CS992: Database Development – Lab 5

Explore SQL 'window'/analytic functions

This week we will continue to use the *Live SQL* platform to explore some 'window' functions in SQL, that you may not have come across to date. (These are also referred to in Oracle as "analytic" functions – referring to their primary purpose/use, in data analytics.)

You should already be familiar with some common aggregate functions, such as COUNT(), SUM(), AVG(), etc. and know that when using these you need to include a GROUP BY clause to specify at what level of grouping the aggregation is to be applied. However, once you have 'grouped' you can no longer view the values in the individual rows, even though it is often helpful to be able to see *both* outcomes in the same query. The "window" functions allow you to do this, by specifying a *window* into your data within which you can perform some operations against a set of rows, and perform those with respect to the current row. Rather than using GROUP BY, a new OVER() clause is used to determine the rows that will be included in the *window* within which you will be operating. There are a number of additional functions that can only be used with window/analytic functions; these include: RANK(), LAG(), LEAD(), etc. The creation of these 'sliding windows' allows you to compute running totals over groups of rows in a straightforward manner.

The combination of OVER() [and its associated PARTITION BY clause] and windowing functions, can make certain types of queries much less cumbersome, as well as improve their performance. However, it can take a little while to get your head around these as they diverge slightly from the strict set-at-a-time constructs in the rest of SQL's Data Manipulation Language (DML). The best way to understand the approach is to look at some examples.

At various points below, you will see "**SUBMIT**" entries; these are items for which you should provide solutions as part of your Lab 5 submission.

1. Try out some basic analytic functions in *Live SQL*

Log in to your *Live SQL* account and open the *SQL Worksheet* window. For these first few exercises we will use data on populations of countries over time, provided in Oracle's ***WORLD*** schema. You can take a look at the, slightly unusual, structure of the World_Population table by selecting the "Schema" option, and you will need to prefix this table name in any queries with Oracle's schema name, as the table does not exist in your own ("My Schema") context.

> Let's say we want to see the population of each country for a given year (say for 2012):
>
> ```
> SELECT country AS Name, "2012" AS Population_in_2012
> FROM world.world_population
> ORDER BY country;
> ```
>
> The only thing that might seem a bit 'odd' here is that we need to quote the second column reference. The is because each year is contained in the world_population table as a separate column / field, rather than there being, for example, some general "year" field that is also defined in the table. (If we don't quote the column name, Oracle will treat this as a literal numeric value.)

Now what about finding the total world population (for all countries present in this world data set) in 2012? Again, this is quite straightforward:

```
SELECT sum("2012") AS World_Population_in_2012
FROM world.world_population;
```

This number is a bit hard to read – we will get to improving the format later – but this seems about right, i.e. just over 7 billion people in 2012. (N.B. Above I said that you "need" to include a GROUP BY clause for aggregate functions. This is normally the case, but here we wish to aggregate over ALL the data and so this is not necessary.)

However, what if we wanted BOTH the country populations and the world population in the same output? Well using what we know about sub-queries we could try:

```
SELECT country AS cntry_name, "2012" AS cntry_population,
                sub_q.world_population
FROM world.world_population,
(
  SELECT SUM("2012") AS world_population
  FROM world.world_population
) sub_q
ORDER BY cntry_population DESC;
```

This works, but it seems a bit complex. This is where we could use the OVER clause with an aggregation. Here we use the "SUM" function and as we are going to do this over the whole world we do not need a condition in the OVER() clause. If we had, for example, the continents stored in a different column we could have specified "OVER (PARTITION BY Continent)" – but we will get to that later.

```
SELECT country AS cntry_name,
       "2012" AS cntry_population,
       SUM("2012") OVER() AS world_population
FROM world.world_population
ORDER BY country;
```

Above we used an aggregate function that you are already familiar with, and one that can also be used with the GROUP BY clause. However, there are also *window* functions that can only be used with the OVER clause. One such is RANK() which returns the ranking of a given row within the specified window. Try this:

```
SELECT country, "2012",
       RANK() OVER(order by "2012") AS Popl_rank_in_2012
FROM world.world_population
ORDER BY country;
```

We can now see that each country (row) is ranked according to its population, so that Afghanistan is rank 173, etc. It would seem that smaller countries have a low rank – for example American Samoa is rank 13.

***SUBMIT* 5.a** – Alter the query above to rank the countries from largest to smallest (as opposed to the 'default' ranking) in 2012. You should still use the alphabetic ordering to view the output, so when you re-run your modified query in the first output segment you will see that Afghanistan is the 43rd largest country, Algeria the 35th largest, etc.

