PostgreSQL Notes

Tomás Bilal Iaquinta

July 2020

Contents

Chapter 1

SQL Notes

- SQL is case insensitive, by convention it is recommended to use uppercase for **SQL keywords**.
- (*): queries all data.
- Be wary, the *'s use is not recommended, since you may be querying unnecessary data.
- The index of the first position of an array is 1 in SQL.
- Single line comments may be added by using the `--` symbol.
- Using the `/*` and `*/` symbols we can also do comments on a single line or in multiple lines. We can the add the comment in between the text or put each symbol respectively in the query, with the lines of code in between. In this case, it is recommend that we also put a `*` symbol in each line of code commented.
- Since *Nulls* are not numerical data, we cant use the `=` operator to check if a row in *Null*, we'll have to use `IS` instead.
- (%): pattern wildcard for matching any sequence of characters.
- (_): pattern wildcard for matching a single character.
- In SQL we can use what is called as *ordinal position notation*, which is a way to refer to columns as an integer, according to its position on the `SELECT` statement, instead of using the full column's name or its alias. We can use this to `GROUP BY int`, or `ORDER BY int`.

- By default indicating `JOIN`, SQL applies and `INNER JOIN`.
- For an `OUTER JOIN`, we can specify only `LEFT JOIN` or `RIGHT JOIN`. When indicating the direction of the join, SQL will automatically know we are referring to the outer join.
- Most of the times, the outer joins are `LEFT`, it is uncommon to have `RIGHT` joins. `Keep in mind that SQL refers to date in the format` *`YYYY-MM-DD`*.
- By default, SQL will look for tables from the *public schema*, if the tables we want to query from are in a different *schema*, we will have to specify: `FROM schema.table`.

Chapter 2

Statements

2.1 SELECT Statement

Syntax:

```
SELECT column_1, column_2, ... FROM table;
```

Selects columns from which to query the data.

2.1.1 SELECT DISTINCT Statement

Syntax:

```
SELECT DISTINCT column_1, column_2, ... FROM table;
```

By adding the `DISTINCT` keyword, the `SELECT` statement will only query unique values. This is useful for eliminating duplicates.

2.1.2 SELECT WHERE Statement

Syntax:

```
SELECT column_1, column_2, ...
FROM table WHERE conditions;
```

The `WHERE` keyword allow us to set a *condition*, in order to indicate from which **rows** to select. The condition can apply to a different column than the one we are selecting from.

2.2 Aggregate functions

2.2.1 COUNT Statement

Syntax:

```
SELECT COUNT(*) FROM table;
```

The `COUNT(*)` function returns the number of rows queried by a `SELECT` statement. When applying this function, the table is scanned sequentially.

It is possible to specify a column using `COUNT(column)` for readability. This function does not consider `NULL` valus in the column.

It is also possible to count the distinct rows for a column using `SELECT COUNT (DISTINCT column)`.

2.2.2 MIN, MAX, SUM and AVG

Syntax:

```
SELECT MAX(column_1) FROM table;
```

Aggregate functions return the desired value, we can specify how many decimal places we want returned by adding the `ROUND` function before the aggregate, using for example:

```
SELECT ROUND(MAX, 2) ...;
```

2.3 LIMIT Statement

Syntax:

```
SELECT ... LIMIT int;
```

`LIMIT` allows us to specify the number of rows we want to get by the query. This statement usually goes at the end of a query.

2.4 ORDER BY Statement

Syntax:

```
SELECT ... ORDER BY column_1 ASC / DESC;
```

When querying data from a table, the rows are returned in the order that they were inserted in the table. We can sort the resulting rows using

the `ORDER BY` statement. If we leave the argument blank, it will sort by `ASC` order.

We can sort by multiple columns by using a comma to separate them. [**!**] Be aware that *PostgreSQL* allows the querying of different rows than the ones we are ordering by, this may not be the case using other SQL alternatives. Because of this, it is a good practice to `SELECT` the column from which we are ordering by.

