Database Management Systems.

Lecture 3

Content:

- SQL DML
- SQL DQL
- Functions
- Filtering Data
- Conditional Expressions & Operators

SQL DML

- The SQL commands that deals with the manipulation of data present in the database belong to **DML** or **Data Manipulation Language** and this includes most of the SQL statements.
- It is the component of the SQL statement that controls access to data and to the database.

- **SQL DML commands:**
- **INSERT** - is used to insert data into a table.
- **UPDATE** - is used to update existing data within a table.
- **DELETE** - is used to delete records from a database table.

INSERT STATEMENT

- The **INSERT** statement of SQL is used to insert a new row in a table. There are two ways of using **INSERT** statement for inserting rows:

- **Only values:** First method is to specify only the value of data to be inserted without the column names
- `INSERT INTO table_name VALUES (value1, value2, value3,…);`

name of the table

values of first column, second column, etc…

- **Column names and values both:** In the second method we will specify both the columns which we want to fill and their corresponding values as shown below:
- `INSERT INTO table_name (column1, column2, column3,..) VALUES ( value1, value2, value3,..);`

INSERT multiple rows

- **To insert multiple rows in a table using Single SQL Statement:**

```
INSERT INTO
table_name(Column1,Column2,Column3,.......)

VALUES (Value1, Value2,Value3,.....),

    (Value1, Value2,Value3,.....),

    (Value1, Value2,Value3,.....);
```

Using SELECT in INSERT INTO Statement

- We can use the **SELECT** statement with **INSERT INTO** statement to copy rows from one table and insert them into another table.
- The use of this statement is like that of **INSERT INTO** statement.
- The difference is that the **SELECT** statement is used here to select data from a different table.
- The different ways of using **INSERT INTO SELECT** statement are shown below:

- **Inserting all columns of a table.** We can copy all the data of a table and insert into in a different table:
- `INSERT INTO first_table SELECT * FROM second_table;`

- **Inserting specific columns of a table.** We can copy only those columns of a table which we want to insert into in a different table:
- `INSERT INTO first_table(names_of_columns1) SELECT names_of_columns2 FROM second_table WHERE condition;`

UPDATE Statement

- The **UPDATE** statement in SQL is used to update the data of an existing table in database.
- We can update single columns as well as multiple columns using **UPDATE** statement as per our requirement.

- **Basic syntax:**

```
UPDATE table_name
SET column1 = value1, column2 = value2,...
WHERE condition;
```

UPDATE JOIN

- **Sometimes, you need to update data in a table based on values in another table. In this case, you can use the PostgreSQL UPDATE join syntax as follows:**

```
UPDATE table1
SET table1.column1 = new_value
FROM table2
WHERE table1.column2 = table2.column2;
```

DELETE Statement

- The **DELETE** Statement in SQL is used to delete existing records from a table.
- We can delete a single record or multiple records depending on the condition we specify in the WHERE clause.

- **Basic syntax:**

```
DELETE FROM table_name
WHERE some_condition;
```

- **Delete all the records:**
- `DELETE FROM table_name ;`

DELETE JOIN

- PostgreSQL doesn't support the DELETE JOIN statement.
- However, it does support the USING clause in the DELETE statement that provides similar functionality as the DELETE JOIN.

```
DELETE FROM table_name1
USING table_expression
WHERE condition
RETURNING returning_columns;
```

```
DELETE FROM t1
USING t2
WHERE t1.id = t2.id
```

SQL DQL: SELECT statement

- **SELECT** is the most used statement in SQL.
- The **SELECT** Statement in SQL is used to retrieve or fetch data from a database.
- We can fetch either the entire table or according to some specified rules.
- The data returned is stored in a result table.
- This result table is also called result-set.

