# Querying Data with SELECT Statements

Part 4

# Ordering and Limiting Results

- The results of a query are not ordered by default.
- The order of the rows is not defined and may depend on their physical location on disk, the joining algorithm or on other factors.
- In many cases, it's required to sort the result set.
- This is done with the `ORDER BY` clause.
- The list of expressions whose values should be sorted is specified after the `ORDER BY` keyword.
- At the beginning, the records are sorted on the basis of the first expression of the `ORDER BY` list.
- If some rows have the same value for the first expression, they are sorted by the values of the second expression, and so on.

- For each item of the `ORDER BY` list, it's possible to specify whether the order should be ascending or descending.
- This is done by specifying the `ASC` or `DESC` keywords after the expression.
- Ascending is the default.
- `NULL` values are considered greater than any other values by default, but it's possible to explicitly define that `NULLS` should precede other rows by specifying `NULLS FIRST`, or `NULLS LAST` if `NULLS` should be at the end.

- It's not necessary for the `ORDER BY` clause to contain the same expressions as the SELECT-list, but it usually does.
- So, to make it more convenient, in the `ORDER BY` list, it's possible to use the output column names that are assigned to the expression in the SELECT-list instead of fully-qualified expressions.
- It's also possible to use the numbers of the columns.

- So, these examples are equivalent:
  - #Example 1:
```
SELECT   number_of_owners, manufacture_year,
         trunc(mileage/1000) as kmiles
FROM     car_portal_app.car
ORDER BY number_of_owners, manufacture_year,
         trunc(mileage/1000) DESC;
```
  - #Example 2:
```
SELECT   number_of_owners, manufacture_year,
         trunc(mileage/1000) as kmiles
FROM     car_portal_app.car
ORDER BY number_of_owners, manufacture_year, kmiles DESC;
```
  - #Example 3:
```
SELECT   number_of_owners, manufacture_year,
         trunc(mileage/1000) as kmiles
FROM     car_portal_app.car
ORDER BY 1, 2, 3 DESC;
```

- Sometimes, it's necessary to limit the output of a query to a certain number of rows and discard the rest.
- This is done by specifying that number after the `LIMIT` keyword:

```
car_portal=> SELECT *
car_portal-> FROM   car_portal_app.car_model
car_portal-> LIMIT  5;
 car_model_id | make | model
--------------+------+-------
            1 | Audi | A1
            2 | Audi | A2
            3 | Audi | A3
            4 | Audi | A4
            5 | Audi | A5
(5 rows)
```

- Another similar task is to skip several rows at the beginning of the output.
- This is done using the `OFFSET` keyword.
- The `OFFSET` and `LIMIT` keywords can be used together:

```
car_portal=> SELECT *
car_portal-> FROM   car_portal_app.car_model
car_portal-> OFFSET 5
car_portal-> LIMIT  5;
 car_model_id | make | model
--------------+------+-------
            6 | Audi | A6
            7 | Audi | A8
            8 | BMW  | 1er
            9 | BMW  | 3er
           10 | BMW  | 5er
(5 rows)
```

- The typical use case for `OFFSET` and `LIMIT` is the implementation of paginated output in web applications.
- For example, if 10 rows are displayed on a page, then on the third page, rows 21-30 should be shown.
- Then, the `OFFSET 20 LIMIT 10` construct is used.
- In most cases of using `OFFSET` and `LIMIT`, the rows should be ordered, otherwise, it's not clear which records are returned.
- The keywords are then specified after the `ORDER BY` clause.

- Note that using the `OFFSET` keyword for pagination with big tables has a performance drawback.
- Simply speaking, to skip the first X rows and return the others, PostgreSQL would need to read those X rows from disk, which might take a lot of time.

# Subqueries

- **Subqueries** are a very powerful feature of SQL.
- They can be used almost everywhere in queries.
- The most obvious way to use subqueries is in a `FROM` clause, as a source for the main query:

```
car_portal=> SELECT *
car_portal-> FROM   (SELECT   car_model_id, count(*) c
car_portal(>         FROM     car_portal_app.car
car_portal(>         GROUP BY car_model_id) subq
car_portal-> WHERE  c = 1;
 car_model_id | c
--------------+---
           56 | 1
            8 | 1
           80 | 1
           10 | 1
           86 | 1
           39 | 1
           66 | 1
           22 | 1
           59 | 1
           13 | 1
           49 | 1
           28 | 1
           33 | 1
            5 | 1
(14 rows)
```