2.5 Logical statements

2.5.1 BETWEEN Statement

Syntax:

```
... WHERE value BETWEEN low AND high;
```

We use the `BETWEEN` statement to match a value against a range of values. It is possible to rewrite it by using the `<=` and `>=` operators. It offers a more readable version to the `SELECT WHERE` statement when comparing values.

It is possible to check for a value out of a range with the `NOT BETWEEN` operator.

2.5.2 IN Statement

Syntax:

```
... WHERE value IN (value1,value2,...);
```

It is used with the `WHERE` statement to check if a value matches any in a list of values. The expression returns *True* if the value matches anyone in the list. It also allows to select value that is `NOT IN`.

It is a more readable and faster option than to use several `OR` statements as logical operators.

The list of values is not limited to a list of numbers or strings, but also a result set of a `SELECT` statement as follows:

```
... WHERE value IN(SELECT value FROM table);
```

This is now as a **subquery**.

2.6 LIKE Statement

Syntax:

```
... WHERE column_1 LIKE '%pattern%';
```

By using the *wildcards* `%` and _, we construct a pattern to use in the statement. Then, we indicate whether to query values that are `LIKE` or `NOT LIKE` it.

The `LIKE` statement is case sensitive, *PostgreSQL* offers the alternative of using the `ILIKE` statement, which isn't case sensitive.

2.7 GROUP BY Statement

Syntax:

```
SELECT column_1, aggregate_function(column_2)
FROM table GROUP BY column_1;
```

The `GROUP BY` statement divides the rows returned from `SELECT` into groups. It's common to use the along with aggregate functions. Without it, it works similar to a `SELECT DISTINCT` statement.

PostgreSQL is flexible in the columns indicated for the aggregate function and the `SELECT` and `GROUP BY` statements. Other SQL engines may have rules, such as grouping by the same column which is selected or in which the aggregate function is applied. Some SQL engines may also require to always apply an aggregate function.

It is a good practice to indicate the same column in the `GROUP BY` and `SELECT` statements.

2.8 HAVING Statement

Syntax:

```
SELECT column_1, aggregate_function(column_2)
FROM table GROUP BY column_1
HAVING condition;
```

It is often used in conjunction with the `GROUP BY` statement to filter groups that don't satisfy a certain condition. It sets the condition for group rows after appying `GROUP BY`, whereas the `WHERE` statement sets the condition for individual rows before `GROUP BY`

2.9 AS Statement

Syntax:

```
... column_1 AS alias ...;
```

It allow as to rename columns or table selections with an alias. It is possible to use it along with different statements, not only `SELECT`. SQL also interprets blank spaces after a space or column as an implicit `AS` statement.

2.10 CASE, WHEN, THEN and END Statements

Syntax:

```
SELECT ...
CASE WHEN condition THEN result END operation
```

The `CASE` statement is SQL ways of handling *if ¿ then* logic. This statement usually goes in the `SELECT` clause, and must include the `WHEN`, `THEN` and `END` components. It is optional to add an `ELSE` component.

For example, this statement can be useful for creating a labeled group column for a certain condition. In this case we will indicate `END AS column_name` to create a new column for the query. We can also specify an aggregation and create a new column, such as: `END AS column_name, COUNT(*) AS column_alias`.

Let's say that we would now want to do a `COUNT` according to this condition. In that case after we create the labeled column , we calculate the count in a new column just after and then we count by it in a new column