- **Basic syntax:**

```
SELECT column1,column2 FROM table_name;
```

- **To fetch the entire table or all the fields in the table:**
- `SELECT * FROM table_name;`

Column and Table Aliases

- **Alias allows you to assign a column(s) or table(s) in the select list of a SELECT statement temporary name(s).**
- **The alias exists temporarily during the execution of the query.**

```
SELECT column_name AS column_alias
FROM table_name AS table_alias;
```

OR

```
SELECT column_name column_alias
FROM table_name table_alias;
```

- **Column aliases that contain spaces:**

```
SELECT column_name "column_alias"
FROM table_name table_alias;
```

PostgreSQL ORDER BY

- When you query data from a table, the SELECT statement returns rows in an unspecified order. To sort the rows of the result set, you use the ORDER BY clause in the SELECT statement.
- The ORDER BY clause allows you to sort rows returned by a SELECT clause in ascending or descending order based on a sort expression.
- The following illustrates the syntax of the ORDER BY clause:

```
SELECT select_list

FROM table_name

ORDER BY sort_expression1 [ASC | DESC],

...

sort_expressionN [ASC | DESC];
```

PostgreSQL evaluates the clauses in the SELECT statment in the following order: FROM, SELECT, and ORDER BY:

PostgreSQL ORDER BY clause and NULL

- In the database world, NULL is a marker that indicates the missing data or the data is unknown at the time of recording.
- When you sort rows that contains NULL, you can specify the order of NULL with other non-null values by using the NULLS FIRST or NULLS LAST option of the ORDER BY clause:

```
ORDER BY sort_expresssion [ASC | DESC] [NULLS FIRST | NULLS LAST]
```

- The NULLS FIRST option places NULL before other non-null values and the NULL LAST option places NULL after other non-null values.
- If you use the ASC option, the ORDER BY clause uses the NULLS LAST option by default.

PostgreSQL SELECT DISTINCT

- The DISTINCT clause is used in the SELECT statement to remove duplicate rows from a result set.
- The DISTINCT clause keeps one row for each group of duplicates. The DISTINCT clause can be applied to one or more columns in the select list of the SELECT statement.
- The following illustrates the syntax of the DISTINCT clause:

```
SELECT DISTINCT column1
FROM table_name;
```

- If you specify multiple columns, the DISTINCT clause will evaluate the duplicate based on the combination of values of these columns.

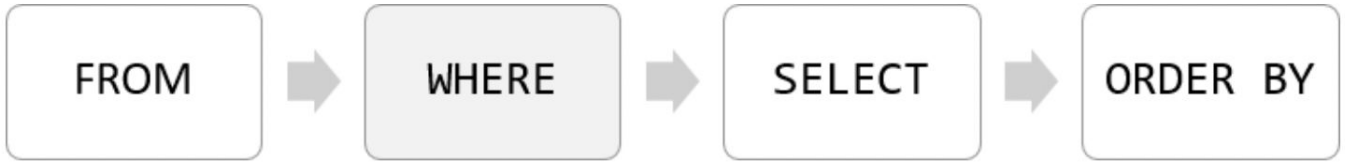
```
SELECT DISTINCT column1, column2
FROM table_name;
```

PostgreSQL WHERE

- The SELECT statement returns all rows from one or more columns in a table.
- To select rows that satisfy a specified condition, you use a WHERE clause:

```
SELECT select_list
FROM table_name
WHERE condition
ORDER BY sort_expression
```

- The WHERE clause appears right after the FROM clause of the SELECT statement.
- The WHERE clause uses the condition to filter the rows returned from the SELECT clause.
- The condition must evaluate to true, false, or unknown. It can be a boolean expression or a combination of boolean expressions using the AND and OR operators.
- PostgreSQL evaluates the WHERE clause after the FROM clause and before the SELECT and ORDER BY clause:



Operators in WHERE clause

Operator	Description
=	Equal
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
<>	Not equal. **Note:** In some versions of SQL this operator may be written as !=
BETWEEN	Between a certain range
LIKE	Search for a pattern
IN	To specify multiple possible values for a column

PostgreSQL IN

- You use IN operator in the WHERE clause to check if a value matches any value in a list of values.
- The syntax of the IN operator is as follows:

```
SELECT select_list
FROM table_name
WHERE condition IN (value1,value2,...)
ORDER BY sort_expression
```

works like ”equal” sign (=)

