EI1020/MT1020. Databases Academic Year 2020-21

Session 7. Aggregate Functions and Grouping

Document version: 2021-05-03

1 Objectives

- To be able to use the aggregate functions (`SUM`, `MAX`, `MIN`, `AVG`, `COUNT`, etc.) in the `SELECT` query.
- To be able to distinguish different uses of `COUNT` function.
- To be able to understand a simple `SELECT` query with a subquery that includes an aggregate function.
- To be able to decide when it is necessary to group the rows of the tables (`GROUP BY` clause) to answer a query and to use it correctly.
- To be able to decide when it is necessary to include restrictions (`HAVING` clause) after the rows of the tables have been grouped to answer a query and to use it correctly.
- To be able to use aggregate functions both in `HAVING` clause and in `SELECT` clause.

2 Some advice for this session

- Create a working directory called `s07` to store all the files created in this session inside your `ei1020` (or `mt1020`) directory.
- Save each SQL query in a different file with the **.sql** extension. Use a coherent naming system that can be employed in all sessions. For example, the **s07_ej01.sql** file will contain the exercise 1 of session 7.
- Before writing each query, you can check in the course website the results your query should obtain, so that you can find out if you have understood the problem statement correctly.

3 In the previous session ...

We saw how to write queries in which the `JOIN` clause is used to show data from different tables.

We also saw how to join tables in which rows are lost when an inner join is used (`JOIN`). If we do not want to lose these rows we have to use the `OUTER JOIN` clause.

4 In this session ...

We will practice how to apply aggregate operations taking into account either all the rows in a table (without the `WHERE` clause), or only some of them (with the `WHERE` clause).

We will see that the aggregate functions can also be applied to groups of rows obtained with the `GROUP BY` clause. This usage is necessary when the computations are to be applied not just once, to all the rows in a table (or a subset of them when there is a restriction), but several times, to the rows included in the different groups.

We will also see that it is possible to include restrictions on the groups of rows defined by the `GROUP BY` clause, by means of the `HAVING` clause.

4.1 Aggregate functions

In some cases, it is required to count data: *how many customers are there in Castellón?*, or to obtain calculated data: *how much is the VAT (IVA) paid in the invoice number 3752?*

SQL has a set of functions that can be used in `SELECT` clause and that operate on the column values. Using these functions, we can perform different calculations such as: adding values, obtaining the maximum value or the average value.

The most common aggregate functions are shown next:

`COUNT( * )`	Counts the numbers of rows.
`COUNT( column )`	Counts not NULL values.
`SUM( column )`	Sums the column values.
`MAX( column )`	Obtains the maximum value of the column.
`MIN( column )`	Obtains the minimum value of the column.
`AVG( column )`	Obtains the average value of the column.

The column specified in the `SUM` and `AVG` functions has to be of numeric datatype. The rest of functions have to be applied to the following datatypes: numeric, arrays or dates.

If there is no restriction (no `WHERE` clause) in the `SELECT` query where the aggregate functions are used, they will be applied to all table rows specified in `FROM` clause.

In case some restriction is specified in the `WHERE` clause, the aggregate functions will be only applied to the rows selected by the restriction.

Next, some examples are shown.

```
1  -- Average quantity per invoice line.
2  SELECT AVG( cant )
3  FROM   lineas_fac;
```

```
1  -- Average quantity per invoice line of article with code TLFXK2.
2  SELECT AVG( cant )
3  FROM   lineas_fac
4  WHERE  codart = 'TLFXK2';
```

```
1  -- Several calculations could be done at the same time.
```

```
SELECT SUM( cant ) AS suma, COUNT( * ) AS lineas
FROM   lineas_fac;
```

The `COUNT()` function makes different operations depending on its argument:

- `COUNT( * )`: Counts rows.
- `COUNT( column )`: Counts the number of non-null values in the column.
- `COUNT( DISTINCT column )`: Counts the number of different values in the column.

Look carefully at the rows on table `P`.

```
SELECT *
FROM   P;
```

```
 pnum |  pnombre   |   color    | peso |   ciudad
------+------------+------------+------+------------
 P1   | tuerca     | verde      |   12 | París
 P2   | perno      | rojo       |      | Londres
 P3   | birlo      | azul       |   17 | Roma
 P4   | birlo      | rojo       |   14 | Londres
 P5   | leva       |            |   12 | París
 P6   | engrane    | rojo       |   19 | París
(6 rows)
```

Try and take note of the result obtained when you execute the following query:

```
SELECT COUNT( * ) AS cuenta1, COUNT( color ) AS cuenta2,
       COUNT( DISTINCT color ) AS cuenta3
FROM   P;
```

```
cuenta1 =  ......................
cuenta2 =  ......................
cuenta3 =  ......................
```

Now, describe the results of the query with your own words:

- `cuenta1` returns the number of .....
- `cuenta2` returns the number of .....
- `cuenta3` returns the number of .....

