Analytic Functions

Section Recap

In this EXTREMELY important section, you learned about "Analytic Functions".

I call it an EXTREMELY important section because analytic functions are VERY powerful, and can help you solve many types of problems in a simpler, and most of the times, more efficient way than using other, more "traditional", methods.

We saw that one of the differences between aggregate and analytic functions is that analytic functions return multiple rows per group, as opposed to aggregate ones, which return one row per group.

Another difference, which is actually a consequence of the previous one, is that we can include all of the columns we may want in the SELECT list without needing to include a GROUP BY clause with the same columns, as shown in this example:

```
SELECT e.*,
   COUNT( * ) OVER (PARTITION BY department_id) AS the_count
FROM employee e
ORDER BY department_id, id;
```

This returns all the columns from the employee table, and there is also a call to COUNT, and a GROUP BY clause is not needed.

And you can tell that it is an analytic function because of the "OVER" keyword that appears in the second line after the call to the function.

We also saw that we can call more than one analytic function in the same query, and we can have different levels of aggregation for each one, as in this example:

```
SELECT id,
   name,
   department_id,
   salary,
   salary - AVG(salary) OVER(PARTITION BY department_id) AS diff_dept_avg,
   salary - AVG(salary) OVER() as diff_total_avg
FROM employee e
ORDER BY department_id, id;
```

The part that comes after the OVER keyword is the analytic clause, which is inside parentheses.

For the first average I'm partitioning by department, so this will give me the average salary per department, and in the second one, since I don't have any PARTITION clause, it returns the average salary of the whole result set.

So, the partition clause works in a similar way that the GROUP BY clause.

Now, what else can we have in the analytic clause? Do you remember?

We can also have an ORDER BY clause, which defines not the order in which rows are going to be returned by the query, but the order that will be used to apply the analytic function.

So, for example, if I have a query like this one:

```
SELECT
   id,
   name,
   SUM(salary) OVER(ORDER BY id) AS sum_salary
FROM employee;
```

It means that the SUM function will be applied in order of employee id, so, for the first employee, the sum will be his own salary. For the second one, it will sum his salary and the salary of the previous employees if we order them by id, and so on.

And I can return the results in a different order if I want to, for example in alphabetical order of name:

```
SELECT
   id,
   name,
   salary,
   SUM(salary) OVER(ORDER BY id) AS sum_salary
FROM employee
ORDER BY name;
```

But the SUM would still be calculated in order of ID because these two ORDER BY clauses are completely independent of each other.

And the other thing we can have in the analytic clause is a WINDOWING clause, which allows

us to define the group of rows on which we want to apply the function. This group of rows is different from the partition, because the partition is static, but the window can move, grow or shrink inside the partition, as we process different rows.

We saw that when we add an ORDER BY analytic clause, we are implicitly adding a windowing clause, but if we don't want to use the default windowing clause, we can add our own.

Here is an example:

```
SELECT
   id,
   name,
   salary,
   department_id,
   SUM(salary) OVER(PARTITION BY department_id ORDER BY id ROWS 1 PRECEDING) AS
accumulated_rows
FROM employee
ORDER BY department_id, id;
```

The windowing clause, in this case, starts with the "ROWS" keyword, which is one of 2 possibilities:

When we use ROWS, it means we are defining the window in terms of physical numbers of rows, as in this example, where I defined a window that starts one row before the current one, and since I didn't define the end of the partition, it defaults to the current row.

And the other possibility is to use RANGE, instead of ROWS, in which case the window is defined as a range of values in relation to the column used in the ORDER BY analytic clause, so if in this example I use RANGE 5 PRECEDING, it would mean that the window will include rows whose id falls inside the range of the id of the current row minus 5 and the id of the current row.

If only one row falls inside this range, the function would be applied to that row alone.

There are 2 other important things about RANGE windowing:

1) RANGE is the default option that is used when Oracle adds a windowing clause implicitly, which happens when we add an order by analytic clause but don't add an explicit windowing clause.
2) We can only use RANGE if we are ordering by a column that is of a numeric or date data type.
Another thing we saw is that we can not only define the beginning of the window but also the

end of it, so, for example, I could say "ROWS BETWEEN 5 PRECEDING AND 2 FOLLOWING".

I want the window to start at the first row of the partition I use UNBOUNDED PRECEDING, and if I want it to end at the last row of the partition, I use UNBOUNDED FOLLOWING.

In this section, we saw also that the functions we have traditionally known as aggregate functions, such as SUM, COUNT, AVERAGE, etc, can also be used as analytic ones.

There is another very important thing you need to remember about analytic functions: They are computed AFTER the WHERE clause, GROUP BY, and HAVING clauses are evaluated, and thus, they cannot be used in any of those clauses. They can only be used in the SELECT list and in the ORDER BY clause.

If you need to filter your data based on the results of an analytic function, you have to apply the analytic function in a subquery, and then filter the results in the main query.

In this section, we revisited the RANKING FUNCTIONS, which were mentioned for the first time in the TOP-N queries lesson.

