Querying Data with SELECT Statements

Part 3

The WHERE Clause

- In many cases, after the rows are taken from the input tables, they should be filtered.
- This is done via the `WHERE` clause.
- The filtering condition is specified after the `WHERE` keyword.
- The condition is a SQL expression that returns a Boolean value.

- Simple examples of WHERE conditions are as follows:
  - ```
    SELECT *
    FROM   car_portal_app.car_model
    WHERE  make = 'Peugeot';
    ```
  - ```
    SELECT *
    FROM   car_portal_app.car
    WHERE  mileage < 25000;
    ```
  - ```
    SELECT *
    FROM   car_portal_app.car
    WHERE  number_of_doors > 3 AND
           number_of_owners <= 2;
    ```
    ```

- Logical operators are commonly used in conditional expressions.
- They are `AND`, `OR`, and `NOT`.
- They take Boolean arguments and return Boolean values.
- Logical operators are evaluated in the following order: `NOT`, `AND`, `OR`, but they have a lower priority than any other operators.
- PostgreSQL tries to optimize the evaluation of logical expressions.
- For example, when the `OR` operator is evaluated, and it's known that the first operand is `True`, PostgreSQL may not evaluate the second operand at all, because the result of `OR` is already known.
- For that reason, PostgreSQL may change the actual order of evaluating expressions on purpose to get the results faster.

- Sometimes, this might cause problems.
- For example, it's not possible to divide by zero, and if you wanted to filter rows based on the result of a division, this would not be correct:
  - ```
    SELECT *
    FROM   t
    WHERE  b/a>0.5 and a<>0;
    ```
- It's not guaranteed that PostgreSQL will evaluate the `a<>0` condition before the other one, `b/a>0.5`, and if a has a value of `0`, this could cause an error.
    ```

- To be safe, you should use a `CASE` statement because the order of evaluation of `CASE` conditions is determined by the statement, as follows:
  - ```
    SELECT *
    FROM   t
    WHERE  CASE WHEN a=0 THEN false
                ELSE b/a>0.5
           END;
    ```
    ```

- There are some other operators and expressions that return Boolean values that are used in conditional expressions:
  - Comparison operators
  - Pattern-matching operators
  - The OVERLAPS operator
  - Row and array comparison constructs
  - Subquery expressions
  - Any function that returns a Boolean or is convertible to Boolean values

- As in the SELECT-list, functions can be used in the `WHERE` clause as well as anywhere in expressions.
- Imagine you want to search for car models whose name is four letters long.
- This can be done using a `length()` function:

```
car_portal=> SELECT *
car_portal-> FROM   car_portal_app.car_model
car_portal-> WHERE  length(model)=4;
 car_model_id |    make    | model
--------------+------------+-------
           47 | KIA        | Seed
           57 | Nissan     | GT-R
           70 | Renault    | Clio
           76 | Skoda      | Yeti
           83 | Toyota     | RAV4
           88 | Volvo      | XC70
           89 | Volvo      | XC90
           90 | Volkswagen | Golf
           99 | Alfa Romeo | Mito
(9 rows)
```

Comparison Operators

- Comparison operators are less (<), more (>), less or equal (<=), more or equal (>=), equal (=), and not equal (<> or !=—these two are synonyms).
- These operators can compare not only numbers but any values that can be compared, for example, dates or strings.

- There is a `BETWEEN` construct that also relates to comparing.
- Consider the following:
  - `x BETWEEN a AND b`
- The preceding code is equivalent to the following:
  - `x >= a AND x <= b`

- The `OVERLAPS` operator checks to see whether two ranges of dates overlap.
- Here is an example:

```
car_portal=> SELECT 1
car_portal-> WHERE  (date '2018-10-15', date '2018-10-31') OVERLAPS (date '2018-10-25', date '2018-11-15');
 ?column?
----------
        1
(1 row)
```

- Formally, comparison operators have different precedence: >= and <= have the highest priority.
- Then comes `BETWEEN`, then `OVERLAPS`, then < and >.
- = has the lowest priority.
- However, it's difficult to come up with a practical example of using several comparison operators in the same expression without any parentheses.

Pattern Matching

- Pattern matching is always about strings.
- There are two similar operators: `LIKE` and `ILIKE`.
- They check whether a string matches a given pattern.
- Only two wildcards can be used in a pattern: an underscore, `_`, for exactly one character (or number) and a percent sign, `%`, for any number of any characters, including an empty string.
- `LIKE` and `ILIKE` are the same, except that the first is case-sensitive and the second is not.

- Apart from `LIKE` and `ILIKE`, there are the `~~` and `~~*` operators, which are equivalent to `LIKE` and `ILIKE`, respectively.
- The `LIKE` operator belongs to the SQL standard; the others don't.
- There are also the `!~~` and `!~~*` operators, which behave like `NOT LIKE` and `NOT ILIKE`, respectively, and they are PostgreSQL-specific.

- For example, to get car models whose names start with `s` and have exactly four characters, you can use the following query:

```
car_portal=> SELECT *
car_portal-> FROM   car_portal_app.car_model
car_portal-> WHERE  model ILIKE 's___';
 car_model_id | make | model
--------------+------+-------
           47 | KIA  | Seed
(1 row)
```

- There are two other pattern matching operators: `SIMILAR` and `~` (the tilde sign).
- They check for pattern matching using regular expressions.
- The difference between them is that `SIMILAR` uses regular expression syntax defined in SQL standard, while `~` uses Portable Operating System Interface (POSIX) regular expressions.

- In the following example, we select all car models whose names consist of exactly two words:

```
car_portal=> SELECT *
car_portal-> FROM   car_portal_app.car_model
car_portal-> WHERE  model ~ '^\w+\W+\w+$';
 car_model_id |     make      |    model
--------------+---------------+--------------
           21 | Citroen       | C4 Picasso
           33 | Ford          | C-Max
           34 | Ford          | S-Max
           52 | Mercedes Benz | S klasse
           53 | Mercedes Benz | C klasse
           54 | Mercedes Benz | A klasse
           57 | Nissan        | GT-R
           79 | Toyota        | Land Cruiser
           95 | Ferrari       | 458 Italia
           96 | Ferrari       | 458 Spider
(10 rows)
```

Row and Array Comparison Constructs

- The `IN` expression is used to check whether a value equals any of the values from a list.
- The expression is as follows:
  - `a IN (1, 2, 3)`
- It's a shorter and cleaner way of implementing the logic than using comparison operators, as follows:
  - `(a = 1 OR a = 2 OR a = 3)`

- SQL allows the use of array types, which are several elements as a whole in one single value.
- Arrays can be used to enrich comparison conditions.
- For example, this checks whether a is bigger than any of x, y, or z:
  - `a > ANY (ARRAY[x, y, z])`
- The preceding code is equivalent to the following:
  - `(a > x OR a > y OR a > z)`

- This checks whether a is bigger than x, y, and z:
  - `a > ALL (ARRAY[x, y, z])`
- The preceding code is equivalent to the following:
  - `(a > x AND a > y AND a > z )`

- The `IN`, `ALL`, and `ANY` (which has a synonym: `SOME`) keywords can also be used with subquery expressions to implement the same logic.
- The result of a subquery can be used in any place where it's possible to use a set of values or an array.
- This makes it possible, for instance, to select records from one table, when some values exist in another table.

- For example, in the `car_portal` database, we have the tables with car models and cars separated for the sake of normalization.
- To get a list of all car models when there is a car of that model, the following query can be used:

```
car_portal=> SELECT *
car_portal-> FROM   car_portal_app.car_model
car_portal-> WHERE  car_model_id IN (SELECT car_model_id
car_portal(>                         FROM   car_portal_app.car);
```

```
 car_model_id |     make      |    model
--------------+---------------+-------------
            2 | Audi          | A2
            3 | Audi          | A3
            4 | Audi          | A4
            5 | Audi          | A5
            6 | Audi          | A6
            7 | Audi          | A8
            8 | BMW           | 1er
            9 | BMW           | 3er
           10 | BMW           | 5er
           11 | BMW           | 6er
           12 | BMW           | 7er
           13 | BMW           | z4
           14 | BMW           | X3
           15 | BMW           | X5
           91 | Volkswagen    | Passat
           92 | Volkswagen    | Phaeton
           93 | Volkswagen    | Tuareg
           94 | Volkswagen    | Scirocco
           95 | Ferrari       | 458 Italia
           96 | Ferrari       | 458 Spider
           98 | Fiat          | 500
           99 | Alfa Romeo    | Mito
(86 rows)
```

- Sometimes an `IN` expression can be replaced by `INNER JOIN`, but not always.
- Consider the following example:

```
car_portal=> SELECT car_model.*
car_portal-> FROM   car_portal_app.car_model INNER JOIN car_portal_app.car
car_portal->        USING (car_model_id);
```

car_model_id	make	model
65	Peugeot	308
61	Opel	Corsa
19	Citroen	C3
53	Mercedes Benz	C klasse
79	Toyota	Land Cruiser
95	Ferrari	458 Italia
9	BMW	3er
31	Ford	Mondeo
93	Volkswagen	Tuareg
74	Skoda	Fabia
57	Nissan	GT-R
30	Ford	Focus
92	Volkswagen	Phaeton
64	Peugeot	208
96	Ferrari	458 Spider
3	Audi	A3
55	Nissan	Almera
26	Daewoo	Nubira
25	Daewoo	Espero
51	Lincoln	Towncar
20	Citroen	C4
2	Audi	A2
2	Audi	A2
2	Audi	A2
2	Audi	A2

(233 rows)

- Although the same table is queried and the same columns are returned, the number of records is greater.
- This is because there are many cars of the same model, and for them, the model is selected several times.

- The `NOT IN` construct with a subquery is sometimes very slow, because the check for the nonexistence of a value is more expensive than the opposite.
- This construct can be replaced with `NOT EXISTS` or, sometimes, with a `LEFT JOIN` and a negative predicate in the `WHERE` clause.

Grouping and Aggregation

- SQL provides a way to get aggregated results of processing several records at a time and then get the results in a single row.
- The easiest example would be counting the total number of records in a table.

The GROUP BY Clause

- The `GROUP BY` clause is used for grouping.
- Grouping means splitting the whole input set of records into several groups, with a view to having only one result row for each group.
- Grouping is performed on the basis of a list of expressions.
- All records that have the same combination of values of grouping expressions are grouped together.
- This means that the groups are identified by the values of expressions defined in the `GROUP BY` clause.
- Usually, it makes sense to include these expressions in the SELECT-list to indicate which group is represented by the result row.

- For example, let's group the cars by `make` and `model` and select the groups:

```
car_portal=> SELECT   make, model
car_portal-> FROM     car_portal_app.car a INNER JOIN car_portal_app.car_model b
car_portal->          ON a.car_model_id=b.car_model_id
car_portal-> GROUP BY make, model;
```

```
    make      |    model
--------------+-------------
 Citroen      | C3
 Toyota       | Prius
 Nissan       | Patrol
 Citroen      | C4 Picasso
 KIA          | Sportage
 Nissan       | GT-R
 Volvo        | XC70
 Toyota       | Yaris
 Toyota       | Land Cruiser
 BMW          | X5
 BMW          | z4
 BMW          | X3
 Volvo        | S50
```

```
 Citroen      | C5
 Ferrari      | 458 Spider
 GMC          | Yukon
 Skoda        | Superb
 Infiniti     | QX4
 Volkswagen   | Tuareg
 Volkswagen   | Phaeton
(86 rows)
```

- It's almost useless just to group rows.
- Usually, some computing is performed on the groups.
- As for the last example, it would be interesting to know how many cars of which model are in the system.
- This is done using aggregation.
- **Aggregation** means performing a calculation on a group of records that returns a single value for the whole group.
- This is done by the special aggregating functions that are used in SELECT-list.

- To get the number of cars, you need to use the `count` function:

```
car_portal=> SELECT   make, model, count(*)
car_portal-> FROM     car_portal_app.car a INNER JOIN car_portal_app.car_model b
car_portal->          ON a.car_model_id=b.car_model_id
car_portal-> GROUP BY make, model;
```

```
    make      |    model      | count
--------------+---------------+-------
 Citroen      | C3            |     2
 Toyota       | Prius         |     2
 Nissan       | Patrol        |     1
 Citroen      | C4 Picasso    |     3
 KIA          | Sportage      |     3
 Nissan       | GT-R          |     3
 Volvo        | XC70          |     4
 Toyota       | Yaris         |     1
 Toyota       | Land Cruiser  |     3
 BMW          | X5            |     4
```