- You can combine the IN operator with the NOT operator to select rows whose values do not match the values in the list.

```
SELECT select_list
FROM table_name
WHERE condition NOT IN (value1,value2,...)
ORDER BY sort_expression
```

works like ”not equal” sign (<>)

PostgreSQL BETWEEN

- You use the BETWEEN operator to match a value against a range of values. The following illustrates the syntax of the BETWEEN operator:

```
SELECT select_list
FROM table_name
WHERE value BETWEEN low AND high;
ORDER BY sort_expression
```

- If the value is greater than or equal to the low value and less than or equal to the high value, the expression returns true, otherwise, it returns false.

- If you want to check if a value is out of a range, you combine the NOT operator with the BETWEEN operator as follows:

```
SELECT select_list
FROM table_name
WHERE value NOT BETWEEN low AND
high;
ORDER BY sort_expression
```

PostgreSQL LIKE and NOT LIKE

- The PostgreSQL (NOT) **LIKE** operator is used to match text values against a pattern using wildcards. If the search expression can be matched to the pattern expression, the LIKE operator will return true, which is **1**.
- There are two wildcards used in conjunction with the **LIKE** and **NOT LIKE** operators:
- The percent sign (%) - represents zero, one, or multiple numbers or characters.
- The underscore (_) - represents a single number or character.
- These symbols can be used in combinations.

```
SELECT FROM table_name
WHERE column (NOT)LIKE '%XXX%';
```

OR

```
SELECT FROM table_name
WHERE column (NOT)LIKE  '_XXX_';
```

PostgreSQL IS NULL

- In the database world, NULL means missing information or not applicable.
- NULL is not a value; therefore, you cannot compare it with any other values like numbers or strings.
- The comparison of NULL with a value will always result in NULL, which means an unknown result.
- In addition, NULL is not equal to NULL, so the following expression returns NULL: NULL = NULL;
- To check whether a value is NULL or not, you use the IS NULL operator instead:

```
SELECT select_list
FROM table_name
WHERE value IS NULL;
```

- To check if a value is not NULL, you use the IS NOT NULL operator:

```
SELECT select_list
FROM table_name
WHERE value IS NOT NULL;
```

PostgreSQL LIMIT

- PostgreSQL LIMIT is an optional clause of the SELECT statement that constrains the number of rows returned by the query.
- The following illustrates the syntax of the LIMIT clause:

```
SELECT select_list
FROM table_name
ORDER BY sort_expression
LIMIT row_count
```

- The statement returns row_count rows generated by the query.
- If row_count is zero, the query returns an empty set.
- In case row_count is NULL, the query returns the same result set as it does not have the LIMIT clause.
- In case you want to skip a number of rows before returning the row_count rows, you use OFFSET clause placed after the LIMIT clause as the following statement:

```
SELECT select_list
FROM table_name
ORDER BY sort_expression
LIMIT row_count OFFSET rows_to_skip
```

PostgreSQL FETCH

- To constrain the number of rows returned by a query, you often use the LIMIT clause. The LIMIT clause is widely used by many relational database management systems such as MySQL, H2, and HSQLDB. However, the LIMIT clause is not a SQL-standard.
- To conform with the SQL standard, PostgreSQL supports the FETCH clause to retrieve a few rows returned by a query. Note that the FETCH clause was introduced in SQL:2008.
- The following illustrates the syntax of the PostgreSQL FETCH clause:

```
SELECT select_list

FROM table_name

ORDER BY sort_expression

OFFSET start { ROW | ROWS }

FETCH { FIRST | NEXT } [ row_count ] { ROW | ROWS } ONLY
```

PostgreSQL SERIAL and SEQUENCE

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

- When creating a new table, the sequence can be created through the SERIAL pseudo-type as follows:

```
CREATE TABLE table_name(
id SERIAL );
```