- Subqueries are often used in SQL conditions in `IN` expressions:

```
car_portal=> SELECT car_id, registration_number
car_portal-> FROM   car_portal_app.car
car_portal-> WHERE  car_model_id IN (SELECT car_model_id
car_portal(>                         FROM   car_portal_app.car_model
car_portal(>                         WHERE  make='Peugeot');
 car_id | registration_number
--------+--------------------
      1 | MUWH4675
     14 | MTZC8798
     18 | VFZF9207
     19 | HPMA4871
     30 | YXMR3726
     58 | WYEW3260
     60 | TBHB8051
     97 | UPFC8027
    115 | RHUD9051
    123 | XVRF6824
    167 | FVSX9849
    168 | DAFM7215
    170 | ZZVH6530
    178 | ESKD3114
    193 | JKBE2220
    194 | AHDT2547
    208 | ZBSS1709
    215 | OHWQ2102
(18 rows)
```

- Scalar subqueries can be used everywhere in expressions—in the SELECT-list, the `WHERE` clause, and the `GROUP BY` clause, for example. Even in `LIMIT`:

```
car_portal=> SELECT (SELECT count(*) FROM car_portal_app.car_model)
car_portal-> FROM   car_portal_app.car
car_portal-> LIMIT  (SELECT MIN(car_id)+2 FROM car_portal_app.car);
 count
-------
    99
    99
    99
(3 rows)
```

- This is a PostgreSQL-specific feature.
- Not every RDBMS supports subqueries in every place where an expression is allowed.

- It isn't possible to refer to the internal elements of one subquery from inside another.
- However, subqueries can refer to the elements of the main query.

```
car_portal=> SELECT   make, model, (SELECT count(*)
car_portal(>                        FROM   car_portal_app.car
car_portal(>                        WHERE  car_model_id = main.car_model_id)
car_portal-> FROM     car_portal_app.car_model AS main
car_portal-> ORDER BY 3 DESC
car_portal-> LIMIT    5;
  make   |  model   | count
---------+----------+-------
 Audi    | A2       |     7
 Peugeot | 208      |     7
 Opel    | Corsa    |     6
 Jeep    | Wrangler |     5
 Peugeot | 407      |     5
(5 rows)
```

- Subqueries can be nested.
- This means it's possible to use subqueries inside another subquery.

# Set Operations – UNION, EXCEPT, and INTERSECT

- There are three set operations:
  - **UNION**: Appends the result of one query to the result of another query.
  - **INTERSECT**: Returns the records that exist in the results of both queries, effectively performing an `INNER JOIN` operation.
  - **EXCEPT**: Returns the records from the first query that don't exist in the result of the second query—the difference.

- The syntax of the set operations is as follows:
  - `#Syntax for UNION operator:`
    `<query1> UNION <query2>;`
  - `#Syntax for INTERSECT operator:`
    `<query1> INTERSECT <query2>;`
  - `#Syntax for EXCEPT operator:`
    `<query1> EXCEPT <query2>;`

- It's possible to use several set operations in one statement:
  - ```
    SELECT a, b FROM t1
    UNION
    SELECT c, d FROM t2
    INTERSECT
    SELECT e, f FROM t3;
    ```
    ```

- The priority of all set operations is the same.
- This means that, logically, they are executed in the same order as is used in the code.
- However, the order in which the records are returned is not predicted, unless the `ORDER BY` clause is used.
- In this case, the `ORDER BY` clause is applied after all of the set operations.
- For this reason, it doesn't make sense to put `ORDER BY` in the subqueries.

- All set operations, by default, remove duplicated records as if `SELECT DISTINCT` is used.
- To avoid this and return all the records, the `ALL` keyword should be used, which is specified after the name of the set operation:
  - `<query1> UNION ALL <query2>`

- Set operations can be used to determine the difference between two tables:

```
car_portal=> SELECT 'a', *
car_portal-> FROM   (SELECT * FROM car_portal_app.a
car_portal(>         EXCEPT ALL
car_portal(>         SELECT * FROM car_portal_app.b) v1
car_portal-> UNION ALL
car_portal-> SELECT 'b', *
car_portal-> FROM   (SELECT * FROM car_portal_app.b
car_portal(>         EXCEPT ALL
car_portal(>         SELECT * FROM car_portal_app.a) v2;
 ?column? | a_int | a_text
----------+-------+--------
 a        |     1 | one
 b        |     4 | four
(2 rows)
```

- It's possible to append one set of records to another only when they have the same number of columns and they have, respectively, the same data types, or compatible data types.
- The output names for the columns are always taken from the first subquery, even if they are different in subsequent queries.

# Dealing with NULLS

- `NULL` is a special value that any field or expression can have, except for the fields when it's explicitly forbidden.
- `NULL` means the absence of any value.
- It can also be treated as an unknown value in some cases.
- In relation to logical values, `NULL` is neither true nor false.
- Working with `NULL` can be confusing, because almost all operators, when taking `NULL` as an argument, return `NULL`.
- If you try to compare some values and one of them is `NULL`, the result will also be `NULL`, which is not true.

- For example, consider the following condition:
  - `WHERE a > b`
- This will return `NULL` if `a` or `b` have a `NULL` value.
- This is expected, but for the following condition, this is not obvious:
  - `WHERE a = b`
- Here, even if both `a` and `b` have a value of `NULL`, the result will still be `NULL`.
- The = operator always returns `NULL` if any of the arguments is `NULL`.
- Similarly, the following will also be evaluated as `NULL`, even if `a` has a `NULL` value:
  - `WHERE a = NULL`
- To check the expression for a `NULL` value, a special predicate is used: `IS NULL`.

- In the previous examples, when it's necessary to find records when `a = b` or both `a` and `b` are `NULL`, the condition should be changed this way:
  - `WHERE a = b OR (a IS NULL AND b IS NULL)`

- There is a special construct that can be used to check the equivalence of expressions taking `NULL` into account: `IS NOT DISTINCT FROM`.
- The preceding example can be implemented in the following way, which has the same logic:
  - `WHERE a IS NOT DISTINCT FROM b`

- The `AND` operator always returns false when any of the operands is false, even if the second is `NULL`.
- `OR` always returns true if one of the arguments is true.
- In all other cases, the result is unknown, therefore `NULL`.

