*) in the SELECT statement if you don't really need all columns.
  - It will automatically query everything, which increases traffic between the database server and the application, which can slow down the retrieval of results.

Ex : Suppose we have a table named employees with the following columns: employee_id, first_name, last_name, department, and salary.

i. We want to retrieve the first_name, last_name, and department columns from the employees table.
ii. We only want to retrieve data for employees who belong to the IT department, so we use the WHERE clause to specify the condition department = 'IT'.

```
SELECT first_name, last_name, department
FROM employees
WHERE department = 'IT';
```

2. **DISTINCT**

It is used to eliminate duplicate rows from the result set of a SELECT query. It ensures that each row returned by the query is unique (It works in conjunction with the SELECT statement)

```
SELECT DISTINCT column1, column2, ...
FROM table_name
WHERE condition;
```

- SELECT DISTINCT: This specifies that you want to retrieve unique values for the specified columns.

1. **Placement:** You add the `DISTINCT` keyword right after the `SELECT` clause in your query.
2. **Functionality:** When `DISTINCT` is used, the database engine identifies and removes duplicate rows based on the selected columns. It essentially returns only one instance of each unique combination of values in those columns.

Ex . 1
- Suppose we have a table named employees with the following columns: employee_id, department, and salary
- We want to retrieve unique department names from the employees table.
- We use the DISTINCT keyword to ensure that each department name is only listed once, even if there are multiple employees in the same department.
- Since we're not using a WHERE clause, all departments from the employees table will be considered.

```
SELECT DISTINCT department
FROM employees;
```

Ex.2
- `DISTINCT` can be used with multiple columns. It removes duplicates based on the combined uniqueness of all the specified columns.
- You can use `DISTINCT` with aggregate functions like `COUNT` to get the number of distinct values in a column. For example:

```
SELECT COUNT(DISTINCT City) FROM Customers;
```

- This query will count the number of unique cities present in the City column of the Customers table.
- By incorporating DISTINCT into your SQL queries, you can ensure your results are free of redundancies and provide a clearer picture of the unique data points within your tables.

Note : DISTINCT works with or without parentheisis.

SELECT DISTINCT column **FROM** table

SELECT DISTINCT(column) **FROM** table

3. **Count**

The COUNT function in SQL is used to count the number of rows that meet a specified condition within a query. It can be used with the SELECT statement to return the number of rows that satisfy the given criteria

```
SELECT COUNT(column_name)
FROM table_name
WHERE condition;
```

- COUNT(column_name): This specifies the column for which you want to count the number of non-null values. You can also use COUNT(*) to count all rows, regardless of whether they contain null values.
- The asterisk (*) acts as a wildcard, signifying that you want to count all rows in the table

Example : count the number of employees in IT Department

```
SELECT COUNT(*)
FROM employees
WHERE department = 'IT';
```

Example 2: if each column has same number of rows

Name	Choice
Zach	Green
David	Green
Claire	Yellow
David	Red

- SELECT COUNT(name) FROM table;
- SELECT COUNT(choice) FROM table;
- SELECT COUNT(*) FROM table;
- All return the same thing, since the original table had 4 rows.

Count
4

Count is more poweful when combined with other commands such as distinct.
Example : how many unique names are there in above table.

```
SELECT COUNT ( DISTINCT ( Name))
FROM table_name;
```

Name	Choice
Zach	Green
David	Green
Claire	Yellow
David	Red

Name	Choice
Zach	Green
David	Green
Claire	Yellow
David	Red

Name	Choice
Zach	Green
David	Green
Claire	Yellow

Count
3

- COUNT can also be used with the GROUP BY clause to perform aggregate counts for specific groups within your data.
- COUNT will disregard NULL values by default. You can use COUNT(*) to include NULL values in the count.

4. **SELECT WHERE Statement**

The WHERE clause is used to filter rows returned by a SELECT, UPDATE, DELETE, or INSERT INTO statement based on a specified condition. It allows you to retrieve only the rows that meet certain criteria.

5. ORDER BY

The ORDER BY statement in SQL is used to sort the result set of a query based on one or more columns. It allows you to specify the order in which the rows should appear in the output

```
SELECT column1, column2, ...
FROM table_name
ORDER BY column1 [ASC|DESC], column2 [ASC|DESC], ...;
```

- ORDER BY: This keyword is followed by the column(s) based on which you want to sort the result set.
- ASC (optional): This specifies ascending order, which is the default if not specified explicitly. Rows are sorted in ascending order by default.
- DESC (optional): This specifies descending order. Rows are sorted in descending order if this keyword is specified.

- **Placement:** The ORDER BY clause is added after the WHERE clause (if used) in your SELECT statement.
- **Sorting Criteria:** You specify the column(s) you want to sort by within the ORDER BY clause. Separate multiple columns with commas.
- **Ascending vs. Descending Order:**
  - By default, ORDER BY sorts the data in ascending order (from lowest to highest value for numbers, A to Z for text).
  - To sort in descending order (highest to lowest or Z to A), you can use the DESC keyword after the column name.

Example 1 : retieve first_name, last_name, and salary of all employees sorted by salary in descending order, meaning the employees with the highest salary will appear first in the result set.

```
SELECT first_name, last_name, salary
FROM employees
ORDER BY salary DESC;
```

Example 2 : ORDER BY with multiple columns

Employee table

employee_id	department	salary	hire_date
1	IT	60000	2022-01-10
2	HR	55000	2021-12-05
3	IT	65000	2022-02-15
4	Finance	70000	2022-03-20
5	HR	50000	2022-01-20

```
SELECT employee_id, department, salary, hire_date
FROM employees
ORDER BY department ASC, salary DESC, hire_date ASC;
```

1. Sorting by Department (ASC)

employee_id	department	salary	hire_date
1	IT	60000	2022-01-10
3	IT	65000	2022-02-15
4	Finance	70000	2022-03-20
2	HR	55000	2021-12-05
5	HR	50000	2022-01-20

2. Sorting by Salary (DESC) within each Department

employee_id	department	salary	hire_date
3	IT	65000	2022-02-15
1	IT	60000	2022-01-10
4	Finance	70000	2022-03-20
2	HR	55000	2021-12-05
5	HR	50000	2022-01-20

3. Sorting by Hire Date (ASC) within each Department and Salary group

employee_id	department	salary	hire_date
1	IT	60000	2022-01-10
3	IT	65000	2022-02-15
5	HR	50000	2022-01-20
2	HR	55000	2021-12-05
4	Finance	70000	2022-03-20

- The result set will be sorted first by the department column in ascending order (ASC). This means that all rows with the same department will be grouped together
  1. **"IT" vs. "Finance":**
     - The first characters of both strings are "I" and "F" respectively.
     - In ASCII/Unicode, the character "I" has a lower value than "F" ('I' < 'F'). Therefore, "IT" would appear before "Finance" when sorted in ascending order.
  2. **"Finance" vs. "HR":**
     - The first characters of both strings are "F" and "H" respectively.
     - In ASCII/Unicode, the character "F" has a lower value than "H" ('F' < 'H'). Therefore, "Finance" would appear before "HR" when sorted in ascending order.
- Within each department group, the rows will now be sorted based on the salary column in descending order (DESC).
- Within each department and salary group, the rows will now be sorted based on the hire_date column in ascending order (ASC).
- This final result set is sorted first by department in ascending order, then by salary within each department group in descending order, and finally by hire date within each department and salary group in ascending order.

- Notice ORDER BY towards the end of a query, since we want to do any selection and filtering first, before finally sorting.
  - **SELECT** column_1,column_2
    FROM table
    ORDER BY column_1 **ASC / DESC**

6. **LIMIT**

In SQL, the LIMIT clause is used to constrain the number of rows returned by a query. It is commonly used with the SELECT statement to limit the number of rows in the result set. The LIMIT clause is especially useful when dealing with large datasets or when you only need to retrieve a subset of the results.

Limit goes at very end of a query request, and is the last command to be executed.

```
SELECT column1, column2, ...
FROM table_name
LIMIT number_of_rows;
```

1. **Placement:** The `LIMIT` clause is added at the end of your `SELECT` statement, following any `WHERE` or `ORDER BY` clauses you might have used.
2. **Specifying the Limit:** You provide a number within parentheses after `LIMIT` to indicate the maximum number of rows you want to retrieve.

- **Zero Rows:** Specifying `LIMIT 0` will return zero rows, effectively acting as a filter that retrieves none of the data.
- **LIMIT with OFFSET:** `LIMIT` can be used in conjunction with `OFFSET` to retrieve data from a specific position within the results.

Example 1: you have a table named Books with hundreds of entries. You might only be interested in browsing the most recent 10 additions

```
SELECT Title, Author
FROM Books
ORDER BY PublishDate DESC
LIMIT 10;
```

7. **LIMIT with OFFSET**

The LIMIT clause with an offset is used to retrieve a subset of rows from a result set, starting from a specified position. This is particularly useful when you want to paginate through a large result set or skip a certain number of rows before retrieving the next set of rows. The OFFSET keyword specifies the number of rows to skip before starting to return rows

```
SELECT column1, column2, ...
FROM table_name
LIMIT number_of_rows OFFSET offset_value;
```

offset_value: This is the number of rows to skip before starting to return rows

Example: This query sorts by PublishDate (newest first), skips the first 5 entries using OFFSET 5, and then returns the following 3 books using LIMIT 3.

```
SELECT *
FROM Books
ORDER BY PublishDate DESC
LIMIT 3 OFFSET 5;
```

4. SELECT WHERE Statement

The WHERE clause is used to filter rows returned by a SELECT, UPDATE, DELETE, or INSERT INTO statement based on a specified condition. It allows you to retrieve only the rows that meet certain criteria.

```
SELECT column1, column2, ...
FROM table_name
WHERE condition;
```

- WHERE: This keyword is followed by a condition that must be met for a row to be included in the result set.
- condition: This is the condition that each row must satisfy to be included in the result set. It can include comparisons, logical operators, and other SQL expressions.

Example 1: We want to retrieve the employee_id, department, and salary whose salary is gretaer than 50000.

```
SELECT employee_id, department, salary
FROM employees
WHERE salary > 50000;
```

The WHERE clause can include various conditions such as equalities, inequalities, comparisons (<, >, =), logical operators (AND, OR, NOT), and functions.

- Comparison Operators

Operator	Description
=	Equal
>	Greater than
<	Less Than
>=	Greater than or equal to
<=	Less than or equal to
<> or !=	Not equal to

- Logical Operators
  - Allow us to combine multiple comparison operators
    - AND
    - OR
    - NOT

Example 2: We want to retrieve the employee_id, department, and salary whose salary is gretaer than 50000 and are from IT Department.

```
SELECT employee_id, department, salary
FROM employees
WHERE salary > 50000 AND department = 'IT';
```

Example 3: homay many total number of movies are there whose rental rate is greater than 4 and replacement cose is greater than or equal to 19.99 and rating is "R"

Query Editor | Query History

```
SELECT COUNT(*) FROM film
WHERE rental_rate > 4 AND replacement_cost >= 19.99
AND rating='R';
```

Example 4 : Count of movies with rating R or PG-13

Query Editor | Query History

```
SELECT COUNT(*) FROM film
WHERE rating = 'R' OR rating = 'PG-13';
```

1. AS Clause :

This is used to assign an alias to a table or column, which can make your queries more readable and the results easier to interpret. Aliases are temporary and only exist for the duration of the query. They do not affect the actual table or column names in the database

The AS keyword in SQL serves two primary purposes:

Aliasing Columns
That is assigning a new name to the column in the result set.

```
SELECT column_name AS alias_name
FROM table_name;
```

Example:
Suppose we have a table named employees with columns first_name, last_name, and salary. We want to concatenate the first and last names and display the result as full_name.

```
SELECT first_name || ' ' || last_name AS full_name, salary
FROM employees;
```

- We're concatenating first_name and last_name using the || operator (for databases like PostgreSQL, SQLite; in MySQL, use CONCAT(first_name, ' ', last_name)).
- We're using AS to give the concatenated result an alias full_name.
- The result set will display full_name and salary.

Aliasing Tables
That is assigning a new name to the table within the query, which can simplify references to the table, especially when dealing with complex queries or joins.

```
SELECT column_name
FROM table_name AS alias_name;
```

Example:
Suppose we have two tables: employees and departments. We want to join these tables and display the department name and the employee's full name.

```
SELECT e.first_name || ' ' || e.last_name AS full_name, d.department_name
FROM employees AS e
JOIN departments AS d ON e.department_id = d.department_id;
```

- We're giving the employees table an alias 'e' and the departments table an alias 'd'.
- We're using these aliases to reference the columns in the SELECT clause and the JOIN condition.
- The result set will display full_name and department_name.

- Aliases can be any valid identifier but should be descriptive to enhance code readability.
- You can use aliases in WHERE and HAVING clauses as well, referencing the aliased names instead of the original column or table names.

As Operator can also be used with functions.

```
SELECT SUM(amount) AS net_revenue
FROM payment;
```

Data Output | Explain | Messages | Notifications

net_revenue numeric
61312.04

Note:

The AS operator gets executed at the very end of a query, meaning that we can not use the ALIAS inside a WHERE operator.

```
SELECT customer_id,SUM(amount) AS total_spent
FROM payment
GROUP BY customer_id
HAVING total_spent > 100
```

Data Output | Explain | Messages | Notifications

```
ERROR:  column "total_spent" does not exist
LINE 4: HAVING total_spent > 100
SQL state: 42703
Character: 88
```

```
SELECT customer_id,SUM(amount) AS total_spent
FROM payment
GROUP BY customer_id
HAVING SUM(amount) > 100
```

Data Output | Explain | Messages | Notifications

customer_id [PK] integer	total_spent numeric
87	137.72
477	136.79
273	138.72
550	153.69
51	133.70

```
SELECT customer_id,amount AS new_name
```

```
SELECT customer_id,amount AS new_name
```

Example: This query sorts by PublishDate (newest first), skips the first 5 entries using OFFSET 5, and then returns the following 3 books using LIMIT 3.

```
SELECT *
FROM Books
ORDER BY PublishDate DESC
LIMIT 3 OFFSET 5;
```

Skip the first 5 books and then return the next 3.