Chapter 3

Joining tables

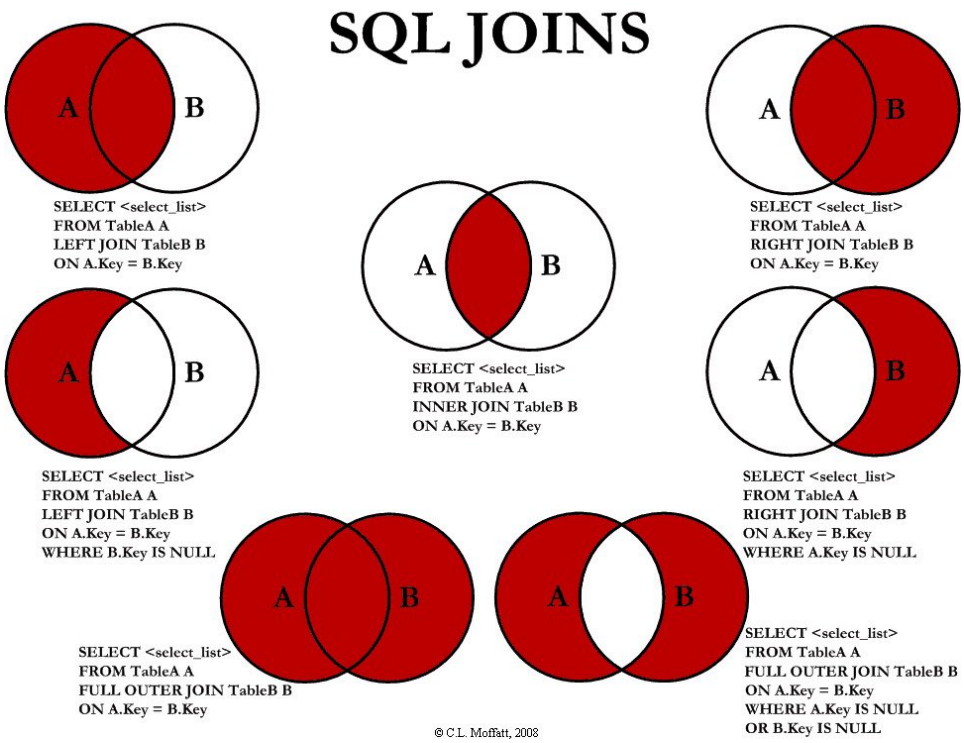

Figure 3.1: SQL Joins as Venn diagrams.

- `INNER JOIN`: Produces only the set of records that match both in table

A and table B.

- `FULL OUTER JOIN`: Produces the set of all records in both tables, with matching records from both where available. If there is no match, the missing side will contain *Null*.

- `LEFT/RIGHT OUTER JOIN`: Produces a complete set of records from the respective table, with the matching records where available. If there is no match, the other side will contain *Null*.

- `LEFT/RIGHT OUTER JOIN` **with** `WHERE`: We can get the set of records that are only in the `FROM` table, and not in the joining one. This is done by adding a `WHERE` statement, to exclude the records we don't want. This can be done doing:

```
SELECT * FROM TableA
LEFT OUTER JOIN TableB
ON TableA.column_1 = TableB.column_1
WHERE TableB.column_2 IS null;
```

- `FULL OUTER JOIN` **with** `WHERE`: This way we can produce the set of records which are unique to both tables. This is done with a `WHERE` statement in which we exclude the records we don't want from both sides. For example we can say:

```
SELECT * FROM TableA
FULL OUTER JOIN TableB
ON TableA.column_1 = TableB.column_1
WHERE TableA.column_2 IS null
OR TableB.column_2 IS null;
```

- `INEQUALITY JOIN`: We can add a conditional statement for extra filtering on the returned rows, in which we will only get the rows that are *True* for that condition.

3.1 INNER JOIN

Syntax:

```
SELECT A.column_1, A.column_2, B.column_3, B.column_4
FROM A
INNER JOIN B ON A.column_1 = B.column_5;
```

The `INNER JOIN` statement returns rows in the main table that have the corresponding rows in the joining table.

First, we have to specify the column in both tables from which we want to `SELECT`. Second, we specify the main table in the `FROM` statement. Third, we indicate the table that the main one joins to, along with the join condition.

PostgreSQL sequentially scans the joining table to check if there is any row that matches the join condition. When it finds a match, it combines the joining columns of both tables into one, and adds the combined row to the returned result set.

[!] In the `SELECT` statement, it is only necessary to specify the table for the columns that have matching name in both. For unique tables it is better to not specify `table.column`.

3.2 UNION

Syntax:

```
SELECT column_1, column_2 FROM table_1
UNION
SELECT column_1, column_2 FROM table_2;
```

The `UNION` statement combines results sets of two or more `SELECT` statements into a single result set. It is commonly used to combine data from similar tables that are not perfectly normalized. Be aware that both queries must return the same number of columns, and the corresponding columns must have compatible data types.

