Published in DP6 US

DP6 Team Follow
Jul 27 · 11 min read · Listen

7 Advanced SQL Concepts You Need to Know!

Whether you are just entering the world of data or have been here for a long time, at some point you **will** come across SQL. This language (which has been around for almost 50 years!) has stood the test of time and is the most versatile language for working with data.

As a declarative language (where you have to worry about **what** you want to query, not **how** the query is carried out), the learning curve has a very steep beginning. In other words, you can absorb the basics of the language in a relatively short time. However, once you get past that stage, the concepts become more complex and increasingly depend on mastering the basics.

Note: Each system has its own version of SQL, adding or removing features or changing syntax. However, the concepts shown here are valid for any system.

1. Order of Execution of Select Commands

Let's simulate a situation where we have a table of "students" from which we need to extract the number of records grouped by the first letter of the "name" column. We write our query, placing a "FirstLetter" alias in the column with the first letter of the name, and use that same alias in the GROUP BY command. We run the query and get this result:

```
1   SELECT SUBSTR(name, 1, 1) AS FirstLetter
2         ,COUNT(1) AS Count
3   FROM students
4   GROUP BY FirstLetter;
```

gistfile1.txt hosted with ❤ by GitHub view raw

> Error: "FirstLetter" invalid identifier

At first it may seem strange, as we used the right formula, named the column correctly, and used the correct reference within the GROUP BY. So, we try to replace the GROUP BY argument with the function used in SELECT and we get this result:

1 SELECT SUBSTR(name, 1, 1) AS FirstLetter

2 ,COUNT(1) AS Count

3 FROM students

4 GROUP BY SUBSTR(name, 1, 1);

FirstLetter	Count
G	2

referencing the function inside the GROUP BY, why did we get an error when using the alias?

What happens here is that we are led to believe that the order in which we are writing the command is also the order in which the command will be processed by the Database Management System (DBMS). In practice, however, query processing follows a different order, as shown in the list below:

1. FROM (including Join operations)
2. WHERE
3. GROUP BY
4. HAVING
5. SELECT
6. DISTINCT
7. UNION
8. ORDER BY
9. LIMIT/TOP

As we can see, the first step is the preparation of the data source, joining all the tables in the **FROM** clause. This is followed by the application of the **WHERE** filter, grouping via **GROUP BY**, the grouping filter using **HAVING**, and, only then, the selection of columns by **SELECT**. Because of this, trying to use aliases in the **GROUP BY** clause would be the equivalent of trying to use a variable before it is declared in any other programming language, as aliases will only be recognized after the **SELECT** is processed.

Knowing the order of the clauses in the Select command is essential in order to speed up the debugging of possible query problems, to guide the structuring of queries and, above all, to identify possible optimization points in SQL codes.




2. Subqueries

Subqueries are SELECT statements executed within other SQL statements. These helper commands simplify the process of writing queries that would otherwise be too complex or even impossible to write.

It is possible to use subqueries in 3 different parts within a query:

1. In the data source of a main query

In this case, the subquery is used as a data source for the external query. This practice is often used when we want to use a subset (for example, a filtered table) as a data source for the main query.

```
SELECT t1.surname,
       COUNT(1) as CountStudents
FROM (
        SELECT surname
        FROM students
        WHERE name = "Gian"
    ) t1
GROUP BY t1.surname;
```

gistfile1.txt hosted with ❤ by GitHub view raw

In the code above, we have an inner query, which obtains a list of surnames from the “students” table, and an outer query, which uses this list to count students by surname. First, the internal command is executed.

```
SELECT surname
FROM students
WHERE name = "Gian";
```

gistfile1.txt hosted with ❤ by GitHub view raw

The result is considered a new table, which will be used inside the external command.

```
SELECT t1.surname,
```

This application of subqueries has an important implication for SQL as a whole: **every SELECT statement executed generates a new dataset that**, in the eyes of the DBMS, **is equivalent to a new table**. Therefore, you can nest SELECT statements inside SELECT statements inside other SELECT statements, and so on. The only limitation in this case is the readability of the code, which tends to get more complex with each level added. Below is an example of a query with 2 levels of subqueries.

```
SELECT t2.surname, t2.CountStudents
FROM    (
      -- Level 2
    SELECT t1.surname,
            COUNT(1) as CountStudents
    FROM (
            -- Level 1
            SELECT surname
            FROM students
            WHERE name = 'Gian'
            -- End Level 1
              ) t1
    GROUP BY t1.surname
    -- End level 2
    ) t2
WHERE t2.surname LIKE 'A%';
```

gistfile1.txt hosted with ❤ by GitHub view raw

2. Inside the list of columns chosen in SELECT

In this second case, we use SELECT statements within the column list of another SELECT statement. This feature is useful when we need to calculate metrics that depend on other tables, but without the requirement of joining both tables.