8. BETWEEN

In SQL, the BETWEEN operator is used to filter rows based on a range of values within a specified range. It allows you to select rows where a column value falls within a specific range of values, inclusive of both endpoints

It's useful for scenarios where you want to select data that falls between a minimum and a maximum value.

```
SELECT column1, column2, ...
FROM table_name
WHERE column_name BETWEEN value1 AND value2;
```

- BETWEEN: This is the operator used to specify the range of values.
- value1: The lower bound (inclusive) of the range.
- value2: The upper bound (inclusive) of the range.

Inclusive range - This means the query will return rows where the value in the specified column is greater than or equal to value1 and less than or equal to value2.

- The BETWEEN operator is the same as:
  - value >= low AND value <= high
  - value BETWEEN low AND high

Example : get list of students with age between 18 and 25 -
This query will return all rows from the students table where the age column is between 18 and 25, inclusive.

```
SELECT *
FROM students
WHERE age BETWEEN 18 AND 25;
```

We can also combine BETWEEN with NOT logical operator.

- The NOT BETWEEN operator is the same as:
  - value < low OR value > high
  - value NOT BETWEEN low AND high

Example :

Query Editor | Query History

```
SELECT COUNT(*) FROM payment
WHERE amount NOT BETWEEN 8 AND 9;
```

Data Output | Explain | Messages | Notifications

count bigint
14157

- The BETWEEN operator can also be used with dates. Note that you need to format dates in the ISO 8601 standard format, which is YYYY-MM-DD
  - date BETWEEN '2007-01-01' AND '2007-02-01'

you need to be careful when using the BETWEEN operator with dates that also include timestamp information. This is because a datetime starts at 00:00:00, so if you simply use BETWEEN with two dates, you might not get the expected results.

- Example : For instance, let's say you have a table named Customers with columns for CustomerID, CustomerName, and RegisteredOn (datetime). You want to find all customers who registered between '2024-05-01' and '2024-05-02'. Here's what might go wrong:

```
SELECT *
FROM Customers
WHERE RegisteredOn BETWEEN '2024-05-01' AND '2024-05-02';
```

This query might return only customers who registered on exactly '2024-05-01' at 00:00:00 and '2024-05-02' at 00:00:00, excluding everyone who registered in between those times.

```
SELECT *
FROM Customers
WHERE RegisteredOn BETWEEN '2024-05-01 00:00:00' AND '2024-05-02 23:59:59';
```

9. IN

In SQL, the IN operator is used to filter rows based on a specific set of values. It allows you to specify multiple values for a column and retrieve rows where the column value matches any of the specified values

```
SELECT column1, column2, ...
FROM table_name
WHERE column_name IN (value1, value2, ...);
```

- IN: This is the operator used to specify the set of values.
- (value1, value2, ... valueN) : A comma-separated list of values to compare against the column. These values can be numbers, strings, or even subqueries in some cases
- **Matching Logic:** The IN operator checks if the value in the specified column matches any of the values in the provided list. It returns rows where there's a match.

Example 1 :

```
SELECT *
FROM students
WHERE age IN (18, 19, 20);
```

This query will return all rows from the students table where the age column matches any of the specified values (18, 19, or 20).

- You can use `IN` with subqueries to dynamically generate the list of values for comparison.
- `IN` can be used with multiple columns as well. Just add them to the `WHERE` clause with separate comparisons.
- The `NOT IN` operator can be used to filter for rows where the column value doesn't match any of the values in the list.

10. NOT IN

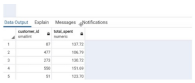
In SQL, the NOT IN operator is used to filter rows based on a specific set of values, excluding rows where the column value matches any of the specified values. It's essentially the negation of the IN operator.

```
SELECT column1, column2, ...
FROM table_name
WHERE column_name NOT IN (value1, value2, ...);
```

- `NOT IN` follows the same logic as `IN` but flips the matching condition to exclude rows where the column value matches any of the values in the list.
- You can use subqueries with `NOT IN` to create more dynamic filtering criteria based on your data.

```
SELECT *
FROM Employees
WHERE Department NOT IN ('Sales');
```

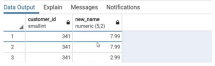
This query will return information for all employees where department is **not** "Sales". It could include departments like "Marketing"



```sql
SELECT customer_id,amount AS new_name
FROM payment
WHERE new_name > 2
```



```sql
SELECT customer_id,amount AS new_name
FROM payment
WHERE amount > 2
```




```sql
SELECT *
FROM Employees
WHERE Department NOT IN ('Sales');
```

This query will return information for all employees whose department is **not** "Sales". It could include departments like "Marketing", "Engineering", "Human Resources", and so on.

11 . LIKE and ILIKE

- In SQL, the LIKE operator is used to search for a specified pattern within a column. It's often used with wildcard characters (% for matching any sequence of characters and _ for matching any single character) to perform pattern matching.
- The ILIKE operator is similar to LIKE, but it performs a case-insensitive pattern matching

```sql
SELECT column1, column2, ...
FROM table_name
WHERE column_name LIKE pattern;
```

```sql
SELECT column1, column2, ...
FROM table_name
WHERE column_name ILIKE pattern;
```

- All names that begin with an 'A'
  - WHERE name LIKE 'A%'
- All names that end with an 'a'
  - WHERE name LIKE '%a'

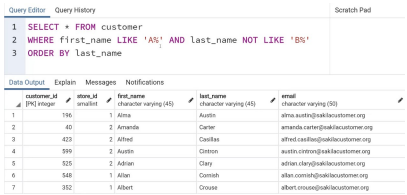
Example 1: LIKE - Case Sensitive

```sql
-- Using LIKE operator to find employees with a last name starting with 'S'
SELECT *
FROM employees
WHERE last_name LIKE 'S%';
```

Example 2: ILIKE - Case Insensitive

```sql
-- Using ILIKE operator to find employees with a first name containing 'john'
SELECT *
FROM employees
WHERE first_name ILIKE '%john%';
```

- We can also combine pattern matching operators to create more complex patterns
  - WHERE name LIKE '_her%'
    - Cheryl
    - Theresa
    - Sherri



first_name starts with 'A' and last_name does not starts with 'B' sort based on last_name in ascending order.

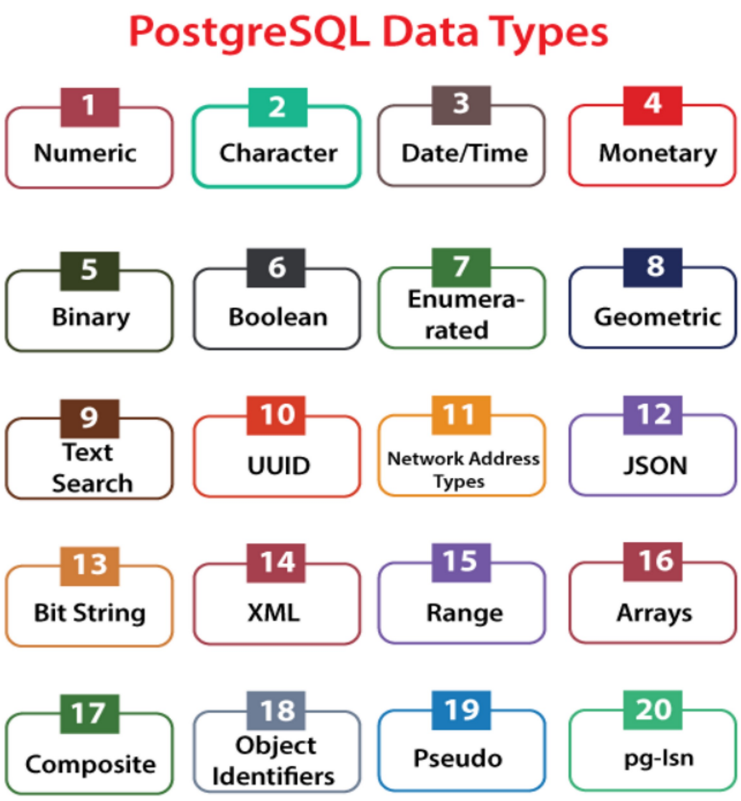
3. Data Types

21 April 2024 16:39

Data Types :

In PostgreSQL, each database table has many columns and has precise data types for each column. It supports an extensive range of data types.

- **Performance:** It enhances our performance if we are using these data types correctly and efficiently to store the data values.
- **Validation:** The correct use of data types involves the validation of data and dismissal of data outside the scope of the data type.
- **Compactness:** It stores efficiently, as a column can store a single type of value.
- **Consistency:** The activities in contradiction of columns of the same data type provide reliable results and are usually the fastest.



1. Numeric

- **Four and Eight byte floating-point numbers**
- **Two , Four and eight-byte integers**
- **Selectable-precision decimals**.

name	description	storage size	range
smallint	stores whole numbers, small range.	2 bytes	-32768 to +32767
integer	stores whole numbers. use this when you want to store typical integers.	4 bytes	-2147483648 to +2147483647
bigint	stores whole numbers, large range.	8 bytes	-9223372036854775808 to 9223372036854775807
decimal	user-specified precision, exact	variable	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point.
numeric	user-specified precision, exact	variable	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point.
real	variable-precision, inexact	4 bytes	6 decimal digits precision.
double precision	variable-precision, inexact	8 bytes	15 decimal digits precision
serial	auto incrementing integer	4 bytes	1 to 2147483647
bigserial	large auto incrementing integer	8 bytes	1 to 9223372036854775807

2. Characters:

Datatype	Explanation
char(size)	Here size is the number of characters to store. Fixed-length strings. Space padded on right to equal size characters.
character(size)	Here size is the number of characters to store. Fixed-length strings. Space padded on right to equal size characters.
varchar(size)	Here size is the number of characters to store. Variable-length string.
character varying(size)	Here size is the number of characters to store. Variable-length string.
text	Variable-length string.

3. Date Time

Name	Description	Storage size	Minimum value	Maximum value	Resolution
timestamp [(p)] [without time zone]	both date and time (no time zone)	8 bytes	4713 bc	294276 ad	1 microsecond / 14 digits
timestamp [(p)] with time zone	both date and time, with time zone	8 bytes	4713 bc	294276 ad	1 microsecond / 14 digits
date	date (no time of day)	4 bytes	4713 bc	5874897 ad	1 day
time [(p)] [without time zone]	time of day (no date)	8 bytes	00:00:00	24:00:00	1 microsecond / 14 digits
time [(p)] with time zone	times of day only, with time zone	12 bytes	00:00:00+1459	24:00:00-1459	1 microsecond / 14 digits

interval [fields] [(p)]	time interval	12 bytes	-178000000 years	178000000 years	1 microsecond / 14 digits

4. Monetry Type :

Name	Description	Storage size	Range
money	currency amount	8 bytes	-92233720368547758.08 to +92233720368547758.07

5. Binary :

Name	Storage size	Description
bytea	1 or 4 bytes plus the actual binary string	variable-length binary string

6. Boolean :

Name	Description	Storage size
boolean	it specifies the state of true or false.	1 byte

7. Geometric Data Type :

Name	Storage size	Representation	Description
point	16 bytes	point on a plane	(x, y)
line	32 bytes	infinite line (not fully implemented)	((x1,y1),(x2,y2))
lseg	32 bytes	finite line segment	((x1,y1),(x2,y2))
box	32 bytes	rectangular box	((x1,y1),(x2,y2))
path	16+16n bytes	closed path (similar to polygon)	((x1,y1),...)
path	16+16n bytes	open path	[(x1,y1),...]
polygon	40+16n	polygon (similar to closed path)	((x1,y1),...)
circle	24 bytes	circle	<(x, y),r> (center point and radius)

8. Text Search Data Type

Data types	Description
tsvector	It is used to display a document in a form, which enhance text search.
tsquery	It is used to represent a text query.

9. Network Address Data Type

Data type	Description	Storage Size
inet	It stores the IPv4 and IPv6 hosts and networks.	7 or 19 bytes
cidr	It is used to store the IPv4 and IPv6 networks.	7 or 19 bytes
macaddr	It stores the MAC addresses.	6 bytes

While using the data types, we can refer to the following points:

- If we have an IEEE 754 data source, we can use the **float data type**
- For integer data type, we can use **int**.
- Never use **char**.

- If we want to limit the input, we can apply a **text** data type.
- When we have vast numbers, we can use **bigint** only.

4. Database

21 April 2024 20:59

Databases are systems that allow users
to store and organise data.
And they're useful when dealing with large amounts of data.

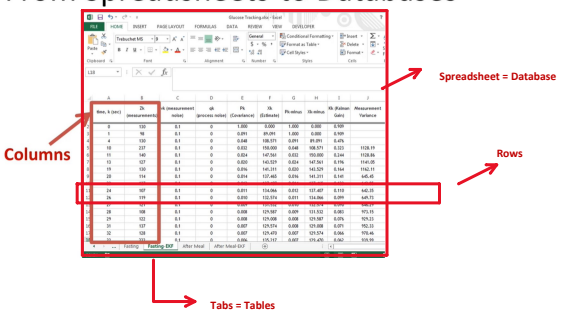
Typical Users

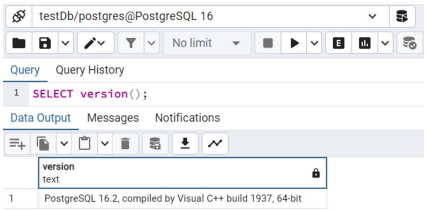
- Databases have a wide variety of users!
- Analysts
  - Marketing
  - Business
  - Sales
- Technical
  - Data Scientist
  - Software Engineers
  - Web Developers
- Basically anyone needing to deal with data!