```
 Lincoln      | Towncar       |     3
 Citroen      | C5            |     1
 Ferrari      | 458 Spider    |     4
 GMC          | Yukon         |     3
 Skoda        | Superb        |     2
 Infiniti     | QX4           |     2
 Volkswagen   | Tuareg        |     4
 Volkswagen   | Phaeton       |     2
(86 rows)
```

- There are several aggregating functions available in PostgreSQL.
- The most-frequently used are `count`, `sum`, `max`, `min`, and `avg` to compute, respectively, the number of records in a group, the total sum of a numeric expression for all the records in a group, to find the biggest and the smallest value, and to calculate the average value of an expression.
- There are some other aggregating functions, such as `corr`, which computes the correlation coefficient of the two given arguments, `stddev` for standard deviation, and `string_agg`, which concatenates the string values of an expression.

- When grouping and aggregation is used, records are grouped.
- This means that several records become one.
- Therefore, no other expressions except the aggregation functions and expressions from the `GROUP BY` list can be included in the SELECT-list.
- If this is done, the database will raise an error:

```
car_portal=> SELECT   a_int, a_text
car_portal-> FROM     car_portal_app.a
car_portal-> GROUP BY a_int;
ERROR:  column "a.a_text" must appear in the GROUP BY clause or be used in an aggregate function
LINE 1: SELECT   a_int, a_text
                        ^
```

- It's possible to create new expressions based on the expressions from the `GROUP BY` list.
- For example, if we have `GROUP BY a, b`, it's possible to use `SELECT a + b`.

- What if we need to group all of the records of the table together, not on the basis of the values of some field, but the whole table?
- To do this, you should include aggregating functions in the SELECT-list (and only them!) and not use the `GROUP BY` clause:

```
car_portal=> SELECT count(*)
car_portal-> FROM   car_portal_app.car;
 count
-------
   233
(1 row)
```

- Note that the SQL queries that have aggregating functions in the SELECT-list and don't have the `GROUP BY` clause always return exactly one row, even if there are no rows in the input tables, or if all of them are filtered out:

```
car_portal=> SELECT count(*)
car_portal-> FROM   car_portal_app.car
car_portal-> WHERE  number_of_doors = 15;
 count
-------
     0
(1 row)
```

- It's possible to count the number of unique values of the expression with `count(DISTINCT <expression>)`:

```
car_portal=> SELECT count(*), count(DISTINCT car_model_id)
car_portal-> FROM   car_portal_app.car;
 count | count
-------+-------
   233 |    86
(1 row)
```

The HAVING Clause

- Aggregating functions are not allowed in the `WHERE` clause, but it's possible to filter groups that follow a certain condition.
- This is different from filtering in the `WHERE` clause, because `WHERE` filters input rows, and groups are calculated afterward.

- The filtering of groups is done in the `HAVING` clause.
- This is very similar to the `WHERE` clause, but only aggregating functions are allowed there.
- The `HAVING` clause is specified after the `GROUP BY` clause.
- Suppose you need to know which models have more than `5` cars entered in the system.
- This can be done using a subquery:

```
car_portal=> SELECT make, model
car_portal-> FROM   (SELECT   make, model, count(*) c
car_portal(>         FROM     car_portal_app.car a INNER JOIN car_portal_app.car_model b
car_portal(>                  ON a.car_model_id = b.car_model_id
car_portal(>         GROUP BY make, model) subq
car_portal-> WHERE  c > 5;
  make   | model
---------+-------
 Opel    | Corsa
 Audi    | A2
 Peugeot | 208
(3 rows)
```

- A simpler and clearer way is to do it with a `HAVING` clause:

```
car_portal=> SELECT   make, model
car_portal-> FROM     car_portal_app.car a    INNER JOIN car_portal_app.car_model b
car_portal->          ON a.car_model_id=b.car_model_id
car_portal-> GROUP BY make, model
car_portal-> HAVING   count(*) > 5;
  make   | model
---------+-------
 Opel    | Corsa
 Audi    | A2
 Peugeot | 208
(3 rows)
```