We might also want to order our overall output by population rank (rather than by alphabetic country name) and also look at some more recent data (say from 2019):

```
SELECT country, indicator_name, "2019",
       RANK() OVER(order by "2019") AS Popl_rank_in_2019
FROM world.world_population
ORDER BY Popl_rank_in_2019;
```

OOPS! It would seem that ALL countries have the same rank… This is because the entries for recent years are (sadly!) all nulls. In fact, the most recent data are for 2012 (Oracle really should update this table!). Alter the "2019" back to "2012" and you should see the countries in population rank order, starting with little Tuvalu – with just 9,860 people…

However, we are mostly used to thinking about 'rank' in the opposite direction – i.e. we would expect that our #1 would be the largest country, here China. To do this we need to order in descending order:

```
SELECT country, indicator_name, "2012",
       RANK() OVER(order by "2012" DESC) AS Popl_rank_in_2012
FROM world.world_population
ORDER BY Popl_rank_in_2012;
```

Notice that we now get a new "Not classified" row as our Rank #1 country. This is because a null row exists in Oracle's data set for 2012. We didn't see it before because by default null rows come at the end of a list. However, now that we have used DESC for our order the row comes first!

At least this null row shows up with the value of "-" in our output. However, it is the case that when calculating averages, median, etc. (less so totals), we need to be careful to remove null rows as they can affect the results. In the present example, we can resolve this by filtering out null rows in the usual way:

```
SELECT country, indicator_name, "2012",
       RANK() OVER(order by "2012" DESC) AS Popl_rank_in_2012
FROM world.world_population
WHERE "2012" is not null
ORDER BY Popl_rank_in_2012;
```

***SUBMIT* 5.b** – Build a query that provides a list of countries which were in the 'top 20' in terms of population ranking by 2012, but had not been in that grouping half a century before (i.e. in 1962). You should find a list of 6 countries (Congo, Ethiopia, Iran, Philippines, Thailand and Turkey). There are various ways to tackle this question but perhaps creating two sets and then looking for the 'difference' (in terms of country membership) between these two sets is the most straightforward approach.

2. Look at rows 'close' to the current row using *window* functions

So far we have seen the OVER() clause and used some aggregate functions with it – one that we already knew (the 'generic' SUM) and one which is only available as a *window* function (RANK). In addition to being able to operate on specific rows, we can also manipulate the rows *preceding* or *succeeding* the current row – using LEAD and LAG.

Let's start by limiting our output to the top 15 countries (by population):

```
SELECT * FROM
(
SELECT country, "2012",
       RANK() OVER(order by "2012" desc) AS cntry_pop_rank
FROM world.world_population
Where "2012" is not null
ORDER BY cntry_pop_rank
) x
where cntry_pop_rank <16;
```

In many contexts we may wish to compare the difference between two ranked items in a list. For example, how much more revenue does my top selling product bring in compared to the second most popular product?

Before looking at that type of comparison, we should add a little bit of formatting to make the population figures easier to read:

```
SELECT * FROM
(
SELECT country, to_char("2012" ,'999G999G999G990') pop_2012,
       RANK() OVER(order by "2012" desc) AS pop_order
FROM world.world_population
Where "2012" is not null
ORDER BY pop_order
) x
where pop_order <16;
```

We can now look at rows relative to each other – i.e. within a given *window*. For example, if we wanted to see how much smaller a 'top 10' country's population was compared to its next largest neighbour we could use the **LEAD** function with a value of "1" – i.e. to look at the country one 'above':

```
SELECT * FROM
(
SELECT RANK() OVER(order by "2012" desc) AS pop_order,
       country,
       to_char("2012" ,'999G999G999G990') pop_2012,
       to_char(LEAD("2012", 1, 0) over (order by "2012") -
"2012",'999G999G999G999G999G990') difference
FROM world.world_population
Where "2012" is not null
ORDER BY pop_order
) x
where pop_order <11;
```

The only problem here is that China is compared to 0 - as it has no **LEAD (1)** row, being the highest ranked country. We can use the standard SQL 'decode' function

to specify that if a row has “pop_order” equal to “1” (i.e. here China) we will return a null value, otherwise the value returned for “Smaller_than”.

```
SELECT pop_order, country, pop_2012,
decode(pop_order,1,null, Smaller_than) AS Smaller_than_next_largest
FROM
(
SELECT RANK() OVER(order by "2012" desc) AS pop_order,
  country,
  to_char("2012" ,'999G999G999G990') pop_2012,
  to_char(LEAD("2012", 1, 0) over (order by "2012") -
"2012",'999G999G999G999G999G990') Smaller_than
FROM world.world_population
Where "2012" is not null
ORDER BY pop_order
) x
where pop_order <11;
```

***SUBMIT* 5.c** – Use the **LAG** function to see how much the 5 smallest countries in the world would need to grow in terms of population size to move ‘up’ the population rankings table (based on data from 2012).