From Spreadsheets to Databases

- Databases
  - Data Integrity
  - Can handle massive amounts of data
  - Quickly combine different datasets
  - Automate steps for re-use
  - Can support data for websites and applications

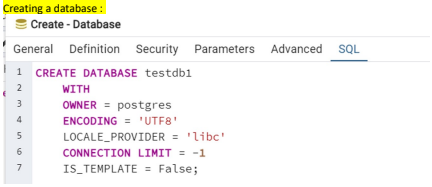
From Spreadsheets to Databases





- **CREATE DATABASE**

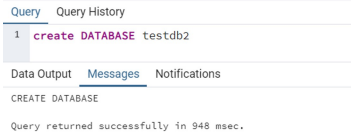
Creating a database :



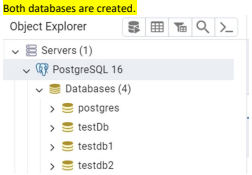
Parameters	Description
db_name	• We will use this parameter to specify the new database name, which we want to create. • And we also ensure that the database must be unique because If we try to create a new database with the same name as an existing database, it will show an error.
role_name	It is used to describe the role name for the user who will have the new database, and by default, it is **postgres.**
Template	While creating the new database, we will require database template name.
Encoding	It is used to describe the character set encoding for the new database, and by default, it is **UTF8**.
Collate	It is used to define the sort order of strings that mark the result of the ORDER BY clause if we are using a SELECT statement.
Ctype	This parameter is used to display the character classification for the new database.
tablespace_name	It is used to define the tablespace name for the new database, and by default, it is the **template database's tablespace.**
max_concurrent_connection	This parameter is used to define the maximum parallel connections of a new database, and by default, **it is -1 (unlimited).**

From <https://www.javatpoint.com/postgresql-create-database>

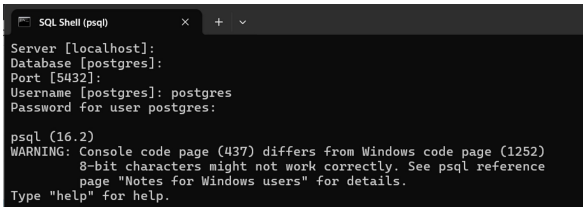
Creating another database: (with default options)

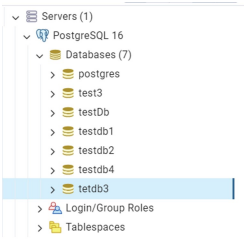


Both databases are created.



Or Create from SQL shell





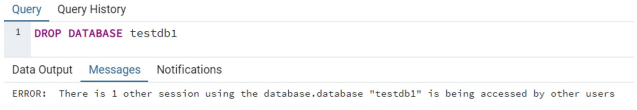
Get List of all databases :

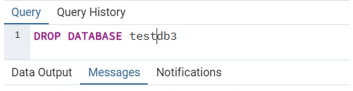
- **DROP DATABASE**

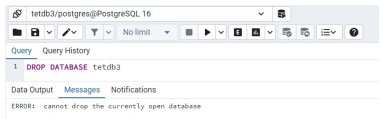
The **Drop/delete** command is used to eternally delete all the file entries and data directory from the PostgreSQL platform
>> DROP DATABASE [IF EXISTS] name

1. **IF EXISTS**
   Do not throw an error if the database does not exist. A notice is issued in this case.
2. **name**
   The name of the database to remove.

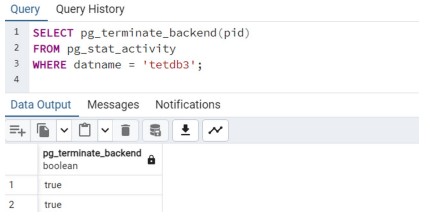
From <https://www.tutorialspoint.com/postgresql/postgresql_drop_database.htm>







1. To drop, first, we have to revoke the connection
   REVOKE CONNECT ON DATABASE db_name **from public**;
2. Disconnect other sessions: Ask other users who are connected to the database to disconnect or close their sessions.
3. Terminate active connections: If you have the necessary permissions, you can terminate active connections to the database using the pg_terminate_backend() function. First, identify the process ID (PID) of the active connections using the following query:



4. Connect to a different database: Connect to a different database using the psql command or any other PostgreSQL client tool.

Tablespaces

Get List of all databases :

```
postgres=# \l
                                                List of databases
   Name    |  Owner   | Encoding | Locale Provider |   Collate   |    Ctype    | ICU Locale | ICU Rules |   Access privileges
-----------+----------+----------+-----------------+-------------+-------------+------------+-----------+-----------------------
 postgres  | postgres | UTF8     | libc            | English_India.1252 | English_India.1252 |            |           |
 template0 | postgres | UTF8     | libc            | English_India.1252 | English_India.1252 |            |           | =c/postgres          +
           |          |          |                 |             |             |            |           | postgres=CTc/postgres
 template1 | postgres | UTF8     | libc            | English_India.1252 | English_India.1252 |            |           | =c/postgres          +
           |          |          |                 |             |             |            |           | postgres=CTc/postgres
 test3     | postgres | UTF8     | libc            | English_India.1252 | English_India.1252 |            |           |
 testDb    | postgres | UTF8     | libc            | English_India.1252 | English_India.1252 |            |           |
 testdb1   | postgres | UTF8     | libc            | English_India.1252 | English_India.1252 |            |           |
 testdb2   | postgres | UTF8     | libc            | English_India.1252 | English_India.1252 |            |           |
 testdb4   | postgres | UTF8     | libc            | English_India.1252 | English_India.1252 |            |           |
 tetdb3    | postgres | UTF8     | libc            | English_India.1252 | English_India.1252 |            |           |
(9 rows)
```

Connect to database :

```
postgres=# \c testDb1
connection to server at "localhost" (::1), port 5432 failed: FATAL:  database "testDb1" does not exist
Previous connection kept
postgres=# \c testdb1
You are now connected to database "testdb1" as user "postgres".
testdb1=#
```

To list all the tables in the database, use the below command :

\dt

Quit shell:

\q or \quit

- **Create Database remotely**

  # createdb [option...] [dbname [description]]

S. No.	Parameter & Description
1	**dbname** The name of a database to create.
2	**description** Specifies a comment to be associated with the newly created database.
3	**options** command-line arguments, which createdb accepts.

Option :

S. No.	Option & Description
1	**-D tablespace** Specifies the default tablespace for the database.
2	**-e** Echo the commands that createdb generates and sends to the server.
3	**-E encoding** Specifies the character encoding scheme to be used in this database.
4	**-l locale** Specifies the locale to be used in this database.
5	**-T template** Specifies the template database from which to build this database.
6	**--help** Show help about createdb command line arguments, and exit.
7	**-h host** Specifies the host name of the machine on which the server is running.
8	**-p port** Specifies the TCP port or the local Unix domain socket file extension on which the server is listening for connections.
9	**-U username** User name to connect as.
10	**-w** Never issue a password prompt.
11	**-W** Force createdb to prompt for a password before connecting to a database.

Example :

```
# createdb -h localhost -p 5432 -U postgres testdb
  password ******
```

- Select a database remotely

  # plsql -h host -p port -U userName dbName

```
psql -h localhost -p 5432 -U postgress testdb
Password for user postgress: ****
psql (9.2.4)
Type "help" for help.
You are now connected to database "testdb" as user "postgres".
testdb=#
```

	pg_terminate_backend boolean
1	true
2	true

4. Connect to a different database: Connect to a different database using the psql command or any other PostgreSQL client tool. You cannot drop the currently connected database.
5. Retry dropping the database: Once all connections to the database have been terminated, retry dropping the database using the DROP DATABASE statement.

Query | Query History

```
1  DROP DATABASE tetdb3
```

Data Output | Messages | Notifications

```
DROP DATABASE

Query returned successfully in 196 msec.
```

6. Trying to delete same database again.
   Got an error

Query | Query History

```
1  DROP DATABASE tetdb3
```

Data Output | Messages | Notifications

```
ERROR:  database "tetdb3" does not exist

SQL state: 3D000
```

7. Using IF EXISTS while deleting database.
   It will not give error, if database dosen't exist it will skip the query.

Query | Query History

```
1  DROP DATABASE IF EXISTS tetdb3;
```

Data Output | Messages | Notifications

```
NOTICE:  database "tetdb3" does not exist, skipping
DROP DATABASE

Query returned successfully in 82 msec.
```

Drop a Database Remotely :

dropdb [option...] dbname

S. No.	Parameter & Description
1	**dbname** The name of a database to be deleted.
2	**option** command-line arguments, which dropdb accepts.

S. No.	Option & Description
1	**-e** Shows the commands being sent to the server.
2	**-i** Issues a verification prompt before doing anything destructive.
3	**-V** Print the dropdb version and exit.
4	**--if-exists** Do not throw an error if the database does not exist. A notice is issued in this case.
5	**--help** Show help about dropdb command-line arguments, and exit.
6	**-h host** Specifies the host name of the machine on which the server is running.
7	**-p port** Specifies the TCP port or the local UNIX domain socket file extension on which the server is listening for connections.
8	**-U username** User name to connect as.
9	**-w** Never issue a password prompt.
10	**-W** Force dropdb to prompt for a password before connecting to a database.
11	**--maintenance-db=dbname** Specifies the name of the database to connect to in order to drop the target database.

dropdb -h host -p port -U username dbName

```
dropdb -h localhost -p 5432 -U postgress testdb
Password for user postgress: ****
```

5. Tables

22 April 2024 22:50

1. Creating a table

```
CREATE TABLE table_name (
    column1 datatype constraints,
    column2 datatype constraints,
    ...
);
```

Ex :

```
CREATE TABLE inventory (
    serialN0 INT PRIMARY KEY,
    deviceType VARCHAR(10),
    deviceValue NUMERIC(5,2)
)
```

Data Output | Messages | Notifications

```
CREATE TABLE

Query returned successfully in 319 msec.
```

Display All tables of a database.

Ex: displaying all tables of testDb

```
SELECT table_name
FROM information_schema.tables
WHERE table_schema = 'public'
AND table_catalog = 'testDb'
AND table_type = 'BASE TABLE';
```

	table_name name
1	inventory

- `SELECT table_name`: This selects the table_name column from the information_schema.tables view.
- `FROM information_schema.tables`: This specifies the view from which to retrieve information about tables.
- `WHERE table_schema = 'public'`: This filters the results to only include tables in the specified schema. Change 'public' to the name of your schema if it's different.
- `AND table_catalog = 'testDb'`: This filters the results to only include tables in the specified database. Replace 'testDb' with the name of your database if it's different.
- `AND table_type = 'BASE TABLE'`: This further filters the results to only include base tables (not views or system tables).

Creating another table myTable

testDb/postgres@PostgreSQL 16

```
CREATE TABLE myTable (
    h1 INT,
    h2 VARCHAR,
    h3 NUMERIC
)
```

Data Output | Messages | Notifications

```
CREATE TABLE

Query returned successfully in 104 msec.
```

Displaying tables again

```
SELECT table_name
from information_schema.tables
Where table_schema = 'public'
AND table_catalog = 'testDb'
AND table_type = 'BASE TABLE'
```

	table_name name
1	inventory
2	mytable

Inheriting a table

- you can use table inheritance to create a new table that inherits the structure (columns, data types, constraints, etc.) of an existing table.
- This is useful when you want to create a new table that shares the same structure as an existing one but with potentially additional columns or constraints.

Example : we have an existing table called employees

```
CREATE TABLE employees (
    emp_id SERIAL PRIMARY KEY,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
```

Creating usign psql :

>> \c db_name - to connect to that database

```
testDb-# \c
You are now connected to database "testDb" as user "postgres".
testDb-#
```

>>

```
testDb=# create table Student(id int, name text,age int, adress char(30));
CREATE TABLE
```

>> \d or \dt to check the table relations in a particular database

```
testDb=# \d
          List of relations
 Schema |   Name    | Type  |  Owner
--------+-----------+-------+----------
 public | inventory | table | postgres
 public | mytable   | table | postgres
 public | student   | table | postgres
(3 rows)
```

>> Create same table again - error

```
testDb=# create table Student(id int, name text,age int, adress char(30));
ERROR:  relation "student" already exists
```

>> we can define some of the essential lists of parameters that we use while creating a table

Parameter	Description
If not exists	If a table already occurs with a similar name, a warning will be displayed in place of an error.
Unlogged	This parameter does not enter data into the write-ahead log (WAL) because of the deletion of this further IO operation, write performance is improved.
Of_type_name	In this parameter, a table can have structure from the defined composite type.
Temporary or Temp	It is used to generate a temporary table, and it will delete after the existing operation or at the end of a session.

>> dropping a table

DROP TABLE table_name;

```
DROP TABLE mytable;
```

Data Output | Messages | Notifications

```
DROP TABLE

Query returned successfully in 87 msec.
```

```
testDb-# \dt
```

Example : we have an existing table called employees

```
CREATE TABLE employees (
    emp_id SERIAL PRIMARY KEY,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    department VARCHAR(50)
);
```

let's create a new table called manager that inherits from the employees table

```
CREATE TABLE manager (
    bonus DECIMAL(10, 2),
    join_date DATE
) INHERITS (employees);
```

- We create a new table called manager.
- It has two additional columns: bonus and join_date.
- We use the INHERITS clause to specify that the manager table inherits the structure of the employees table.