This statement may replace rows in the first query before, after or between the rows in the result set of the second query, so if we want to sort the rows we shall use the `ORDER BY` statement. `UNION` removes all duplicate rows unless the `UNION ALL` statement is used.

Chapter 4

Functions, operators and timestamps

4.1 Timestamps

All available operations, functions and the corresponding syntax to handle timestamps and time zones, are available in the *PostgreSQL* documentation.

Be aware that different SQL engines may use slightly different syntax for timestamps.

4.1.1 EXTRACT() Function

Syntax:

```
... EXTRACT(unit from date) ...;
```

This function is useful for extracting parts from a date. Using it, it's possible to extract many types of time-based information.

The `DATE_PART()` function is equivalent and uses the same syntax for datetime subfield extraction.

4.1.2 DATE TRUNC() Function

Syntax:

```
... DATE_TRUNC(unit, date) ...;
```

With `DATE_TRUNC()` we can get a truncated timestamp for a specific precision. For example, we can use this to assign the same day for several elements.

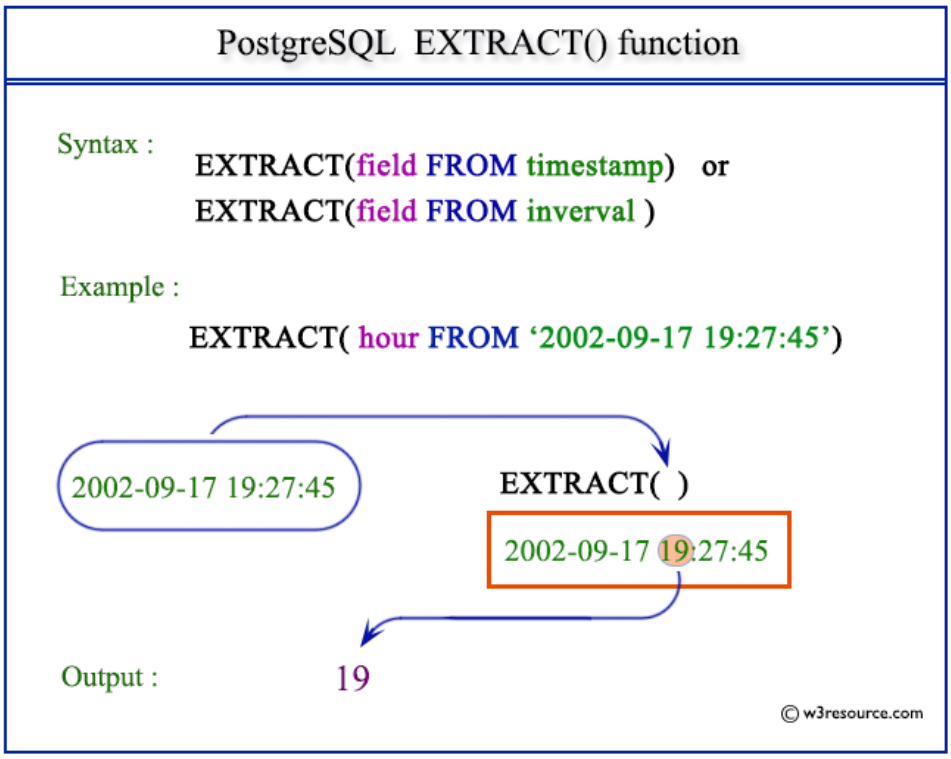

Figure 4.1: `EXTRACT()` function.

4.2 Type operations

4.2.1 Mathematical functions

Most of the mathematical functions on *PostgreSQL* are also available on other SQL engines.

An extended list of all mathematical functions and operators can be found in the *PostgreSQL* documentation.

4.2.2 String functions and operators

PostrgreSQL also allows several useful functions for sting data types.

The available string functions and operators for *PostgreSQL* can be found in its documentation page.