3. Look at some additional tutorial material

You should now feel free to explore aspects of *window*/*analytic* functions that may be of interest to you. Within the *Live SQL* setting you will find the following tutorial -

> Enter “Analytic Functions” into the search bar at Home and then choose the **Analytic Functions: Databases for Developers** option.

This uses the ‘bricks’ sample data (that we came across in the initial sub-query session in an earlier lab), which I am not totally sold on, though the structure of the various examples is well organised.

You may also have noticed that Oracle provides a number of free on-line classes, under the label of its ***Developer Gym.*** A lot of useful course and exercises can be found under this framework - https://devgym.oracle.com/

> In the context of the current subject there is a course called “**Analytic SQL for Developers**”. As you will see, each of the six modules in this class may take around an hour to complete… These may be useful resources for future learning, but clearly, I do **not** expect you to spend 6 hours on this material!
>
> I would suggest that you find time to take a look at a couple of the initial videos in the ‘Introduction’ (Module 1 of the Analytic SQL for Developers). In particular, they will reinforce your understanding of the generic structure of *window* functions, i.e.

```
<function> (<arg1>, <arg2>, …)
OVER (
   <partition clause>
   <sorting clause>
   <windowing clause>
)
```

In the examples shown in some of these videos (and those shown in the tutorial noted at the start of this section), we omitted certain of these clauses; which is entirely legitimate – they are all optional – but looking at a few of the 'Introduction' videos/exercises will help to give a ***fuller picture as to what window/analytic functions*** can be used to achieve.

SUBMIT 5.d – This final part of the submission is a bit more 'open ended' than the questions above which require an SQL-based solution, but I hope that it will encourage you to explore the area of 'SQL for data analytics' in a little more detail. For this last part of the submission I would like your group to find (copy/paste) a piece of sample code that you come across (or maybe even write yourselves!) during your exploration of the various *window* functions. Ideally, this should be an example that you come across and think, "that is really useful / powerful / elegant" (or whatever). Your group should accompany the code with a little description (5-10 lines as comments in the text file) of what the query is doing and why/where you thought it could be useful.

The material outlined above (particularly bits of the *Dev Gym*) should provide ample scope to find such an example. However, as noted the material is quite extensive. Maybe each group member should agree to take 15 minutes to look at some different options (e.g. videos in the "Introduction" section of the *Dev Gym*) and then compare notes / decide on an example to include in your submission.

I have also provided a few additional resource links on the page below that it may be easier to 'dip into' or for those interested in following up on particular aspects of this subject area in more detail at some point in the future.

4. Additional resources on SQL *window* functions (within *Live SQL*)

As noted earlier, there are LOTS of scripts and examples that can be explored within *Live SQL*. The list below notes a few that I came across which go into more detail on the *window* functions we have been exploring in this lab or illustrate how *Oracle* supports data analytics and warehousing. (Entering the text shown below in **bold** into the search bar of the "Code Library" tab will bring up these various additional scripts/tutorials. In actual fact, quite a few of these are actually from Connor's *Dev Gym* series, this just lets you get to them a bit more directly than having to watch the videos, etc.

LAG and LEAD functions
A little bit of a more involved example of how to use the LEAD and LAG windowing functions to show enhanced ordering options within tables.

More complex ranking functions
In addition to just looking at the previous/next row in a ranked list, you sometimes want to look at groups of rows based on some overall ranking. This script has examples that demonstrate the PERCENT_RANK, CUME_DIST and NTILE functions.

Pivot and unpivot examples using Olympic data
These scripts generate a subset of the results from the Rio Olympics to use as an example as to how you can pivot/unpivot tables in Oracle.

Aggregating Data using Weighted Averages in Analytic Views
You often need to use weighted averages to correctly aggregate over different level in a hierarchy. This is something that statistics packages obviously support but they are important to bear in mind if you wish to execute some 'stats' within the context of query processing. (Unlike the 'script' examples above, this is a fuller and more detailed 'tutorial' set of exercises.)

Using the Analytic View Materialized Aggregate Cache
As mentioned in class *Materialized Views* are one of Oracle's approaches to implement Data Warehouses. If you are interested to learn more about this approach, the set of scripts provided in this tutorial provide an introduction.