Now, the manager table has all the columns and constraints of the employees table plus the additional columns bonus and join_date.

```
INSERT INTO employees (first_name, last_name, department)
VALUES
    ('John', 'Doe', 'IT'),
    ('Alice', 'Smith', 'Sales');
```

```
INSERT INTO manager (first_name, last_name, department, bonus, join_date)
VALUES ('Alice', 'Smith', 'Sales', 5000.00, '2024-06-01');
```

```
-- Query from the employees table
SELECT * FROM employees;
```

emp_id	first_name	last_name	department
1	John	Doe	IT
2	Alice	Smith	Sales

```
-- Query from the manager table
SELECT * FROM manager;
```

emp_id	first_name	last_name	department	bonus	join_date
2	Alice	Smith	Sales	5000.00	2024-06-01

Note :

- Inherited tables inherit all columns, indexes, constraints (including foreign keys), and triggers from the parent table.
- When you insert data into the parent table, the data automatically appears in the child table(s) as well.
- However, deleting data from the parent table does not cascade to the child table(s).
- You can still modify the structure of the child table independently from the parent table. Any changes made to the parent table's structure will not affect the child table(s).

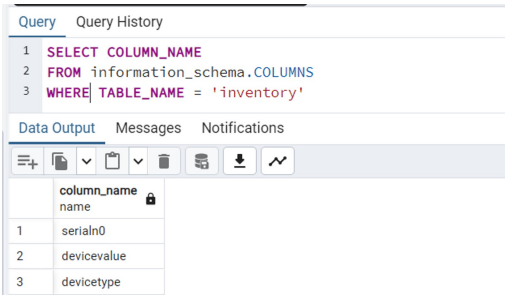
```
DROP TABLE

Query returned successfully in 87 msec.
```

```
testDb-# \dt
          List of relations
 Schema |   Name    | Type  |  Owner
--------+-----------+-------+----------
 public | inventory | table | postgres
(1 row)
```

Describe a table

we use the **information_schema** for describing the tables.

Here, the **information schema** itself is a schema that is automatically present in all databases and called **information_schema**.



Or

>> \d table_name

```
testDb-# \d inventory
                        Table "public.inventory"
   Column    |          Type          | Collation | Nullable | Default
-------------+------------------------+-----------+----------+---------
 serialn0    | integer                |           | not null |
 devicetype  | character varying(10)  |           |          |
 devicevalue | numeric(5,2)           |           |          |
Indexes:
    "inventory_pkey" PRIMARY KEY, btree (serialn0)
```

6. Aggregate Functions

06 May 2024 22:15

Aggregate functions in SQL are functions that perform a calculation on a set of values and return a single value. These functions are commonly used with the SELECT statement to perform calculations across rows in a table

They are often used in conjunction with the GROUP BY clause to analyze data grouped by specific criteria

1. COUNT: Returns the number of rows in a specified column or in a result set.

```sql
SELECT COUNT(*) AS total_students
FROM students;
```

2. SUM: Calculates the sum of values in a specified column.

```sql
SELECT SUM(salary) AS total_salary
FROM employees;
```

3. AVG: Calculates the average of values in a specified column.

```sql
SELECT AVG(age) AS average_age
FROM customers;
```

4. MIN: Returns the minimum value in a specified column.

```sql
SELECT MIN(salary) AS min_salary
FROM employees;
```

5. MAX : Returns the maximum value in a specified column.

```sql
SELECT MAX(salary) AS max_salary
FROM employees;
```

6. ROUND : is used to round a numeric value to a specified number of decimal places. It's often used to make numerical values more readable or to remove extraneous decimal places

```sql
ROUND(number, decimal_places)
```

   - number: This is the numeric value you want to round.
   - decimal_places: This is the number of decimal places to which you want to round the number. If omitted, it defaults to 0.

   Example :

```sql
SELECT ROUND(temperature, 2) AS rounded_temperature
FROM temperatures;
```

   The result set will contain the temperature values rounded to two decimal places, making them easier to read and interpret.

HAVING :

The HAVING clause in SQL is used to filter rows after groups have been created using the GROUP BY clause. It's similar to the WHERE clause, but the HAVING clause applies to groups of rows rather than individual rows
It allows you to apply conditions to the aggregate results calculated for each group, refining the data you retrieve.

```sql
SELECT column1, aggregate_function(column2), ...
FROM table_name
GROUP BY column1
HAVING condition;
```

- HAVING: This keyword is followed by a condition that must be met for a group to be included in the result set.

1. **Placement:** The `HAVING` clause comes after the `GROUP BY` clause in your `SELECT` statement.

Group By with Aggregate functions :

The GROUP BY clause in SQL is a powerful tool used to organize and summarize data based on specific criteria.

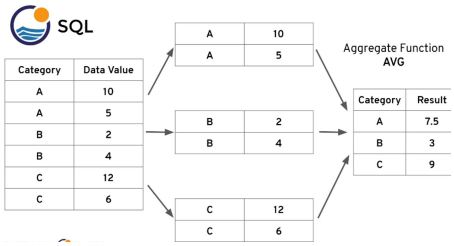
Purpose:

- Groups rows in a result set that have the same values in one or more columns.
- Allows you to perform aggregate functions (like `COUNT`, `SUM`, `AVG`) on these grouped data sets.

How it Works:

1. **Grouping:** The `GROUP BY` clause identifies rows with identical values in the specified columns and categorizes them into distinct groups.
2. **Aggregate Functions:** You can then use aggregate functions to calculate summary statistics for each group. These functions operate on the values within each group, providing insights into trends and patterns across the data.

- SELECT category_col , AGG(data_col)
  FROM table
  GROUP BY category_col



Example 1 :

- SELECT company, division, SUM(sales)
  FROM finance_table
  GROUP BY company,division

- In the SELECT statement, columns must either have an aggregate function or be in the GROUP BY call.

Example 2:

- SELECT company, division, SUM(sales)
  FROM finance_table
  WHERE division IN ('marketing', 'transport')
  GROUP BY company,division

- WHERE statements should not refer to the aggregation result, later on we will learn to use HAVING to filter on those results.

Example 3:

- SELECT company, SUM(sales)
  FROM finance_table
  GROUP BY company
  ORDER BY SUM(sales)

- If you want to sort results based on the aggregate, make sure to reference the entire function

- You can group by multiple columns, creating nested groups for more granular analysis.
- The `HAVING` clause can be used with `GROUP BY` to filter the groups based on specific conditions applied to the aggregate results.
- `GROUP BY` is often used in conjunction with `SELECT` and `WHERE` clauses to retrieve and organize data effectively.

Example 1: customer payment trnsaction table with customer Id and paymentId as PK (primary key)

Query Editor | Query History | Scratch Pad

```sql
SELECT * FROM payment
```

the result set.

1. **Placement:** The `HAVING` clause comes after the `GROUP BY` clause in your `SELECT` statement.
2. **Filtering Groups:** You specify a condition within `HAVING` using comparison operators and aggregate functions. This condition filters the groups based on the calculated aggregate values.

Example :

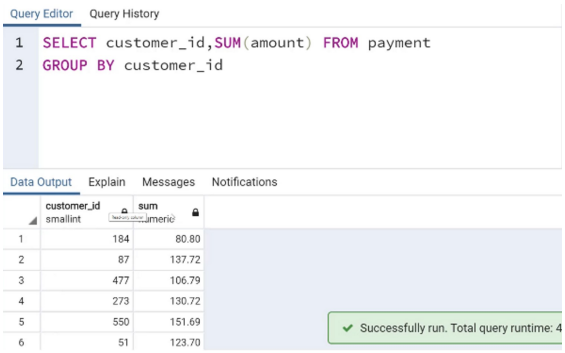
```
SELECT category, SUM(quantity) AS total_quantity
FROM orders
GROUP BY category
HAVING SUM(quantity) > 100;
```

- We're selecting the category column and calculating the total quantity (SUM(quantity)) of each product category from the orders table.
- We're using the GROUP BY clause to group the rows by the category column.
- We're using the HAVING clause to filter out groups where the total quantity is greater than 100.

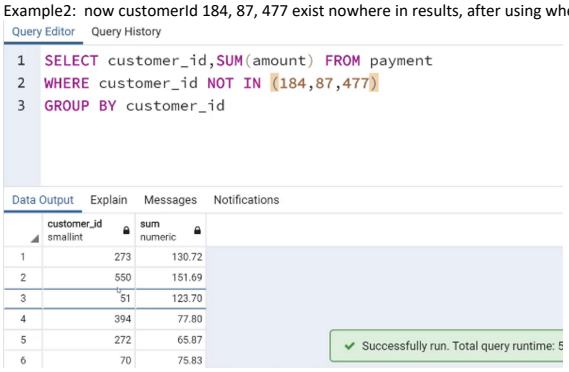
- The result will be a summary of the total quantity for each product category, but only for categories where the total quantity is greater than 100.
- The HAVING clause allows you to apply conditions to the grouped data, providing additional filtering capabilities beyond what is available with the WHERE clause.

- You can use multiple conditions within `HAVING` connected by logical operators like `AND` or `OR` to create more complex filtering criteria.
- `HAVING` is particularly useful when you want to analyze specific characteristics within your grouped data.

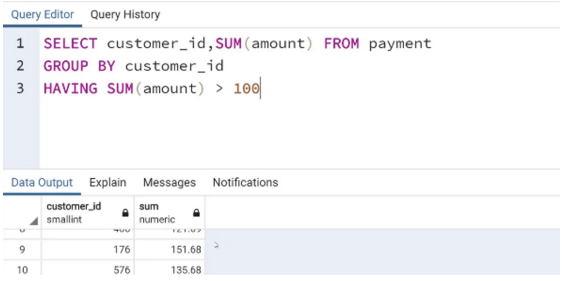
Example1 : list customerIds and sum of payment for each customer.

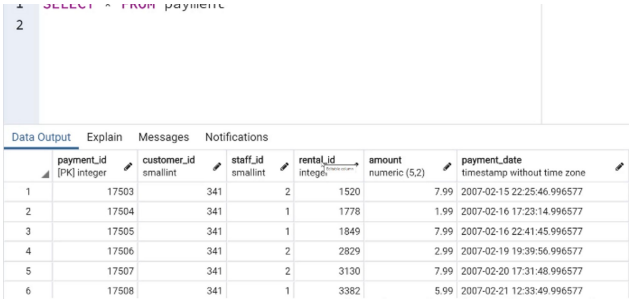


Example2: now customerId 184, 87, 477 exist nowhere in results, after using where condition

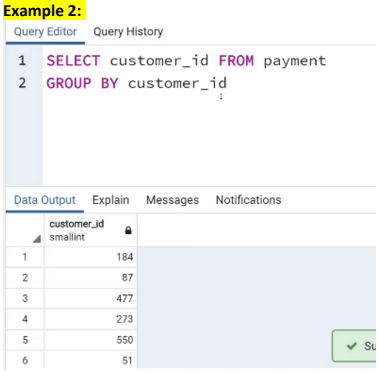


Example3: what we cannot do is filter on where same statement on SUM(amount), because sum of amount is not going to happen until we call GROUP BY.
Use HAVING to filter on AGGREGATE of GROUP BY.



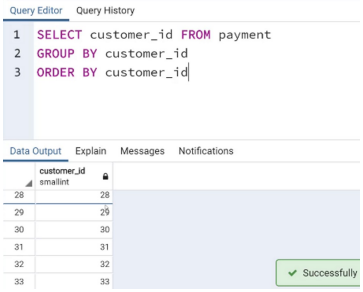


Example 2:



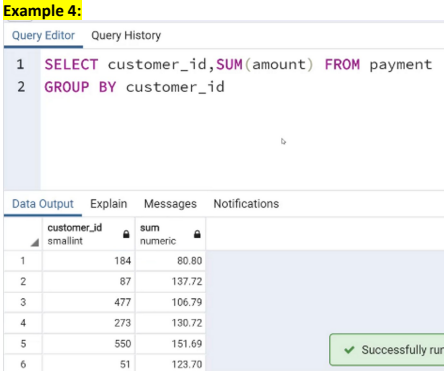
This is same as selecting the distinct customerIds

Example3 :



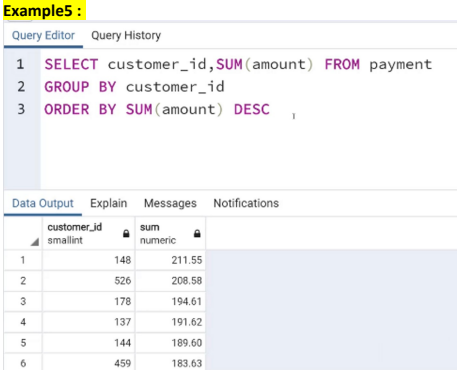
This is same as selecting all distinct customerIds in Ascending order.

Example 4:



Total sum that every customer spends.

Example5 :



This is about which customerId is spending most of money.

Data Output | Explain | Messages | Notifications

	customer_id smallint	sum numeric
9	176	151.68
10	576	135.68
11	309	113.75
12	292	102.78
13	529	115.72

Successfully run. Total query runtime:

Similarly, another example,

Query Editor | Query History

```
1 SELECT store_id,COUNT(*) FROM customer
2 GROUP BY store_id
```

Data Output | Explain | Messages | Notifications

	store_id smallint	count bigint
1	1	326
2	2	273

Query Editor | Query History

```
1 SELECT store_id,COUNT(*) FROM customer
2 GROUP BY store_id
3 HAVING COUNT(*) > 300
```

Data Output | Explain | Messages | Notifications

	store_id smallint	count bigint
1	1	326

Same as,

Query Editor | Query History

```
1 SELECT store_id,COUNT(customer_id) FROM customer
2 GROUP BY store_id
3 HAVING COUNT(customer_id) > 300
```

Data Output | Explain | Messages | Notifications

	store_id smallint	count bigint
1	1	326

4	137	191.62
5	144	189.60
6	459	183.63

This is about which customerId is spending most of money.

Example6:

Query Editor | Query History

```
1 SELECT customer_id, COUNT(amount) FROM payment
2 GROUP BY customer_id
3 ORDER BY COUNT(amount) DESC
```

Data Output | Explain | Messages | Notifications

	customer_id smallint	count bigint
1	148	45
2	526	42
3	144	40
4	75	39
5	236	39
6	178	39

Successfully run. Tota

CustomerId having most number of tramsactions.