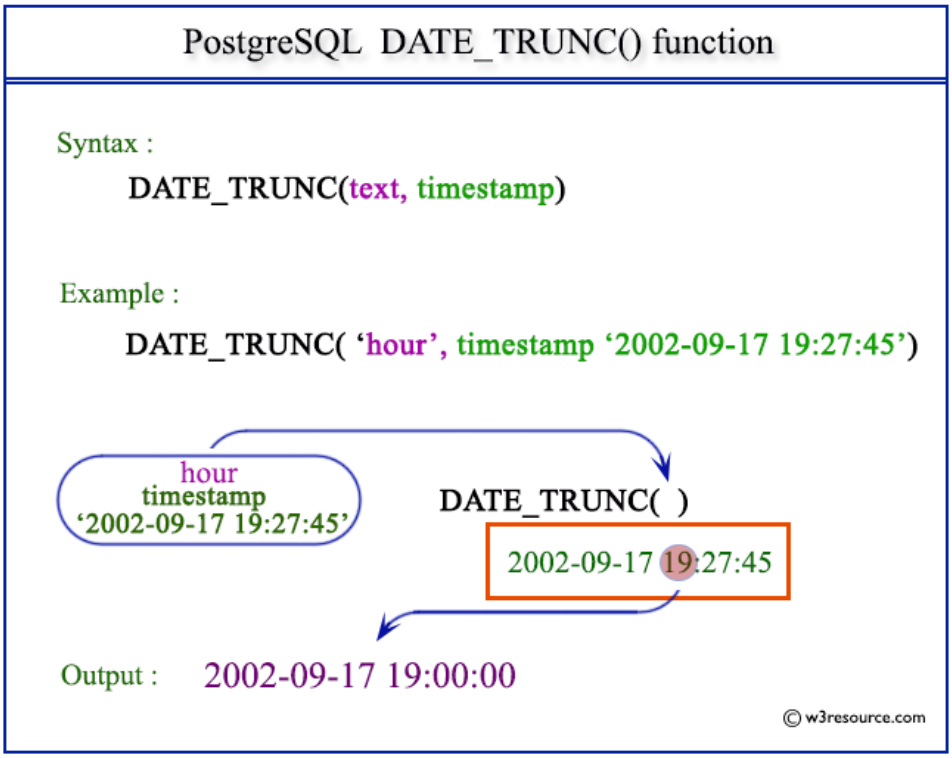

Figure 4.2: `DATE_TRUNC()` function.

Chapter 5

Subquery and self join

5.1 Subquery

Syntax:

```
...
(SELECT ... FROM ...) subquery_alias
...;
```

A subquery is a query nested within another query. It allows us to use multiple `SELECT` statements where we can have a query within a query. Subqueries are very useful to simplify and obtain results faster, avoiding extra steps.

We put the nested query between brackets and we must assign an alias to it if we are using it within a `SELECT` statement (not in a conditional statement). There is no general syntax for subqueries, since they can be used within different statements. It is important to note that the *inner query* (subquery) will run prior to the *outer query*, which runs across the first one's result.

It is recommended to add indents to the subquery in order to get a more readable input.

5.2 CTE: WITH Statement

Syntax:

```
WITH CTE_alias AS (... subquery ...)
SELECT ...
FROM CTE_alias
```

...;

The `WITH` statement, also called a *Common Table Expression* (or *CTE*), is useful for when we have a subquery that takes a long time to run, in which way we can create a *CTE* to run it only once and save it with an alias.

We can create as many *CTEs* as we want using only one `WITH` statement and then separating the tables with commas. They should always be specified at the beginning of our query.

All in all, *CTEs* provides us a more readable and more efficient option for running subqueries.

5.3 Self join

Syntax:

```
SELECT t1.column_1
FROM table AS t1, table AS t2
WHERE
t1.column_2 = t2.column_2
AND t_2.column_1 condition;
```

We use it when we want to combine rows with other rows in the same table. We must use a table alias to help SQL distinguish the left table from the right table of the same table.

Self joins are sometimes more efficient than using a subqueries, and may also offer the advantage of not needing to indicate certain sensitive data. Generally queries that refer to the same table can be greatly simplified by re-writing the queries a self join, offering also a more efficient alternative.