In the code above, a subquery was used to calculate the average grades that each student obtained, with another subquery calculating the students' attendance. It is important to note that even though the join between the tables is not made, it is necessary to define the relationship between them within the WHERE clause of the subquery. As in the previous case, the subquery is enclosed in parentheses.

3. As an external query filter

You can also use SELECT commands as filters for other queries. In this case, the secondary query result is interpreted as a single value or a list of values (depending on what the query returns). The result of this query can be used inside the WHERE clause like any other filter expression.

For example, in the code above the subquery returns a list of missing students' names. This list is used as a filter for the main query, which will only return information for students who have already been absent.

One important thing to note about subqueries is that in most cases it is possible to reach the same result by performing joins between the main and secondary query tables. However, as the main query does not directly use any column of the secondary query, using subqueries is more beneficial, as table join operations are quite costly in terms of processing.

3. Common Table Expressions (CTEs)

The biggest problem with subqueries, as we have already mentioned, is that they tend to make it difficult to read queries. As a result, they can impair our understanding of the query steps. To solve this problem, we can use the **Common Table Expressions** (CTEs) feature.

CTEs allow you to separate and encapsulate each of the steps of a complex query into sets, which can be referenced as if they were tables. The syntax is as follows:




Using the first query of the subqueries topic as an example, we can rewrite it like this:

With programming languages, it is always good practice to give self-explanatory names for variables and functions, and this is also true for CTEs. In the case above, we know exactly what to expect from the CTE, which is a list of students whose name equals Gian. We can put the column name after the CTE name, but it's not mandatory.

You can create several CTEs in the same query, as in the example below:

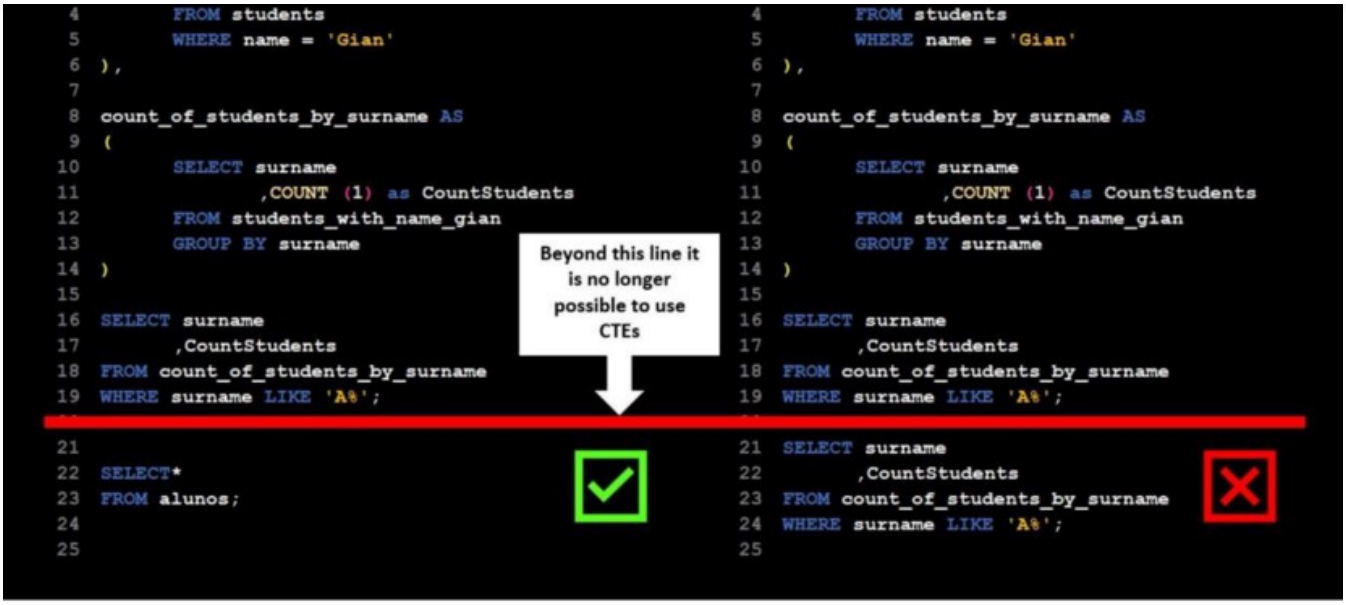



The result of this query is exactly the same as what would be obtained if we used subqueries within subqueries. However, understanding the query flow is much simpler by separating it into CTEs: first the “students” table is filtered, then a count of students is made by last name and, finally, all the last names starting with the letter a.