Example7:

Query Editor | Query History

```
1 SELECT customer_id,staff_id,SUM(amount) FROM payment
2 GROUP BY staff_id,customer_id
3 ORDER BY customer_id
```

Data Output | Explain | Messages | Notifications

	customer_id smallint	staff_id smallint	sum numeric
1	1	2	53.85
2	1	1	60.85
3	2	2	67.88
4	2	1	55.86
5	3	1	59.88
6	3	2	70.88

Successfully run. Total query runt

This is about which customer spends most amount with which staff.

7. Joins

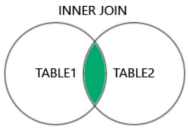
25 May 2024 12:06

SQL joins are a fundamental concept for retrieving data from multiple related tables in a database. They allow you to combine data sets based on a shared column and generate a more comprehensive result.

1. Inner Join (Default Join)

- The INNER JOIN clause in SQL is used to combine rows from two or more tables based on a related column between them. It returns only the rows that have matching values in both tables. If there is no match, the row is excluded from the result set.
- It essentially creates a new result set that includes only rows where there's a match between the join condition in both tables

```
SELECT columns
FROM table1
INNER JOIN table2
ON table1.common_column = table2.common_column;
```



Example: Suppose we have two tables, employees and departments, We want to retrieve the first and last names of employees along with their department names

employees table:

employee_id	first_name	last_name	department_id
1	John	Doe	101
2	Jane	Smith	102
3	Alice	Johnson	101
4	Bob	Brown	103
5	Charlie	White	104

departments table:

department_id	department_name
101	IT
102	HR
103	Finance
105	Marketing

```
SELECT employees.first_name, employees.last_name, departments.department_name
FROM employees
INNER JOIN departments ON employees.department_id = departments.department_id;
```

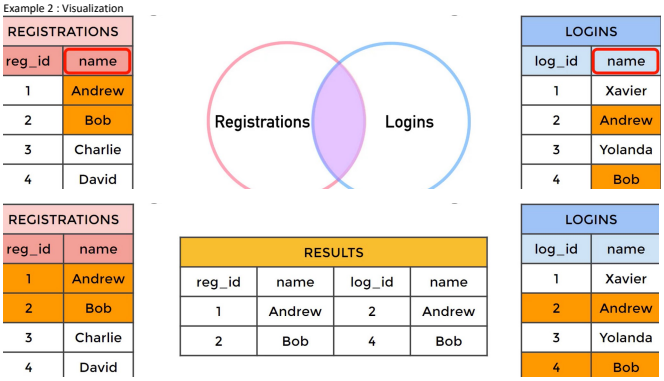
Or

```
SELECT e.first_name, e.last_name, d.department_name
FROM employees AS e
INNER JOIN departments AS d
ON e.department_id = d.department_id;
```

Result set:

first_name	last_name	department_name
John	Doe	IT
Alice	Johnson	IT
Jane	Smith	HR
Bob	Brown	Finance

Example 2 : Visualization




```
SELECT reg_id,Logins.name,log_id
FROM Registrations
INNER JOIN Logins
ON Registrations.name = Logins.name
```

RESULTS

reg_id	name	log_id
1	Andrew	2
2	Bob	4

Note:

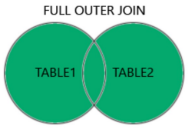
a. table order won't matter in inner join
b. If you use join without inner, POSTGRE SQL will treat it as inner join

2. Outer Joins
They will allow us to specify how to deal with valuesonly present with one of the tables being joined

- Full Outer Join
- Left Outer Join
- Right Outer Join

SQL Outer Joins are used to include rows from one or both tables that do not have matching rows in the other table.

1. FULL OUTER JOIN



- A Full Outer Join returns all rows from both tables. When there is no match, the result is NULL on the side that does not have a match.
- Table Order does not matter in full outer join

```
SELECT columns
FROM table1
FULL OUTER JOIN table2
ON table1.common_column = table2.common_column;
```

Example : Let's assume the following employees and departments tables:

- Employees Table:

employee_id	first_name	last_name	department_id
1	John	Doe	101
2	Jane	Smith	102
3	Alice	Johnson	101
4	Bob	Brown	103
5	Charlie	White	104

- Departments Table:

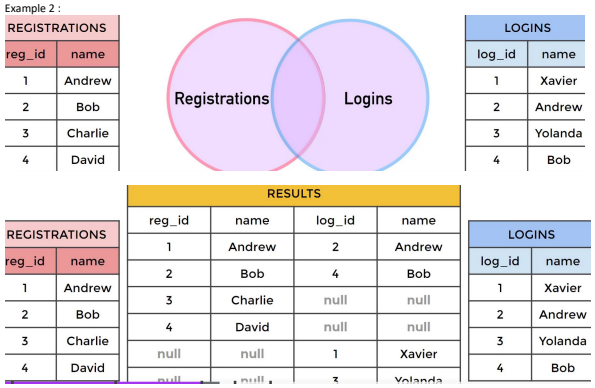
department_id	department_name
101	IT
102	HR
103	Finance
105	Marketing

- return first_name, last_name, and department_name

```
SELECT e.first_name, e.last_name, d.department_name
FROM employees AS e
FULL OUTER JOIN departments AS d
ON e.department_id = d.department_id;
```

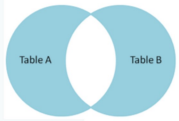
first_name	last_name	department_name
John	Doe	IT
Alice	Johnson	IT
Jane	Smith	HR
Bob	Brown	Finance
Charlie	White	NULL
NULL	NULL	Marketing

Example 2 :



3. **Full Outer Join with where condition :**
This can be used to filter results after performing the join, including cases where there are unmatched rows from either table. This can be useful for getting results that are the opposite of an INNER JOIN, meaning it will include rows that do not have matching values in the related table (i.e., those that would have been excluded by an INNER JOIN).

```
SELECT * FROM TableA
FULL OUTER JOIN TableB
ON TableA.name = TableB.name
WHERE TableA.id IS null
OR TableB.id IS null
```



To produce the set of records unique to Table A and Table B, we perform the same full outer join, then **exclude the records we don't want from both sides via a where clause**.

Example: Let's use the same employees and departments tables:

- Employees Table:

employee_id	first_name	last_name	department_id
1	John	Doe	101
2	Jane	Smith	102
3	Alice	Johnson	101
4	Bob	Brown	103
5	Charlie	White	104

- Departments Table:

department_id	department_name
101	IT
102	HR
103	Finance
105	Marketing

To get rows that are the opposite of what an INNER JOIN would return (i.e., rows that do not match in both tables), we can use a FULL OUTER JOIN and then filter out the matching rows with a WHERE condition:

```
SELECT e.first_name, e.last_name, d.department_name
```

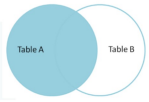
4. Left Outer Join :

- Includes all rows from the left table (specified before LEFT JOIN).
- Matches rows from the right table based on the join condition.
- If there's no match in the right table for a row in the left table, the corresponding columns from the right table will contain NULL values.
- Order matters for left outer join

```
SELECT columns
FROM table1
LEFT JOIN table2
ON table1.common_column = table2.common_column;
```

```
SELECT * FROM TableA
LEFT OUTER JOIN TableB
```

```
SELECT * FROM TableA
LEFT OUTER JOIN TableB
ON TableA.name = TableB.name
```



Example: Suppose we have two tables, employees and departments

employees table:

employee_id	first_name	last_name	department_id
1	John	Doe	101
2	Jane	Smith	102
3	Alice	Johnson	101
4	Bob	Brown	103
5	Charlie	White	104

departments table:

department_id	department_name
101	IT
102	HR
103	Finance
105	Marketing

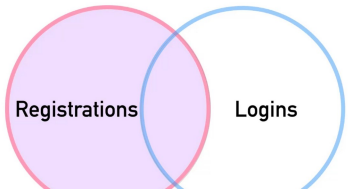
```
SELECT e.first_name, e.last_name, d.department_name
FROM employees AS e
LEFT OUTER JOIN departments AS d
ON e.department_id = d.department_id;
```

- LEFT OUTER JOIN: Returns all rows from the employees table (left table).
- Matched Rows: Where there is a match on department_id, it returns the corresponding department_name.
- Unmatched Rows: If there is no match, it returns NULL for columns from the departments table.

- Entire Left Circle: All rows from employees are included.
- Overlap: Matching rows have corresponding values from departments.
- Non-Overlapping Left Area: Unmatched rows from employees have NULL in columns from departments.

Example 2:

REGISTRATIONS	
reg_id	name
1	Andrew
2	Bob
3	Charlie
4	David



LOGINS	
log_id	name
1	Xavier
2	Andrew
3	Yolanda
4	Bob

```
SELECT * FROM Registrations
LEFT OUTER JOIN Logins
ON Registrations.name = Logins.name
```

REGISTRATIONS	
reg_id	name
1	Andrew
2	Bob
3	Charlie
4	David

RESULTS			
reg_id	name	log_id	name
1	Andrew	2	Andrew
2	Bob	4	Bob
3	Charlie	null	null
4	David	null	null

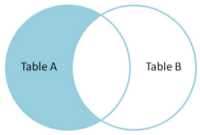
LOGINS	
log_id	name
1	Xavier
2	Andrew
3	Yolanda
4	Bob

5. Left outer Join with WHERE condition

- Get rows unique to left tables (rows found in table 1, not found in table 2)

  To produce the set of records only in Table A, but not in Table B, we perform the same left outer join, then **exclude the records we don't want from the right side via a where clause**.

```
SELECT * FROM TableA
LEFT OUTER JOIN TableB
ON TableA.name = TableB.name
WHERE TableB.id IS null
```



Example:

employees table:

employee_id	first_name	last_name	department_id
1	John	Doe	101
2	Jane	Smith	102
3	Alice	Johnson	101
4	Bob	Brown	103
5	Charlie	White	104

departments table:

department_id	department_name
101	IT
102	HR
103	Finance
105	Marketing

```
SELECT e.first_name, e.last_name, e.department_id
FROM employees AS e
LEFT OUTER JOIN departments AS d
ON e.department_id = d.department_id
WHERE d.department_name IS NULL;
```

- LEFT OUTER JOIN: This will join all rows from the employees table with matched rows from the departments table based on department_id.
- Perform the LEFT OUTER JOIN:

Joined Result Before WHERE Condition:

first_name	last_name	department_id	department_name
John	Doe	101	IT
Alice	Johnson	101	IT
Jane	Smith	102	HR
Bob	Brown	103	Finance
Charlie	White	104	NULL

- Apply the WHERE Condition:
- Filter to include only rows where department_name is NULL.

first_name	last_name	department_id
Charlie	White	104

- Charlie White: Has a department_id of 104, which does not match any department_id in the departments table, resulting in NULL for

5	Charlie	White	104	105	Marketing

To get rows that are the opposite of what an INNER JOIN would return (i.e., rows that do not match in both tables), we can use a FULL OUTER JOIN and then filter out the matching rows with a WHERE condition:

```
SELECT e.first_name, e.last_name, d.department_name
FROM employees AS e
FULL OUTER JOIN departments AS d
ON e.department_id = d.department_id
WHERE e.department_id IS NULL OR d.department_id IS NULL;
```

Joined Result Before WHERE Condition:

first_name	last_name	department_name
John	Doe	IT
Alice	Johnson	IT
Jane	Smith	HR
Bob	Brown	Finance
Charlie	White	NULL
NULL	NULL	Marketing

Result After WHERE Condition:

first_name	last_name	department_name
Charlie	White	NULL
NULL	NULL	Marketing

- Left Circle (employees): All employees.
- Right Circle (departments): All departments.
- Overlap: Employees that have a matching department.
- Non-Overlapping Areas: Employees without a department and departments without employees.

Example 2:

```
SELECT * FROM Registrations FULL OUTER JOIN Logins
ON Registrations.name = Logins.name
```

REGISTRATIONS	
reg_id	name
1	Andrew
2	Bob
3	Charlie
4	David
null	null

RESULTS			
reg_id	name	log_id	name
1	Andrew	2	Andrew
2	Bob	4	Bob
3	Charlie	null	null
4	David	null	null
null	null	1	Xavier
null	null	3	Yolanda

LOGINS	
log_id	name
1	Xavier
2	Andrew
3	Yolanda
4	Bob

```
SELECT * FROM Registrations FULL OUTER JOIN Logins
ON Registrations.name = Logins.name
WHERE Registrations.reg_id IS null OR
Logins.log_id IS null
```

REGISTRATIONS	
reg_id	name
1	Andrew
2	Bob
3	Charlie
4	David
null	null

RESULTS			
reg_id	name	log_id	name
3	Charlie	null	null
4	David	null	null
null	null	1	Xavier
null	null	3	Yolanda

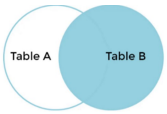
LOGINS	
log_id	name
1	Xavier
2	Andrew
3	Yolanda
4	Bob

7. Right Outer Join

- A RIGHT JOIN (or RIGHT OUTER JOIN) returns all rows from the right table and the matched rows from the left table. If there is no match, the result is NULL on the side of the left table
- A Right Join is same as Left Join, except the tsbles are switched.
- This would be same as switching the table order in LEFT OUTER JOIN.

```
SELECT columns
FROM table1
RIGHT JOIN table2
ON table1.common_column = table2.common_column;
```

```
SELECT * FROM TableA
RIGHT OUTER JOIN TableB
ON TableA.col_match = TableB.col_match
```



Example: Let's use the same employees and departments tables:

- Employees Table:

employee_id	first_name	last_name	department_id
1	John	Doe	101
2	Jane	Smith	102
3	Alice	Johnson	101
4	Bob	Brown	103
5	Charlie	White	104

- Departments Table:

department_id	department_name
101	IT
102	HR
103	Finance
105	Marketing

```
SELECT e.first_name, e.last_name, d.department_name
FROM employees AS e
RIGHT JOIN departments AS d
ON e.department_id = d.department_id;
```

- RIGHT JOIN: Returns all rows from the departments table (right table) and the matched rows from the employees table (left table).
- Matched Rows: Where there is a match on department_id, it returns the corresponding first_name and last_name from the employees table.
- Unmatched Rows: If there is no match, it returns NULL for columns from the employees table.