Another option is to actually add a `JOIN` statement instead of the comma in the above seen `FOR` statement.

By default an inner join will be applied, however we can use also outer joins.

Chapter 6

Creating databases and tables

6.1 Data types

PostgreSQL supports the following data types:

- Boolean.
- Character.
- Numeric.
- Temporal.
- Special types.
- Array.

.

6.1.1 Boolean

A **Boolean** data type has only two possible values: *True* of *False*, in case the value is unknown, the *Null* value will be used. We use **boolean** or **bool** as a keyword for declaring that a column has this type.

PostgreSQL will automatically convert the data into a **Boolean** column. For example, if we input *1*, *yes* or *t*, the values are converted to *True*, values like *0*, *no* or *f* will be translated to *False*.

When selecting data from a **Boolean** column, *PostgreSQL* will display a *t* for *True*, an *f* for *False*, and a space character for *Null*.

6.1.2 Character

There are three different kinds of this data type:

- A single character: *char*.
- Fixed length character strings: *char(n)*. If the string inserted is shorter than this length n, *PostgreSQL* will pad spaces, if it is longer it will issue an error.
- Variable-length character strings: *varchar(n)*. Being n the maximum length of the sequence. In this case, *PostgreSQL* will not need to pad space.

6.1.3 Numeric

Numeric types can be either **integers** or **floating-point numbers**.

Integers

We have three distinct types of **Integers**:

- **smallint**: small integers are a 2-byte signed integer, that have a range of $(-2^{15},\ 2^{15}-1)$.
- **int**: a 4-byte integer that has a range of $(-2^{31},\ 2^{31}-1)$.
- **serial**: the same as integer, except that *PostgreSQL* populates values in the column automatically. This is similar to the *AUTO_INCREMENT* attribute in other SQL engines.

Floating-point numbers

We have also three distinct types of **Floating-point numbers**:

- **float(n)**: is a floating-point number whose precision is at least n, up to a maximum of 8 bytes.
- **real** or **float8**: is a double-precision 8-byte floating-point number.
- **numeric** or **numeric(p,s)**: is a real number with p digits and s decimals.

6.1.4 Temporal

The temporal data types store date and time related data, including:

- **date**: stores date data.
- **time**: stores time data.
- **timestamp**: stores date and time.
- **interval**: stores the difference in timestamps.
- **timestamptz**: stores both timestamp and timezone data.

6.1.5 Keys

Primary keys

A column or group of columns that has a unique value for each column. It is usually the first column in most tables. A table can have only one **primary key**, it is a good practice that every table has one.

When we add a **primary key** table, *PostgreSQL* creates a unique index on the column (or group of columns) and defines the key. If we add new rows, the index will automatically increment.

Foreign keys

A column or group of columns in a table that uniquely identifies another table's **primary key**. The table that contains the **foreign key** is called the *referencing table* or *child table*, and the table which the key references is called the *referenced table* or *parent table*.

A table can have multiple foreign keys depending on its relationship with other tables.

6.2 Creating tables

We can create tables by using the `CREATE TABLE` statement as follows:

```
CREATE TABLE table_name
(column_name TYPE column_constraint,
table_constraint)
INHERITS existing_table_name;
```

We can also create a table based in another table's scheme by putting in the parenthesis `(LIKE original_table)`.

6.2.1 Constraints

The **constraints** defines the rules for each column and table.

Column constraints

- `NOT NULL`: the values of the column can't be *Null*.
- `UNIQUE`: the values of the column are unique across the whole table. However, we can have *Null* vales, since they are not considered in *PostgreSQL*. SQL standars only allows one *Null* value for this constraint.
- `PRIMARY KEY`: this constraint is a combination of the previous ones. If the **primary key** contains multiple columns, we must use table-level **constraints**.
- `CHECK`: enables to check a condition when we insert or update data.
- `REFERENCES`: constraints the value of the foreign key column.