By assigning the SERIAL pseudo-type to the id column, PostgreSQL performs the following:

- First, create a sequence object and set the next value generated by the sequence as the default value for the column.
- Second, add a NOT NULL constraint to the id column because a sequence always generates an integer, which is a non-null value.
- Third, assign the owner of the sequence to the id column; as a result, the sequence object is deleted when the id column or table is dropped

PostgreSQL SERIAL and SEQUENCE

- By definition, a sequence is an ordered list of integers. The orders of numbers in the sequence are important. For example, {1,2,3,4,5} and {5,4,3,2,1} are entirely different sequences.
- A sequence in PostgreSQL is a user-defined schema-bound object that generates a sequence of integers based on a specified specification.
- To create a sequence in PostgreSQL, you use the CREATE SEQUENCE statement:

```
CREATE SEQUENCE [ IF NOT EXISTS ] sequence_name
[ AS { SMALLINT | INT | BIGINT } ]
[ INCREMENT [ BY ] increment ]
[ MINVALUE minvalue | NO MINVALUE ]
[ MAXVALUE maxvalue | NO MAXVALUE ]
[ START [ WITH ] start ] [ CACHE cache ]
[ [ NO ] CYCLE ]
[ OWNED BY { table_name.column_name | NONE } ]
```

PostgreSQL SERIAL and SEQUENCE

- **Behind the scenes, the following statement:**

```
CREATE TABLE table_name(
id SERIAL );
```

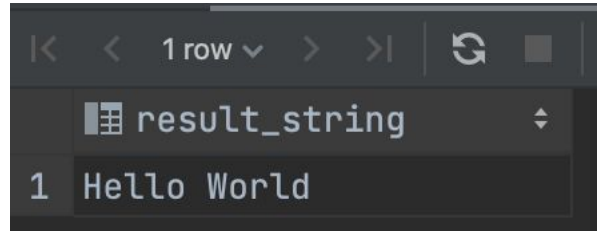
- **is equivalent to the following statements:**

```
CREATE SEQUENCE table_name_id_seq;

CREATE TABLE table_name (
id integer NOT NULL DEFAULT nextval('table_name_id_seq') );

ALTER SEQUENCE table_name_id_seq OWNED BY table_name.id;
```

PostgreSQL Built-in Functions: CONCAT

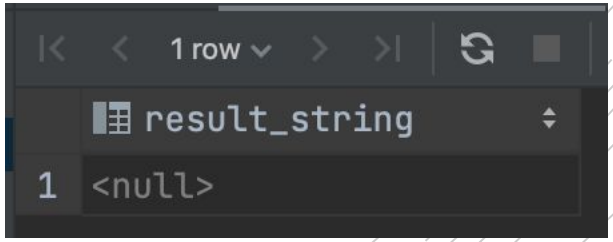
- To concatenate two or more strings into one, you use the string concatenation operator || as the following example:

```
SELECT 'Hello' || ' ' || 'World' AS
result_string;
```



- The following statement concatenates a string with a NULL value:

```
SELECT 'Concat with ' || NULL AS result_string;
```



PostgreSQL Built-in Functions: CONCAT

- The CONCAT function accepts a list of arguments.
- The argument needs to be convertible to a string.
- A string in this context means any of the following data types: char, varchar, or text.
- Unlike the concatenation operator ||, the CONCAT function ignores NULL arguments.

```
SELECT CONCAT(str1, str2);
```

- Besides the CONCAT function, PostgreSQL also provides you with the CONCAT_WS function that concatenates strings into one separated by a particular separator.
- By the way, WS stands for **w**ith **s**eparator.

```
SELECT CONCAT_WS(separator, str1, str2);
```

PostgreSQL LENGTH Function

- The length function accepts a string as a parameter. A string can be any of the following data types:
- character or char
- character varying or varchar
- text
- The length function returns the number of characters in the string.

```
SELECT LENGTH(string);
```

PostgreSQL CAST operator

- There are many cases that you want to convert a value of one data type into another. PostgreSQL provides you with the CAST operator that allows you to do this.
- The following illustrates the syntax of type CAST:

```
CAST ( expression AS target_type );
```

- In this syntax:
- First, specify an expression that can be a constant, a table column, an expression that evaluates to a value.
- Then, specify the target data type to which you want to convert the result of the expression.