Joined Result:

first_name	last_name	department_name
John	Doe	IT
Alice	Johnson	IT

first_name	last_name	department_id
Charlie	White	104

- Charlie White: Has a department_id of 104, which does not match any department_id in the departments table, resulting in NULL for department_name.

Visual Representation

Imagine the tables as two circles in a Venn diagram:

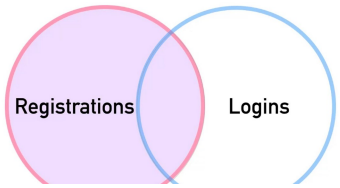
- Left Circle (employees): Represents all rows from the employees table.
- Right Circle (departments): Represents all rows from the departments table.
- Overlap: Represents rows where there is a match on department_id.

LEFT OUTER JOIN with WHERE d.department_name IS NULL Result:

- Non-Overlapping Left Area: Only those rows from the employees table that do not have a matching department_id in the departments table.

Example 2:

REGISTRATIONS	
reg_id	name
1	Andrew
2	Bob
3	Charlie
4	David



LOGINS	
log_id	name
1	Xavier
2	Andrew
3	Yolanda
4	Bob

```
SELECT * FROM Registrations
LEFT OUTER JOIN Logins
ON Registrations.name = Logins.name
```

REGISTRATIONS	
reg_id	name
1	Andrew
2	Bob
3	Charlie
4	David

RESULTS			
reg_id	name	log_id	name
1	Andrew	2	Andrew
2	Bob	4	Bob
3	Charlie	null	null
4	David	null	null

LOGINS	
log_id	name
1	Xavier
2	Andrew
3	Yolanda
4	Bob

```
SELECT * FROM Registrations
LEFT OUTER JOIN Logins
ON Registrations.name = Logins.name
WHERE Logins.log_id IS null
```

REGISTRATIONS	
reg_id	name
1	Andrew
2	Bob
3	Charlie
4	David

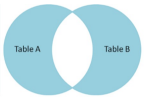
RESULTS			
reg_id	name	log_id	name
3	Charlie	null	null
4	David	null	null

LOGINS	
log_id	name
1	Xavier
2	Andrew
3	Yolanda
4	Bob

6. Left outer Join with WHERE condition

- To produce the set of records unique to Table A and Table B, we perform the same full outer join, then **exclude the records we don't want from both sides via a where clause**.

```
SELECT * FROM TableA
FULL OUTER JOIN TableB
ON TableA.name = TableB.name
WHERE TableA.id IS null
OR TableB.id IS null
```



Example:

employees table:

employee_id	first_name	last_name	department_id
1	John	Doe	101
2	Jane	Smith	102
3	Alice	Johnson	101
4	Bob	Brown	103
5	Charlie	White	104

departments table:

department_id	department_name
101	IT
102	HR
103	Finance
105	Marketing

```
SELECT e.first_name, e.last_name, e.department_id, d.department_name
FROM employees AS e
FULL OUTER JOIN departments AS d
ON e.department_id = d.department_id
WHERE e.department_id IS NULL OR d.department_id IS NULL;
```

Step1: Perform full outer join

Joined Result Before WHERE Condition:

first_name	last_name	department_id	department_name
John	Doe	101	IT
Alice	Johnson	101	IT
Jane	Smith	102	HR
Bob	Brown	103	Finance
Charlie	White	104	NULL
NULL	NULL	105	Marketing

Step 2: Apply where clause

Result After WHERE Condition:

first_name	last_name	department_id	department_name
Charlie	White	104	NULL
NULL	NULL	NULL	Marketing

Joined Result:

first_name	last_name	department_name
John	Doe	IT
Alice	Johnson	IT
Jane	Smith	HR
Bob	Brown	Finance
NULL	NULL	Marketing

- Imagine the tables as two circles in a Venn diagram:
  - Left Circle (employees): Represents all rows from the employees table.
  - Right Circle (departments): Represents all rows from the departments table.
  - Overlap: Represents rows where there is a match on department_id.
- RIGHT JOIN Result:
  - Entire Right Circle: All rows from departments are included.
  - Overlap: Matching rows have corresponding values from employees.
  - Non-Overlapping Right Area: Unmatched rows from departments have NULL in columns from employees.

8. UNION :

- The UNION operator in SQL is used to combine the result sets of two or more SELECT queries.
- The UNION operator selects only distinct values by default, which means duplicate records are removed.
- If you want to include duplicates, you can use the UNION ALL operator.
- Concatenate two results together

```
SELECT column1, column2, ...
FROM table1
UNION
SELECT column1, column2, ...
FROM table2;
```

Example: We have 2 tables employees in US and employees in UK

employees_in_us table:

employee_id	first_name	last_name	country
1	John	Doe	US
2	Jane	Smith	US
3	Alice	Johnson	US
4	Jane	Smith	US

employees_in_uk table:

employee_id	first_name	last_name	country
5	Bob	Brown	UK
6	Charlie	White	UK
7	Jane	Smith	UK
8	John	Doe	US

```
SELECT first_name, last_name, country
FROM employees_in_us
UNION
SELECT first_name, last_name, country
FROM employees_in_uk;
```

result with union

first_name	last_name	country
John	Doe	US
Jane	Smith	US
Alice	Johnson	US
Bob	Brown	UK
Charlie	White	UK
Jane	Smith	UK

- UNION removes duplicate rows, even if they appear multiple times in the same table or in both tables.
- Only distinct combinations of first_name, last_name, and country are included in the final result set.

UNION ALL

- If you want to include all records, including duplicates:

```
SELECT first_name, last_name, country
FROM employees_in_us
UNION ALL
SELECT first_name, last_name, country
FROM employees_in_uk;
```

The result set will include all rows, including duplicates:

first_name	last_name	country
John	Doe	US
Jane	Smith	US
Alice	Johnson	US
Jane	Smith	US
Bob	Brown	UK
Charlie	White	UK
Jane	Smith	UK
John	Doe	US

- UNION ALL includes all rows from both tables, without removing duplicates.
- Every row from both employees_in_us and employees_in_uk is included in the final result set, even if they are exact duplicates or have some differences.

9. Self Join

- A self-join is a query in which a table is joined to itself.
- Self-joins are useful for comparing values in a column of rows within the same table.
- The self join can be viewed as a join of two copies of the same table.
- The table is not actually copied, but SQL performs the command as though it were.
- There is no special keyword for a self join, its simply standard JOIN syntax with the same table in both parts.

- However, when using a self join it is necessary to use an alias for the table, otherwise the table names would be ambiguous.

Syntax :

```
SELECT tableA.col, tableB.col
FROM table AS tableA
JOIN table AS tableB ON
tableA.some_col = tableB.other_col
```

Example :
We want result showing employee name and their reports reciepient name

EMPLOYEES		
emp_id	name	report_id
1	Andrew	3
2	Bob	3
3	Charlie	4
4	David	1

name	rep
Andrew	Charlie
Bob	Charlie
Charlie	David
David	Andrew

```
SELECT emp.name, report.name AS rep
FROM employees AS emp
JOIN employees AS report ON
emp.emp_id = report.report_id
```

name	rep
Andrew	Charlie
Bob	Charlie
Charlie	David
David	Andrew

Example 2 :

empId	name	dept	salary	experience	managerId
1	Clark	Sales	80000	2	NULL
2	Dave	Accounting	10000	2	1
3	Ava	Sales	15000	1	1
4	Eva	Sales	90000	5	NULL
5	John	Sales	20000	3	4

we want to find the names of employees along with the names of their managers. We can achieve this with a self join:

```
SELECT e1.name AS employee_name, e2.name AS manager_name
FROM EMPLOYEE1 e1
LEFT JOIN EMPLOYEE1 e2 ON e1.managerId = e2.empId;
```

- EMPLOYEE1 e1 is the alias for the employee table.
- EMPLOYEE1 e2 is another alias for the same employee table, but it is treated as a different table.
- We perform a LEFT JOIN where e1.managerId matches e2.empId. This associates each employee with their manager.

Output :

employee_name	manager_name
Clark	NULL
Dave	Clark
Ava	Clark
Eva	NULL
John	Eva

8. Advance SQL commands (Date and Time)

26 May 2024 21:51

Timestamps and Extracts

Postgres SQL can hold Data and time information
- **TIME** - Contains only time
- **DATE** - Contains only date
- **TIMESTAMP** - Contains date and time
- **TIMESTAMPTZ** - Contains date,time, and timezone

Functions and Operations related to specific datatype
- TIMEZONE
- NOW
- TIMEOFDAY
- CURRENT_TIME
- CURRENT_DATE

1. Display all functions

```
SHOW ALL
```

Data Output | Messages | Notifications

	name text	setting text	description text
1	allow_in_place_tablespaces	off	Allows tablespaces directly inside pg_tblspc, for testing.
2	allow_system_table_mods	off	Allows modifications of the structure of system tables.
3	application_name	pgAdmin 4 - CONN:1421874	Sets the application name to be reported in statistics and logs.
4	archive_cleanup_command		Sets the shell command that will be executed at every restart point.
5	archive_command	(disabled)	Sets the shell command that will be called to archive a WAL file.
6	archive_library		Sets the library that will be called to archive a WAL file.
7	archive_mode	off	Allows archiving of WAL files using archive_command.
8	archive_timeout	0	Sets the amount of time to wait before forcing a switch to the next WAL file.
9	array_nulls	on	Enable input of NULL elements in arrays.
10	authentication_timeout	1min	Sets the maximum allowed time to complete client authentication.
11	autovacuum	on	Starts the autovacuum subprocess.
12	autovacuum_analyze_scale_factor	0.1	Number of tuple inserts, updates, or deletes prior to analyze as a fraction of reltuples.

2. Get current timezone

```
SHOW TIMEZONE
```

Data Output | Messages | Notifications

	TimeZone text
1	Asia/Calcutta

3. Get timestamp with timezone

```
SELECT NOW()
```

Data Output | Messages | Notifications

	now timestamp with time zone
1	2024-05-26 22:03:56.479726+05:30

4. Get data and time in string format

```
SELECT TIMEOFDAY()
```

Data Output | Messages | Notifications

	timeofday text
1	Sun May 26 22:04:47.701928 2024 IST

5. Current time

```
SELECT CURRENT_TIME
```

Data Output | Messages | Notifications

	current_time time with time zone
1	22:05:25.298713+05:30

6. Current data

```
SELECT CURRENT_DATE
```

Data Output | Messages | Notifications

	current_date date
1	2024-05-26

Extracting Time and Date base Information

- EXTRACT()
- AGE()
- TO_CHAR()

1. EXTRACT()
   - Allows you to “extract” or obtain a sub-component of a date value
     - YEAR
     - MONTH
     - DAY
     - WEEK
     - QUARTER

```
SELECT CURRENT_TIMESTAMP
```

Data Output | Messages | Notifications

	current_timestamp timestamp with time zone
1	2024-06-09 12:03:49.057202+05:30

```
SELECT CURRENT_TIMESTAMP as timestamp
```

Data Output | Messages | Notifications

	timestamp timestamp with time zone
1	2024-06-09 12:04:52.827281+05:30

```
SELECT EXTRACT(YEAR FROM CURRENT_TIMESTAMP) AS current_year;
```

Data Output | Messages | Notifications

	current_year numeric
1	2024

```
SELECT EXTRACT(MONTH FROM CURRENT_TIMESTAMP) AS current_year;
```

Data Output | Messages | Notifications

	current_year numeric
1	6

```
SELECT EXTRACT(DAY FROM CURRENT_TIMESTAMP) AS current_year;
```

Data Output | Messages | Notifications

	current_year numeric
1	9

```
SELECT AGE(CURRENT_TIMESTAMP) AS currentAge;
```

Data Output | Messages | Notifications

	currentage interval
1	-12:35:57.083033

```
SELECT TO_CHAR(CURRENT_TIMESTAMP, 'YYYY-MM-DD') as currentDate;
```

Data Output | Messages | Notifications

	currentdate text

```
1  SELECT TO_CHAR(CURRENT_TIMESTAMP, 'YYYY-MM-DD') as currentDate;
```

Data Output | Messages | Notifications

	currentdate text
1	2024-06-09

TO_CHAR :

```
to_char(value, format)
```

- value: The data you want to format (e.g., a date, timestamp, or number).
- format: The format pattern you want to apply to the value.