Table constraints

- `UNIQUE(column_list)`: to force the value stored in the listed columns to be unique.
- `PRIMARY_KEY(column_list)`: to define a **primary key** that consist of multiple columns.
- `CHECK(condition)`: to check a condition when inserting or updataing data.
- `REFERENCES`: to constrain the value stored in the columns that references other columns in another table.

6.3 Adding and modifying data in a table

6.3.1 INSERT INTO Statement

Syntax:

```
INSERT INTO table(column_1, column_2, ...)
VALUES (value_1, value_2, ...),
       (value_1, value_2, ...);
```

This statement allows the insertion of one or more rows into a table. If we input values for less than the total number of columns of the table, the engine will try to input *Null* for the columns unspecified, so we have to be careful if have a `NOT NULL` constraint.

To insert data from other table we have to replace the `VALUES` statement for a `SELECT` statement, that queries the desired values.

6.3.2 UPDATE Statement

Syntax:

```
UPDATE table
SET column_1 = value_1 ,
    column_2 = value_2 ,
WHERE condition ;
```

We use this statement to change all the values of a column in the table. By adding a `WHERE` statement, we only change the values if a certain condition is satisfied.

[**!**] At the end of the query we can add a `RETURNING` statement, and indicate the comma separated columns that we want the engine to return after applying the operation. In this way, we will see the entries that where modified.

6.3.3 DELETE Statement

```
DELETE FROM table
WHERE condition ;
```

This statement is used to delete rows from a table. If we omit the `WHERE` clause, all rows will be deleted.

This statement returns the number of deleted rows.

6.4 Modifying a table

Syntax:

```
ALTER TABLE table_name action ;
```

We use the `ALTER TABLE` statement when we want to change the existing table structure. *PostgreSQL* provides many actions that allows us to:

- Add, remove or rename a column.

- Set a default value for a column.
- Add a `CHECK` constraint to a column.
- Rename a table.

Some of the available *actions* we can perform are:

- `ADD COLUMN`.
- `DROP COLUMN`.
- `RENAME COLUMN`.
- `RENAME`.

For example, if we want to rename a column we have to indicate `RENAME COLUMN column_1 TO column_a`. We also have to specify the `TO` statement when using `RENAME` to rename a table.

6.5 Removing a table

Syntax:

```
DROP TABLE [IF EXISTS] table;
```

The `IF EXISTS` term is an optional term to avoid an error if the table doesn't exist. In this case, *PostgreSQL* will give us a notice, instead of an error.

The is another additional term that is `RESTRICT` which will prevents the elimination of a table if there are any objects depending on its information. By default, *PostgreSQL* will consider this term so we don't need to input it.

If we want to also eliminate the objects that depend on the table we want to drop, we can add the **CASCADE** term.

6.6 Handling databases

Similarly than for a table we can create a database with an SQL query as follows:

```
CREATE DATABASE name
WITH OWNER = admin
ENCODING = 'UTF-8'
...
```

Using *pgAdmin* we have a simple alternative to specifying the query, which is creating a new database by right-clicking the *Databases* section of the dropdown menu.

The same procedure can be applied to delete a database, and restore a schema.

6.7 Views

By using the `CREATE VIEW` statement, we can quickly access a previously executed query. This helps us in simplifying the access to the data of a complex query. In *PostgreSQL* a view is a logical table that represents its data through a `SELECT` statement. A view does not store data physically except for a materialized view. We can also alter created views or eliminate them as follows:

```
ALTER VIEW name_of_view RENAME TO new_view_name;
```

Like with the tables we can drop a previously created view by using:

```
DROP VIEW [IF EXISTS] name_of_view;
```

Chapter 7

Data cleaning

7.1 LEFT(), RIGHT() and LENGTH() Functions

The `LEFT()` function allows us to extract the characters from the left side of a string and present them as separate strings for each row in a certain column, by indicating `LEFT(column_1, int) AS str_alias`, where *int* indicates the number of characters to pull. We can operate equivalently with the `RIGHT()` function.

It is also possible to indicate an operation that leads to an *int*, rather than only a number. For example, we can extract a number of characters (on the left or right of every column) by using *LENGTH(column_n).*

7.2 POSITION(), STRPOS() and SUBSTR() Functions

`POSITION()` takes a character and a column name by indicating `POSITION( 'char' IN column_name) AS pos`, this will output the position in which the desired string is found on each column's row.