The most commonly used ranking functions are ROW_NUMBER, which assigns a different number to each row, RANK, which can assign the same rank to more than one row if they have the same value for the column used to sort, and DENSE_RANK, which works in a similar way than RANK, but without leaving holes in the rankings.

We saw that besides top-n queries, they can also be used to prioritize rows. There's a link in the resources section of this lesson to an article I wrote, which provides more details about this technique.

In this section, you also learned about the LISTAGG function, which, like other functions we have seen, can be used as an aggregate or an analytic function.

This function is used to take the values of a column from different rows, and return them concatenated in a single string, like in this aggregate example:

```
SELECT
  department_id,
  LISTAGG(name, ', ') WITHIN GROUP (ORDER BY name) emp_list
FROM employee
GROUP BY department_id;
```

With the first parameter, I define the column whose values I want to concatenate, and in the second parameter, I define the string I want to use as separator between those values. In this case, I'm concatenating the names, and I want them separated by a comma and a blank space.

And after the words "WITHIN GROUP", I have to write an ORDER BY clause, inside parentheses.

Like with other functions, if you want to make it analytic, you just have to add the ANALYTIC clause. Also remember that when it is analytic, the query no longer requires a GROUP BY clause.

We also covered in this section the LAG and LEAD functions, which are pretty awesome because they let you read data from a row that is not the current one, without needing to do a self join. LAG gives you access to a previous row, and LEAD gives you access to the following row, based on a given order.

Here is the example I used to explain how they work:

```
SELECT
   name,
   department_id,
   salary,
   LAG(salary) OVER(PARTITION BY department_id ORDER BY salary) AS previous_salary,
   LEAD(salary) OVER(PARTITION BY department_id ORDER BY salary) AS following_salary
FROM employee
ORDER BY department_id, salary;
```

Here LAG was used to access the SALARY of the previous row, and because I'm partitioning by department, I can only access previous rows from the same department. To define which one is the previous row, I'm ordering them by salary, which is coincidently the same column I used as the argument for LAG, but it could be any other column or expression. And LEAD works in the exact same way, but it gives us access to following rows instead.

We saw also that by default these functions access ONE row before and ONE row after the current one, respectively, but this offset, which by default is 1, can be modified by passing a second parameter to the functions with the offset we want.

As we saw, when there are not enough rows to satisfy the offset BEFORE or AFTER the current row, these functions return NULL, but we can pass a third parameter, in which we can define what we want them to return in those cases.

For example, If I want them to return 0 when the offset moves out of the partition, I can simply do this:

```
SELECT
   name,
   department_id,
   salary,
   LAG(salary,1,0) OVER(PARTITION BY department_id ORDER BY salary) AS previous_salary
FROM employee
ORDER BY department_id, salary;
```

Notice that to provide a default value, you have to provide the second parameter (offset), even if you want the default offset (1).

And I mentioned that these functions offer something to decide how to handle NULLs when they are not produced by the offset going out of the partition, but because the related row exists in the partition but the column you are looking at has NULL, and I asked you to look it up in the documentation (whose link was provided in the lesson)…

DID YOU DO IT?

If you didn't, you should do it now ;)

The next 2 functions we covered here were FIRST and LAST.

When you need a value from the first or last row of a sorted group, but the needed value is not the sort key, the FIRST and LAST functions eliminate the need for self-joins, subqueries, or views and enable better performance.

This was the first example we ran:

```
SELECT MAX(hire_date) KEEP (DENSE_RANK LAST ORDER BY id)
FROM employee;
```

In this case, we are sorting the rows by the ID column, and we are keeping the ones that Rank as LAST based on that order. This is done with the KEEP keyword and what we have inside these parentheses.

Then, we are getting the HIRE_DATE of those rows, by using this MAX here. In this case, I'm

using this MAX because it is mandatory to use an aggregate function here, but since we are ordering by ID, which is the primary key of the table, this will always keep only one row, so the MAX function here is not really doing anything, because it is always being applied to a single row.

Do you see the benefit?

We are getting the hire_date of the employee with the greatest id. Without this function, we would have to get the MAX(id) in a subquery, and then in the main query get the hire_date of that employee, which would make the query not only longer and more complex, but also less efficient.

And the FIRST function works exactly the same, but keeps the rows that rank as FIRST as opposed to LAST.

And finally, we covered the FIRST_VALUE and LAST_VALUE functions, which return the first or last value in an ordered set of values.

These functions are somehow similar to FIRST and LAST, but they don't need to be used in conjunction with aggregate functions.

This was one of the examples we saw:

```
SELECT e.*,
  FIRST_VALUE(bonus) OVER (PARTITION BY department_id ORDER BY salary) AS fv
FROM employee e;
```

What happens when this is executed is that the rows are divided into partitions by department id, then each partition is ordered by salary, and then the bonus from the FIRST row is returned.

And the LAST_VALUE function works exactly the same, but it returns the value from the last row, instead of the first one, so, FIRST_VALUE with an ascending order would produce the same result as LAST_VALUE with a descending order.

We also saw that we can add the IGNORE NULLS option before the analytic clause, to make them return the first or last NON NULL values.

And that was it for this section.

I will see you in the next one!