Formatting Dates and Timestamps

Here are some common format patterns for dates and timestamps:

- YYYY: 4-digit year
- MM: 2-digit month
- DD: 2-digit day
- HH24: 2-digit hour in 24-hour format
- MI: 2-digit minute
- SS: 2-digit second

Custome Date format

```
1  SELECT TO_CHAR(CURRENT_DATE, 'Month DD, YYYY') AS custom_formatted_date;
```

Data Output | Messages | Notifications

	custom_formatted_date text
1	June 09, 2024

Function Description Example(s)
to_char (timestamp, text) → text to_char (timestamp with time zone, text) → text Converts time stamp to string according to the given format. to_char(timestamp '2002-04-20 17:31:12.66', 'HH12:MI:SS') → 05:31:12
to_char (interval, text) → text Converts interval to string according to the given format. to_char(interval '15h 2m 12s', 'HH24:MI:SS') → 15:02:12
to_char (*numeric_type*, text) → text Converts number to string according to the given format; available for integer, bigint, numeric, real, double precision. to_char(125, '999') → 125 to_char(125.8::real, '999D9') → 125.8 to_char(-125.8, '999D99S') → 125.80-
to_date (text, text) → date Converts string to date according to the given format. to_date('05 Dec 2000', 'DD Mon YYYY') → 2000-12-05
to_number (text, text) → numeric Converts string to numeric according to the given format. to_number('12,454.8-', '99G999D9S') → -12454.8
to_timestamp (text, text) → timestamp with time zone Converts string to time stamp according to the given format. (See also to_timestamp(double precision) in Table 9.33.) to_timestamp('05 Dec 2000', 'DD Mon YYYY') → 2000-12-05 00:00:00-05

Template Patterns for Date/Time Formatting

https://www.postgresql.org/docs/current/functions-formatting.html

Pattern	Description
HH	hour of day (01–12)
HH12	hour of day (01–12)
HH24	hour of day (00–23)
MI	minute (00–59)
SS	second (00–59)
MS	millisecond (000–999)
US	microsecond (000000–999999)
FF1	tenth of second (0–9)
FF2	hundredth of second (00–99)
FF3	millisecond (000–999)
FF4	tenth of a millisecond (0000–9999)
FF5	hundredth of a millisecond (00000–99999)
FF6	microsecond (000000–999999)
SSSS, SSSSS	seconds past midnight (0–86399)
AM, am, PM or pm	meridiem indicator (without periods)

A.M., a.m., P.M. or p.m.	meridiem indicator (with periods)
Y,YYY	year (4 or more digits) with comma
YYYY	year (4 or more digits)
YYY	last 3 digits of year
YY	last 2 digits of year
Y	last digit of year
IYYY	ISO 8601 week-numbering year (4 or more digits)
IYY	last 3 digits of ISO 8601 week-numbering year
IY	last 2 digits of ISO 8601 week-numbering year
I	last digit of ISO 8601 week-numbering year
BC, bc, AD or ad	era indicator (without periods)
B.C., b.c., A.D. or a.d.	era indicator (with periods)
MONTH	full upper case month name (blank-padded to 9 chars)
Month	full capitalized month name (blank-padded to 9 chars)
month	full lower case month name (blank-padded to 9 chars)
MON	abbreviated upper case month name (3 chars in English, localized lengths vary)
Mon	abbreviated capitalized month name (3 chars in English, localized lengths vary)
mon	abbreviated lower case month name (3 chars in English, localized lengths vary)
MM	month number (01–12)
DAY	full upper case day name (blank-padded to 9 chars)
Day	full capitalized day name (blank-padded to 9 chars)
day	full lower case day name (blank-padded to 9 chars)
DY	abbreviated upper case day name (3 chars in English, localized lengths vary)
Dy	abbreviated capitalized day name (3 chars in English, localized lengths vary)
dy	abbreviated lower case day name (3 chars in English, localized lengths vary)
DDD	day of year (001–366)
IDDD	day of ISO 8601 week-numbering year (001–371; day 1 of the year is Monday of the first ISO week)
DD	day of month (01–31)
D	day of the week, Sunday (1) to Saturday (7)
ID	ISO 8601 day of the week, Monday (1) to Sunday (7)
W	week of month (1–5) (the first week starts on the first day of the month)
WW	week number of year (1–53) (the first week starts on the first day of the year)
IW	week number of ISO 8601 week-numbering year (01–53; the first Thursday of the year is in week 1)
CC	century (2 digits) (the twenty-first century starts on 2001-01-01)
J	Julian Date (integer days since November 24, 4714 BC at local midnight; see Section B.7)
Q	quarter
RM	month in upper case Roman numerals (I–XII; I=January)
rm	month in lower case Roman numerals (i–xii; i=January)
TZ	upper case time-zone abbreviation (only supported in to_char)
tz	lower case time-zone abbreviation (only supported in to_char)
TZH	time-zone hours
TZM	time-zone minutes
OF	time-zone offset from UTC (only supported in to_char)

```
SELECT
CURRENT_DATE AS current_date,
to_char(CURRENT_DATE, 'Day') AS day_of_week_name,
EXTRACT(DOW FROM CURRENT_DATE) AS day_of_week_number;

```

Data Output | Messages | Notifications

	current_date date	day_of_week_name text	day_of_week_number numeric
1	2024-06-09	Sunday	0

9. Advance SQL (Mathematrical functions and operators)

09 June 2024 14:43

https://www.postgresql.org/docs/current/functions-math.html

Operator Description Example(s)
numeric_type + ***numeric_type*** → ***numeric_type*** Addition 2 + 3 → 5
+ ***numeric_type*** → ***numeric_type*** Unary plus (no operation) + 3.5 → 3.5
numeric_type - ***numeric_type*** → ***numeric_type*** Subtraction 2 - 3 → -1
- ***numeric_type*** → ***numeric_type*** Negation - (-4) → 4
numeric_type * ***numeric_type*** → ***numeric_type*** Multiplication 2 * 3 → 6
numeric_type / ***numeric_type*** → ***numeric_type*** Division (for integral types, division truncates the result towards zero) 5.0 / 2 → 2.5000000000000000 5 / 2 → 2 (-5) / 2 → -2
numeric_type % ***numeric_type*** → ***numeric_type*** Modulo (remainder); available for smallint, integer, bigint, and numeric 5 % 4 → 1
numeric ^ numeric → numeric double precision ^ double precision → double precision Exponentiation 2 ^ 3 → 8 Unlike typical mathematical practice, multiple uses of ^ will associate left to right by default: 2 ^ 3 ^ 3 → 512 2 ^ (3 ^ 3) → 134217728
|/ double precision → double precision Square root |/ 25.0 → 5
||/ double precision → double precision Cube root

|/ 64.0 → 4
@ ***numeric_type*** → ***numeric_type*** Absolute value @ -5.0 → 5.0
integral_type & ***integral_type*** → ***integral_type*** Bitwise AND 91 & 15 → 11
integral_type | ***integral_type*** → ***integral_type*** Bitwise OR 32 | 3 → 35
integral_type # ***integral_type*** → ***integral_type*** Bitwise exclusive OR 17 # 5 → 20
~ ***integral_type*** → ***integral_type*** Bitwise NOT ~1 → -2
integral_type << integer → ***integral_type*** Bitwise shift left 1 << 4 → 16
integral_type >> integer → ***integral_type*** Bitwise shift right 8 >> 2 → 2

From <https://www.postgresql.org/docs/current/functions-math.html>

```
1  SELECT ROUND(salary/(experience*13000),2) AS rating FROM EMPLOYEE1 ;
2
```

Data Output | Messages | Notifications

	rating numeric
1	3.00
2	0.00
3	1.00

10. Advance SQL (String Functions and Operators)

09 June 2024 14:55

SQL String Functions and Operators

From <https://www.postgresql.org/docs/current/functions-string.html>

Function/Operator Description Example(s)
text || text → text Concatenates the two strings. 'Post' || 'greSQL' → PostgreSQL
text || anynonarray → text anynonarray || text → text Converts the non-string input to text, then concatenates the two strings. (The non-string input cannot be of an array type, because that would create ambiguity with the array || operators. If you want to concatenate an array's text equivalent, cast it to text explicitly.) 'Value: ' || 42 → Value: 42
btrim (*string* text [, *characters* text]) → text Removes the longest string containing only characters in *characters* (a space by default) from the start and end of *string*. btrim('xyxtrimyyx', 'xyz') → trim
text IS [NOT] [*form*] NORMALIZED → boolean Checks whether the string is in the specified Unicode normalization form. The optional *form* key word specifies the form: NFC (the default), NFD, NFKC, or NFKD. This expression can only be used when the server encoding is UTF8. Note that checking for normalization using this expression is often faster than normalizing possibly already normalized strings. U&'\0061\0308bc' IS NFD NORMALIZED → t
bit_length (text) → integer Returns number of bits in the string (8 times the octet_length). bit_length('jose') → 32
char_length (text) → integer character_length (text) → integer Returns number of characters in the string. char_length('josé') → 4
lower (text) → text Converts the string to all lower case, according to the rules of the database's locale. lower('TOM') → tom
lpad (*string* text, *length* integer [, *fill* text]) → text Extends the *string* to length *length* by prepending the characters *fill* (a space by default). If the *string* is already longer than *length* then it is truncated (on the right). lpad('hi', 5, 'xy') → xyxhi
ltrim (*string* text [, *characters* text]) → text Removes the longest string containing only characters in *characters* (a space by default) from the start of *string*. ltrim('zzzytest', 'xyz') → test
normalize (text [, *form*]) → text Converts the string to the specified Unicode normalization form. The optional *form* key word specifies the form: NFC (the default), NFD, NFKC, or NFKD. This function can only be used when the server encoding is UTF8. normalize(U&'\0061\0308bc', NFC) → U&'\00E4bc'
octet_length (text) → integer Returns number of bytes in the string. octet_length('josé') → 5 (if server encoding is UTF8)
octet_length (character) → integer Returns number of bytes in the string. Since this version of the function accepts type character directly, it will not strip trailing spaces. octet_length('abc '::character(4)) → 4
overlay (*string* text PLACING *newsubstring* text FROM *start* integer [FOR *count* integer]) → text Replaces the substring of *string* that starts at the *start*'th character and extends for *count* characters with *newsubstring*. If *count* is omitted, it defaults to the length of *newsubstring*. overlay('Txxxxas' placing 'hom' from 2 for 4) → Thomas
position (*substring* text IN *string* text) → integer Returns first starting index of the specified *substring* within *string*, or zero if it's not present. position('om' in 'Thomas') → 3
rpad (*string* text, *length* integer [, *fill* text]) → text Extends the *string* to length *length* by appending the characters *fill* (a space by default). If the *string* is already longer than *length* then it is truncated. rpad('hi', 5, 'xy') → hixyx
rtrim (*string* text [, *characters* text]) → text Removes the longest string containing only characters in *characters* (a space by default) from the end of *string*. rtrim('testxxzx', 'xyz') → test
substring (*string* text [FROM *start* integer] [FOR *count* integer]) → text Extracts the substring of *string* starting at the *start*'th character if that is specified, and stopping after *count* characters if that is specified. Provide at least one of *start* and *count*. substring('Thomas' from 2 for 3) → hom substring('Thomas' from 3) → omas substring('Thomas' for 2) → Th
substring (*string* text FROM *pattern* text) → text Extracts the first substring matching POSIX regular expression; see Section 9.7.3. substring('Thomas' from '...$') → mas
substring (*string* text SIMILAR *pattern* text ESCAPE *escape* text) → text substring (*string* text FROM *pattern* text FOR *escape* text) → text Extracts the first substring matching SQL regular expression; see Section 9.7.2. The first form has been specified since SQL:2003; the second form was only in SQL:1999 and should be considered obsolete. substring('Thomas' similar '%#"o_a#"_' escape '#') → oma
trim ([LEADING | TRAILING | BOTH] [*characters* text] FROM *string* text) → text Removes the longest string containing only characters in *characters* (a space by default) from the start, end, or both ends (BOTH is the default) of *string*. trim(both 'xyz' from 'yxTomxx') → Tom
trim ([LEADING | TRAILING | BOTH] [FROM] *string* text [, *characters* text]) → text This is a non-standard syntax for trim(). trim(both from 'yxTomxx', 'xyz') → Tom
upper (text) → text Converts the string to all upper case, according to the rules of the database's locale. upper('tom') → TOM

Query Editor Query History

```
1  SELECT upper(first_name) || ' ' || upper(last_name) AS full_name
2  FROM customer
```

Data Output Explain Messages Notifications

	full_name text
1	JARED ELY
2	MARY SMITH

Query Editor Query History

```
1  SELECT LOWER(LEFT(first_name,1)) || LOWER(last_name) || '@gmail.com'
2  AS custom_email
3  FROM customer
```

Data Output Explain Messages Notifications

	custom_email text
1	jely@gmail.com
2	msmith@gmail.com
3	pjohnson@gmail.c...