`STRPOS()` does the same but its syntax is `STRPOS(column_name, 'char') AS pos`. Both function are case sensitive, so we can use the `UPPER()` and `LOWER()` function to convert the string to upper or lowercase respectively.

`SUBSTR()` does the inverse of the two previous functions. In this case, we indicate the start and end position in a string to get a substring as follows: `SUBSTR(string, start_int, N_int) AS substr`, where `N_int` represents the number of characters we want to get.

7.3 CONCAT() Function and Piping

This function combines strings from two or more columns into a single one. We can indicate column names for the function as well as strings. Let's say that for example we have a first and last name in different columns and we want to create a new column with the full name, so we will say: `CONCAT(first_name, ' ', last_name) AS full_name`.

Alternatively we can use piping `'||'` to apply the same concatenation of strings. We only need the pipes and an alias for the new column.

7.4 REPLACE() Function

`REPLACE` is a simple function that allows us to change a substring in a column to one that we desire by doing: `REPLACE(column_name, substr, new_substr)` For example, this can be useful when eliminating blank spaces.

7.5 CAST() Function

This function allows us to change columns from one date type to other, which we will use saying `CAST(column AS type_target)`. It is also possible to use `'::'` for the same purpose.

For example, by indicating a date with formatted strings we can say `CAST(YYY-MM-DD AS DATE)`.

7.6 COALESCE()

The `COALESCE` function is SQL's way of handling *Null* values, where we can specify a target to replace the *Null*s with, as so: `COALESCE(column_name, target)`.

We might sometimes use `COALESCE` when performing an outer join. In this way we have a quick way of filling all empty values.

Chapter 8

Window Functions

A window function (as introduced in the *PostgreSQL* documentation) performs a calculation across a set of tables rows that are somehow related to the current row. This is comparable to the type of calculation that can be done with an aggregate function. But unlike regular aggregate functions, use of a window function does not cause rows to become grouped into a single output row. Instead, the rows retain their separate identities, while the window function is able to access more than the current row of the query result.

Window functions must always have an `OVER` statement directly following the window function (an aggregate, for example), which indicates how the rows are split up for processing. Within this clause we can add a `PARTITION BY` statement that will divide the rows into different groups of rows or partitions. For each row, the window function will be computed across the rows that fall in the same partition as the current one. We can also control the order in which rows are processed by window functions by placing an `ORDER BY` statement within the main clause.

Be aware that window functions can't be used along with aggregate functions, and can't be included in the `GROUP BY` clause.

8.1 ROW NUMBER(), RANK() and DENSE RANK() Functions

Both are possible window functions that do not aggregate. As its name indicates, the `ROW_NUMBER` function displays the number of a given row within the window we define, starting at 1. It does not require to specify a variable inside the parenthesis.

The same goes for the `RANK` function, this one operates differently such as if two rows have the same value according to the partition, it will display the same number for both, whereas `ROW_NUMBER` will yield the same number. With `ROW`, the function will skip ranks if the prior has more than one row assigned.

The `DENSE_RANK` functions works in the same manner but avoiding the value skipping property.

When we want to perform different functions for the same windows we can use a `WINDOW` statement to create an alias for the window. This clause usually goes between the `WHERE` clause and the `GROUP BY` clause. We indicate: `WINDOWS alias AS (PARTITION BY ... ORDER BY ...).`

In this way, for each function we can say `OVER alias` without using parenthesis.

8.2 LAG() and LEAD() Functions

This functions are useful for comparing rows with preceding or following rows, we can access the adjacent rows which by using the `LAG` and `LEAD` functions respectively.

In this way we only get the preceding or following row's value, which we can then use to perform different calculations and comparisons.

8.3 NTILE() Function

The `NTILE` function allows us to calculate and display a certain percentile as a window function, by indicating `NTILE(N_of_buckets)`. This does not work well if the partition has less number of rows than the number of buckets specified.