```
car_portal=> \pset null '(null)'
Null display is "(null)".
car_portal=> SELECT true AND NULL "true AND NULL", false AND NULL "false AND NULL",
car_portal->        true OR NULL "true OR NULL", false OR NULL "false OR NULL",
car_portal->        NOT NULL "NOT NULL";
 true AND NULL | false AND NULL | true OR NULL | false OR NULL | NOT NULL
---------------+----------------+--------------+---------------+----------
 (null)        | f              | t            | (null)        | (null)
(1 row)
```

- The `IN` subquery expression deals with `NULL` in a way that might not seem obvious:

```
car_portal=> SELECT 1 IN (1, NULL) as in;
 in
----
 t
(1 row)

car_portal=> SELECT 2 IN (1, NULL) as in;
   in
--------
 (null)
(1 row)
```

- When evaluating the `IN` expression and the value that is being checked doesn't appear in the list of values inside `IN` (or in the result of a subquery) but there is a `NULL` value, the result will be also `NULL`, not false.
- This is easy to understand if we treat the `NULL` value as unknown, just as logical operators do.

- Functions can treat `NULL` values differently.
- Their behavior is determined by their code.
- Most built-in functions return `NULL` if any of the arguments are `NULL`.

- Aggregating functions work with `NULL` values in a different way.
- They work with many rows and therefore many values.
- In general, they ignore `NULL`.
- `sum` calculates the total of all non-null values and ignores `NULL`.
- `sum` returns `NULL` only when all the received values are `NULL`.
- For `avg`, `max`, and `min`, it's the same.
- But for `count`, it's different.
- `count` returns the number of non-null values.
- So, if all the values are `NULL`, `count` returns `0`.

- The `count` function can be used with a start (`*`) argument, like this: `count(*)`.
- This would make it count rows themselves, not a particular field or expression.
- In that case, it doesn't matter whether there are `NULL` values, the number of records will be returned.

- In contrast to other databases, in PostgreSQL, an empty string is not `NULL`.
- Consider the following example:

```
car_portal=> SELECT a IS NULL, b IS NULL, a = b
car_portal-> FROM   (SELECT ''::text a, NULL::text b) v;
 ?column? | ?column? | ?column?
----------+----------+----------
 f        | t        | (null)
(1 row)
```

- There are a couple of functions designed to deal with NULL values: COALESCE and NULLIF.

- The `COALESCE` function takes any number of arguments of the same data type or compatible types.
- It returns the value of the first of its arguments that `IS NOT NULL`.
- The following two expressions are equivalent:
  - `#Expression 1:`
    ```
    COALESCE(a, b, c)
    ```
  - `#Expression 2:`
    ```
    CASE WHEN a IS NOT NULL THEN a
         WHEN b IS NOT NULL THEN b
         ELSE c
    END
    ```

- `NULLIF` takes two arguments and returns `NULL` if they are equal.
- Otherwise, it returns the value of the first argument.
- This is somehow the opposite of `COALESCE`.
- The following expressions are equivalent:
  - `#Expression 1:`
    ```
    NULLIF (a, b)
    ```
  - `#Expression 2:`
    ```
    CASE WHEN a = b THEN NULL
         ELSE a
    END
    ```

- Another aspect of `NULL` values is that they are ignored by unique constraints.
- This means that if a field of a table is defined as `UNIQUE`, it's still possible to create several records with a `NULL` value in that field.
- Additionally, B-tree indexes, which are commonly used, do not index `NULL` values.
- Consider the following query:
  ```
  SELECT * FROM t WHERE a IS NULL;
  ```
  The preceding query would not use an index on the a column if it existed.