One important feature of CTEs is that they are scoped to the current query. This means that in the block where the CTEs are defined, you can refer to any other CTE that has already been defined. However, once you pass the first query that follows this block, it is no longer possible to query previously defined CTEs.

For example, in the code below, all CTEs defined above the red line can be referenced at any time. This is no longer possible below the red line, as the scope of the CTEs has already been finalized. Note that this finalization occurs regardless of whether or not the CTEs are used in the query after them.



This is a very powerful feature so you should simplify your queries using CTEs whenever possible!

4. User Defined Functions (UDFs): Scalar Functions

As with programming languages, you can encapsulate logic in functions in SQL, which receive parameters and return a value or set of values. The first type of function we will see is the scalar function, which takes parameters and returns only one value. Syntax between DBMSs can vary greatly in defining functions, but conceptually they work the same way. Below is the syntax for **Google BigQuery**.

Functions become extremely useful when we have a business need that is not included as a standard function in the DBMS we are working on. For example, let's say we repeatedly calculate the attendance frequency and grade point average of students. This calculation can be done via subqueries, as we saw in the subqueries topic. Alternatively, to avoid having to memorize the logic and reproduce it many times, we can create functions like these:




With the functions defined, we can call them only when we need the information, without repeating the same calculations within the executed queries. Calling user-defined functions is the same as any other standard SQL function where we pass the table columns that will make up its parameters.




instead of a single value. Table functions can be viewed as **parameterizable views**. Below is the syntax for **Google BigQuery**.

Unlike CTEs, table functions can be used in the context of the database, not just in the context of the query. For example, imagine that we frequently work with the table to count students by last name, as we did in the CTE example, but instead of fixing the student's name and the first letter of the last name we need to leave the values dynamic. In this case we can use table functions, reusing the same code

With the function created, we can use it as if it were any table, but with parameters, as shown below:




6. Window Functions

In SQL there are several different types of functions. The most common are scalar functions, which take a single value and return a single value (such as CONCAT, UPPER, LOWER), and aggregate functions, which take a set of values and return a single value (such as MAX, MIN, AVG).

There is also a third type of function, which is lesser known but equally important and versatile: window functions. These functions are executed over a set of rows (a window of values), and the values are calculated in a row context, accessing the previous or following rows for the calculation.

To understand this concept better, let's look at the RANK function.




name	grade	rank_grade
Gian	10	1
Gabriel	9	2
Gian	8	3
Gian	7	4
Gabriel	5	5
Gabriel	3	6

The RANK function went through all the records in the “exams” table and ranked each of the rows in relation to the **grade** column, with the highest grade designated as **rank 1** and the lowest as **rank 6**. Note that the rank calculation depends not only on the data of the row itself, but also on the comparison between the rows. It is precisely in this context that window functions operate. They access previous and subsequent rows to calculate the function value for the current row.




rule for the evaluation of the rows is defined in the **OVER** clause, through the **ORDER BY**.

In our example, the **ORDER BY** in the **OVER** clause caused each row to be evaluated in ascending order of grades and returning the row's position in the ranking.

An important detail is that the order in which the window function is calculated depends exclusively on the OVER clause, which is not affected by the order of the query in which it is used, as in the example below:




Gabriel	9	2
Gabriel	5	5
Gabriel	3	6
Gian	10	1
Gian	8	3
Gian	7	4

In the result above, we see that the table is organized by **name** and **grade**, but RANK continues to bring the **same result as the previous query**.

The second argument of window functions is the **PARTITION BY**, which defines data groups where the function will be evaluated. This clause breaks the table into groups of rows with some characteristic in common (this characteristic being the values in the columns defined in the **PARTITION BY**) and locally evaluates the result of the window function used. Once all these groups have been evaluated, they are joined again into a single dataset, which is the result of the query.




name	grade	rank_grade
Gabriel	9	1
Gabriel	5	2
Gabriel	3	3
Gian	10	1
Gian	8	2
Gian	7	3

In the query above, a partition was made by the **name** column, and the query result was the **classification of grades in relation to people's names.**

There are functions that are purely of the window type (like RANK, DENSE_RANK, ROW_NUMBER) and aggregate functions that can be calculated inside windows by adding the OVER clause. If you are interested in window functions, I recommend looking for articles about solutions to the "gaps and islands" problem using SQL.

7. Temporary Tables

The final concept that we are going to deal with is that of temporary tables. As the