Table 9.10. Other String Functions and Operators

Function/Operator Description Example(s)
text ^@ text → boolean Returns true if the first string starts with the second string (equivalent to the starts_with() function). 'alphabet' ^@ 'alph' → t
ascii (text) → integer Returns the numeric code of the first character of the argument. In UTF8 encoding, returns the Unicode code point of the character. In other multibyte encodings, the argument must be an ASCII character. ascii('x') → 120
chr (integer) → text Returns the character with the given code. In UTF8 encoding the argument is treated as a Unicode code point. In other multibyte encodings the argument must designate an ASCII character. chr(0) is disallowed because text data types cannot store that character. chr(65) → A
concat (*val1* "any" [, *val2* "any" [, ...]]) → text Concatenates the text representations of all the arguments. NULL arguments are ignored. concat('abcde', 2, NULL, 22) → abcde222
concat_ws (*sep* text, *val1* "any" [, *val2* "any" [, ...]]) → text Concatenates all but the first argument, with separators. The first argument is used as the separator string, and should not be NULL. Other NULL arguments are ignored. concat_ws(',', 'abcde', 2, NULL, 22) → abcde,2,22
format (*formatstr* text [, *formatarg* "any" [, ...]]) → text Formats arguments according to a format string; see Section 9.4.1. This function is similar to the C function sprintf. format('Hello %s, %1$s', 'World') → Hello World, World
initcap (text) → text Converts the first letter of each word to upper case and the rest to lower case. Words are sequences of alphanumeric characters separated by non-alphanumeric characters. initcap('hi THOMAS') → Hi Thomas
left (*string* text, *n* integer) → text Returns first *n* characters in the string, or when *n* is negative, returns all but last |*n*| characters. left('abcde', 2) → ab
length (text) → integer Returns the number of characters in the string. length('jose') → 4
md5 (text) → text Computes the MD5 hash of the argument, with the result written in hexadecimal. md5('abc') → 900150983cd24fb0d6963f7d28e17f72
parse_ident (*qualified_identifier* text [, *strict_mode* boolean DEFAULT true]) → text[] Splits *qualified_identifier* into an array of identifiers, removing any quoting of individual identifiers. By default, extra characters after the last identifier are considered an error; but if the second parameter is false, then such extra characters are ignored. (This behavior is useful for parsing names for objects like functions.) Note that this function does not truncate over-length identifiers. If you want truncation you can cast the result to name[]. parse_ident('"SomeSchema".someTable') → {SomeSchema,sometable}
pg_client_encoding ( ) → name Returns current client encoding name. pg_client_encoding() → UTF8
quote_ident (text) → text Returns the given string suitably quoted to be used as an identifier in an SQL statement string. Quotes are added only if necessary (i.e., if the string contains non-identifier characters or would be case-folded). Embedded quotes are properly doubled. See also Example 43.1. quote_ident('Foo bar') → "Foo bar"
quote_literal (text) → text Returns the given string suitably quoted to be used as a string literal in an SQL statement string. Embedded single-quotes and backslashes are properly doubled. Note that quote_literal returns null on null input; if the argument might be null, quote_nullable is often more suitable. See also Example 43.1. quote_literal(E'O\'Reilly') → 'O''Reilly'
quote_literal (anyelement) → text Converts the given value to text and then quotes it as a literal. Embedded single-quotes and backslashes are properly doubled. quote_literal(42.5) → '42.5'
quote_nullable (text) → text Returns the given string suitably quoted to be used as a string literal in an SQL statement string; or, if the argument is null, returns NULL. Embedded single-quotes and backslashes are properly doubled. See also Example 43.1. quote_nullable(NULL) → NULL
quote_nullable (anyelement) → text Converts the given value to text and then quotes it as a literal; or, if the argument is null, returns NULL. Embedded single-quotes and backslashes are properly doubled. quote_nullable(42.5) → '42.5'
regexp_count (*string* text, *pattern* text [, *start* integer [, *flags* text]]) → integer Returns the number of times the POSIX regular expression *pattern* matches in the *string*; see Section 9.7.3. regexp_count('123456789012', '\d\d\d', 2) → 3
regexp_instr (*string* text, *pattern* text [, *start* integer [, *N* integer [, *endoption* integer [, *flags* text [, *subexpr* integer]]]]]) → integer Returns the position within *string* where the *N*'th match of the POSIX regular expression *pattern* occurs, or zero if there is no such match; see Section 9.7.3. regexp_instr('ABCDEF', 'c(.)(..)', 1, 1, 0, 'i') → 3 regexp_instr('ABCDEF', 'c(.)(..)', 1, 1, 0, 'i', 2) → 5
regexp_like (*string* text, *pattern* text [, *flags* text]) → boolean Checks whether a match of the POSIX regular expression *pattern* occurs within *string*; see Section 9.7.3. regexp_like('Hello World', 'world$', 'i') → t
regexp_match (*string* text, *pattern* text [, *flags* text]) → text[] Returns substrings within the first match of the POSIX regular expression *pattern* to the *string*; see Section 9.7.3. regexp_match('foobarbequebaz', '(bar)(beque)') → {bar,beque}
regexp_matches (*string* text, *pattern* text [, *flags* text]) → setof text[] Returns substrings within the first match of the POSIX regular expression *pattern* to the *string*, or substrings within all such matches if the g flag is used; see Section 9.7.3. regexp_matches('foobarbequebaz', 'ba.', 'g') → {bar} {baz}
regexp_replace (*string* text, *pattern* text, *replacement* text [, *start* integer] [, *flags* text]) → text Replaces the substring that is the first match to the POSIX regular expression *pattern*, or all such matches if the g flag is used; see Section 9.7.3. regexp_replace('Thomas', '.[mN]a.', 'M') → ThM

regexp_replace (*string* text, *pattern* text, *replacement* text, *start* integer, *N* integer [, *flags* text]) → text
Replaces the substring that is the *N*'th match to the POSIX regular expression *pattern*, or all such matches if *N* is zero; see Section 9.7.3.
regexp_replace('Thomas', '.', 'X', 3, 2) → ThoXas

regexp_split_to_array (*string* text, *pattern* text [, *flags* text]) → text[]
Splits *string* using a POSIX regular expression as the delimiter, producing an array of results; see Section 9.7.3.
regexp_split_to_array('hello world', '\s+') → {hello,world}

regexp_split_to_table (*string* text, *pattern* text [, *flags* text]) → setof text
Splits *string* using a POSIX regular expression as the delimiter, producing a set of results; see Section 9.7.3.
regexp_split_to_table('hello world', '\s+') →
hello
world

regexp_substr (*string* text, *pattern* text [, *start* integer [, *N* integer [, *flags* text [, *subexpr* integer]]]]) → text
Returns the substring within *string* that matches the *N*'th occurrence of the POSIX regular expression *pattern*, or NULL if there is no such match; see Section 9.7.3.
regexp_substr('ABCDEF', 'c(.)(..)', 1, 1, 'i') → CDEF
regexp_substr('ABCDEF', 'c(.)(..)', 1, 1, 'i', 2) → EF

repeat (*string* text, *number* integer) → text
Repeats *string* the specified *number* of times.
repeat('Pg', 4) → PgPgPgPg

replace (*string* text, *from* text, *to* text) → text
Replaces all occurrences in *string* of substring *from* with substring *to*.
replace('abcdefabcdef', 'cd', 'XX') → abXXefabXXef

reverse (text) → text
Reverses the order of the characters in the string.
reverse('abcde') → edcba

right (*string* text, *n* integer) → text
Returns last *n* characters in the string, or when *n* is negative, returns all but first |*n*| characters.
right('abcde', 2) → de

split_part (*string* text, *delimiter* text, *n* integer) → text
Splits *string* at occurrences of *delimiter* and returns the *n*'th field (counting from one), or when *n* is negative, returns the |*n*|'th-from-last field.
split_part('abc~@~def~@~ghi', '~@~', 2) → def
split_part('abc,def,ghi,jkl', ',', -2) → ghi

starts_with (*string* text, *prefix* text) → boolean
Returns true if *string* starts with *prefix*.
starts_with('alphabet', 'alph') → t

string_to_array (*string* text, *delimiter* text [, *null_string* text]) → text[]
Splits the *string* at occurrences of *delimiter* and forms the resulting fields into a text array. If *delimiter* is NULL, each character in the *string* will become a separate element in the array. If *delimiter* is an empty string, then the *string* is treated as a single field. If *null_string* is supplied and is not NULL, fields matching that string are replaced by NULL. See also array_to_string.
string_to_array('xx~~yy~~zz', '~~', 'yy') → {xx,NULL,zz}

string_to_table (*string* text, *delimiter* text [, *null_string* text]) → setof text
Splits the *string* at occurrences of *delimiter* and returns the resulting fields as a set of text rows. If *delimiter* is NULL, each character in the *string* will become a separate row of the result. If *delimiter* is an empty string, then the *string* is treated as a single field. If *null_string* is supplied and is not NULL, fields matching that string are replaced by NULL.
string_to_table('xx~^~yy~^~zz', '~^~', 'yy') →
xx
NULL
zz

strpos (*string* text, *substring* text) → integer
Returns first starting index of the specified *substring* within *string*, or zero if it's not present. (Same as position(*substring* in *string*), but note the reversed argument order.)
strpos('high', 'ig') → 2

substr (*string* text, *start* integer [, *count* integer]) → text
Extracts the substring of *string* starting at the *start*'th character, and extending for *count* characters if that is specified. (Same as substring(*string* from *start* for *count*).)
substr('alphabet', 3) → phabet
substr('alphabet', 3, 2) → ph

to_ascii (*string* text) → text
to_ascii (*string* text, *encoding* name) → text
to_ascii (*string* text, *encoding* integer) → text
Converts *string* to ASCII from another encoding, which may be identified by name or number. If *encoding* is omitted the database encoding is assumed (which in practice is the only useful case). The conversion consists primarily of dropping accents. Conversion is only supported from LATIN1, LATIN2, LATIN9, and WIN1250 encodings. (See the unaccent module for another, more flexible solution.)
to_ascii('Karél') → Karel

to_hex (integer) → text
to_hex (bigint) → text
Converts the number to its equivalent hexadecimal representation.
to_hex(2147483647) → 7fffffff

translate (*string* text, *from* text, *to* text) → text
Replaces each character in *string* that matches a character in the *from* set with the corresponding character in the *to* set. If *from* is longer than *to*, occurrences of the extra characters in *from* are deleted.
translate('12345', '143', 'ax') → a2x5

unistr (text) → text
Evaluate escaped Unicode characters in the argument. Unicode characters can be specified as ***\XXXX*** (4 hexadecimal digits), ***\+XXXXXX*** (6 hexadecimal digits), ***\uXXXX*** (4 hexadecimal digits), or ***\UXXXXXXXX*** (8 hexadecimal digits). To specify a backslash, write two backslashes. All other characters are taken literally.
If the server encoding is not UTF-8, the Unicode code point identified by one of these escape sequences is converted to the actual server encoding; an error is reported if that's not possible.
This function provides a (non-standard) alternative to string constants with Unicode escapes (see Section 4.1.2.3).
unistr('d\0061t\+000061') → data
unistr('d\u0061t\U00000061') → data

11. SubQuery

09 June 2024 15:01

- A subquery, also known as an inner query or nested query, is a query within another SQL query.
- The subquery can be used in various parts of an SQL statement, such as the SELECT list, FROM clause, and WHERE clause.
- Subqueries are powerful tools for filtering, transforming, and combining data in sophisticated ways.

Types of Subqueries

1. **Single-row subquery**
2. **Multiple-row subquery**
3. **Multiple-column subquery**
4. **Correlated subquery**
5. **Nested**

1. **Single-row Subquery**
   A single-row subquery returns a single row and is typically used with comparison operators such as =, <, >, <=, >=, or <>.

```
SELECT name
FROM EMPLOYEE1
WHERE salary > (SELECT AVG(salary) FROM EMPLOYEE1);
```

This query finds all employees whose salary is greater than the average salary of all employees.

2. **Multiple-row Subquery**
   A multiple-row subquery returns more than one row and is used with operators like IN, ANY, or ALL.

```
SELECT name
FROM EMPLOYEE1
WHERE dept IN (SELECT dept FROM EMPLOYEE1 WHERE experience > 1);
```

This query selects names of employees who work in departments where at least one employee has more than 1 year of experience

3. **Multiple-column Subquery**
   A multiple-column subquery returns more than one column and can be used in comparisons involving multiple columns.

```
SELECT name
FROM EMPLOYEE1
WHERE (salary, experience) IN (SELECT salary, experience FROM EMPLOYEE1 WHERE dept = 'Sales');
```

This query finds names of employees who have the same salary and experience as those working in the Sales department.

4. **Correlated Subquery**
   A correlated subquery refers to a column in the outer query. It is executed once for each row selected by the outer query.

```
SELECT e1.name
FROM EMPLOYEE1 e1
WHERE salary > (SELECT AVG(salary) FROM EMPLOYEE1 e2 WHERE e2.dept = e1.dept);
```

This query finds employees whose salary is greater than the average salary of their respective departments.

5. **Nested Subquery**
   A nested subquery is simply a subquery within another subquery. This can be useful for more complex filtering.

```
SELECT name
FROM EMPLOYEE1
WHERE salary > (SELECT AVG(salary)
                FROM EMPLOYEE1
                WHERE dept = (SELECT dept
                              FROM EMPLOYEE1
                              WHERE name = 'Clark'));
```

This query finds names of employees whose salary is greater than the average salary of the department where 'Clark' works.

Detailed Examples

Example 1: Using a Subquery in the SELECT Clause

```
SELECT name,
       (SELECT AVG(salary) FROM EMPLOYEE1) AS avg_salary
FROM EMPLOYEE1;
```

This query lists each employee's name along with the average salary of all employees.

Example 2: Using a Subquery in the FROM Clause

```
SELECT dept, avg_salary
FROM (SELECT dept, AVG(salary) AS avg_salary
      FROM EMPLOYEE1
      GROUP BY dept) AS dept_avg_salary;
```

This query calculates the average salary for each department and then selects the department and its average salary

Example 3: Using a Subquery with EXISTS

```
SELECT name
FROM EMPLOYEE1 e1
WHERE EXISTS (SELECT 1
              FROM EMPLOYEE1 e2
              WHERE e2.dept = 'Sales' AND e2.empId = e1.empId);
```

This query selects names of employees who work in the Sales department by checking the existence of a corresponding row in a subquery.

Example 4: Using a Subquery with NOT EXISTS

```
SELECT name
FROM EMPLOYEE1 e1
WHERE NOT EXISTS (SELECT 1
                  FROM EMPLOYEE1 e2
                  WHERE e2.dept = 'Sales' AND e2.empId = e1.empId);
```

This query selects names of employees who do not work in the Sales department.

12. SQL Constraints

09 June 2024 15:54

1. References

REFERENCES constraint is used to establish a relationship between two tables, typically to enforce referential integrity. This constraint ensures that the values in a column (or columns) of one table correspond to the values in another table's column (usually its primary key)

```
CREATE TABLE table_name (
    column1 datatype,
    column2 datatype,
    ...
    CONSTRAINT constraint_name
    FOREIGN KEY (column_name)
    REFERENCES parent_table (parent_column)
    [ON DELETE action]
    [ON UPDATE action]
);
```

Example : Suppose we have two tables: employees and departments. The employees table contains information about employees, while the departments table contains information about departments. Each employee is associated with a department.

```
CREATE TABLE departments (
    dept_id SERIAL PRIMARY KEY,
    name VARCHAR(50) NOT NULL
);

CREATE TABLE employees (
    emp_id SERIAL PRIMARY KEY,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    dept_id INTEGER,
    FOREIGN KEY (dept_id) REFERENCES departments (dept_id)
);
```

12 June 2024 00:51