PostgreSQL CAS T operator

- **Besides the type CAST syntax, you can use the following syntax to convert a value of one type into another:**

```
expression::type;

SELECT '100'::INTEGER,

'01-OCT-2015'::DATE;
```

PostgreSQL CASE

- The PostgreSQL CASE expression is the same as IF/ELSE statement in other programming languages.
- It allows you to add if-else logic to the query to form a powerful query.
- Since CASE is an expression, you can use it in any places where an expression can be used e.g.,SELECT, WHERE, GROUP BY, or HAVING clause.
- The CASE expression has two forms: general and simple form.

PostgreSQL CASE

- **The following illustrates the general form of the CASE statement:**

```
CASE
    WHEN condition_1 THEN result_1
    WHEN condition_2 THEN result_2
    [WHEN ...]
    [ELSE else_result]
END
```

- **Simple PostgreSQL CASE expression:**

```
CASE expression
    WHEN value_1 THEN result_1
    WHEN value_2 THEN result_2
    [WHEN ...]
ELSE
    else_result
END;
```

PostgreSQL DATE Functions

- Get the current date:

```
SELECT NOW()::date;
```

OR

```
SELECT CURRENT_DATE;
```

- To output a date value in a specific format, you use the TO_CHAR() function.
- The TO_CHAR() function accepts two parameters: the first parameter is the value that you want to format, and the second one is the template that defines the output format.
- For example, to display the current date in dd/mm/yyyy format, you use the following statement:

```
SELECT TO_CHAR(NOW()::DATE, 'dd/mm/yyyy');
```

- Or to display a date in the format like Jun 22, 2016, you use the following statement:

```
SELECT TO_CHAR(NOW() :: DATE, 'Mon dd, yyyy');
```

PostgreSQL DATE Functions

- To get the interval between two dates, you use the minus (-) operator.
- The following example gets service days of employees by subtracting the values in the hire_date column from today's date:

```
SELECT first_name, last_name, now() - hire_date as diff
FROM employees;
```

```
 first_name | last_name |            diff
------------+-----------+----------------------------
 Shannon    | Freeman   | 4191 days 08:25:30.634458
 Sheila     | Wells     | 4922 days 08:25:30.634458
 Ethel      | Webb      | 5652 days 08:25:30.634458
(3 rows)
```

PostgreSQL DATE Functions

- To calculate age at the current date in years, months, and days, you use the AGE() function.
- The following statement uses the AGE() function to calculate the ages of employees in the employees table.

```
SELECT employee_id, first_name, last_name, AGE(birth_date)
FROM employees;
```

```
 employee_id | first_name | last_name |           age
-------------+------------+-----------+-------------------------
           1 | Shannon    | Freeman   | 36 years 5 mons 22 days
           2 | Sheila     | Wells     | 38 years 4 mons 18 days
           3 | Ethel      | Webb      | 41 years 5 mons 22 days
(3 rows)
```

PostgreSQL DATE Functions

- To get the year, quarter, month, week, day from a date value, you use the EXTRACT() function.
- The following statement extracts the year, month, and day from the birth dates of employees:

```
SELECT employee_id, first_name, last_name,
EXTRACT (YEAR FROM birth_date) AS YEAR,
EXTRACT (MONTH FROM birth_date) AS MONTH,
EXTRACT (DAY FROM birth_date) AS DAY
FROM employees;
```

```
 employee_id | first_name | last_name | year | month | day
-------------+------------+-----------+------+-------+-----
           1 | Shannon    | Freeman   | 1980 |     1 |   1
           2 | Sheila     | Wells     | 1978 |     2 |   5
           3 | Ethel      | Webb      | 1975 |     1 |   1
(3 rows)
```