The aggregate functions (`SUM`, `MAX`, `MIN`, `AVG`, etc.) ignore NULL values, that is, they are not taken into account in the computations.

Then, will the values of `media1`, `media2` and `media3` always match when executing the following query? .....

```
SELECT AVG( dto ) AS media1,
       SUM( dto ) / COUNT( * ) AS media2,
       SUM( dto ) / COUNT( dto ) AS media3
FROM   lineas_fac;
```

Why? .....

In which one of the three columns above have the null values been included in the computations as zero values?

.....

What is the meaning of `media1`?

.....

What is the meaning of `media2`?

.....

What is the meaning of `media3`?

.....

The `AVG` function calculates the average value. If the table in the `FROM` clause is `articulos`, the average value will be per article. Similarly, if the table in `FROM` clause is `facturas`, the average value will be per invoice.

In case we want to calculate another type of average, it will be necessary to do a quotient. For example, the average number of *invoices per month* during last year is obtained by dividing the number of invoices by 12 months:

```
SELECT COUNT( * ) / 12 AS media_mensual
FROM   facturas
WHERE  EXTRACT( year FROM fecha ) =
       EXTRACT( year FROM CURRENT_DATE ) - 1 ;
```

It is important to note that the `COUNT` function always returns an integer, and that the operations on integers also return an integer. That is, the operation `SELECT 2/4;` will return zero.

PostgreSQL allows to do datatype conversions using the `::` operator. For example:

```
SELECT COUNT( * )::NUMERIC / 12 AS media_mensual
FROM   facturas
WHERE  EXTRACT( year FROM fecha ) =
       EXTRACT( year FROM CURRENT_DATE ) - 1 ;
```

However, the above operator is specific to this DBMS[1]. An alternative is to multiply one of the operands by `1.0`. In this case it is needed to round the resulting value:

```
SELECT ROUND( COUNT( * )*1.0 / 12, 2 ) AS media_mensual
FROM   facturas
WHERE  EXTRACT( year FROM fecha ) =
       EXTRACT( year FROM CURRENT_DATE ) - 1;
```

4.2 The GROUP BY clause

The `GROUP BY` clause *makes groups* whose rows have the same values on the attributes which appear in the clause. The aggregate functions presented in the previous section can be applied on each group. When these functions are used in the `SELECT` clause, they will be applied several times, once for each group. The following query counts, for each customer, the number of invoices issued last year:

```
SELECT codcli, COUNT( * )
FROM   facturas
WHERE  EXTRACT( year FROM fecha ) =
       EXTRACT( year FROM CURRENT_DATE ) - 1
GROUP  BY codcli;
```

The query is carried out as follows.

- First, the `FACTURAS` table is chosen (using the `FROM` clause) and the rows which fulfill the restriction (in the `WHERE` clause) are selected.
- Then, the invoices are divided in groups, such that in a group there are only invoices of a customer (`GROUP BY codcli`), and there are as many groups as customers with invoices in the last year.
- Finally, the code of the customer and the number of invoices in the corresponding group (`COUNT(*)`) are shown (`SELECT` clause).

If you also want to show the name of the customer, previously the `clientes` and `facturas` tables have to be joined, and the `nombre` attribute of the `clientes` table must be added to the `GROUP BY` clause. You can include an additional attribute in the `SELECT` clause only if it does not split the groups into smaller ones, otherwise the query will need to be rewritten.

```
SELECT c.codcli, c.nombre, COUNT( * )
FROM   facturas AS f JOIN clientes AS C USING( codcli )
WHERE  EXTRACT( year FROM f.fecha ) =
       EXTRACT( year FROM CURRENT_DATE ) - 1
GROUP  BY c.codcli, c.nombre;
```

In some DBMSs, if you execute the above statement you will see that the result is ordered by the customer code, which is the column that defines the grouping. The reason of this behaviour is that, to form the groups of rows, some DBMSs order the rows of the table by the columns in the `GROUP BY` clause.

[1] Check the manuals of the DBMS that you use for specific recommendations.

Therefore, **sometimes, by using the `GROUP BY` clause, we can omit the `ORDER BY` clause**. It should be taken into account that the `GROUP BY` clause can be implemented in other ways, for example, by using a hashing function. Each DBMS chooses the best algorithm, to solve the `GROUP BY` clause as efficiently as possible. **If the chosen algorithm is based on a hashing function, the result will not be ordered as we want, and we will have to use the `ORDER BY` clause**. The manual of the DBMS we use includes the type of algorithm that implements the grouping.

4.3 The `HAVING` clause

The aggregate functions can be used in the `HAVING` clause which can appear after `GROUP BY` clause, to include restrictions on the groups made (the groups which do not fulfill the condition will be removed). The syntax of SQL queries, including the different clauses that we have seen so far, is the following:

```
SELECT [DISTINCT] { * | columna [, columna ]}
FROM   tabla
[ WHERE condición_de_búsqueda ]
[ GROUP BY columna [, columna ]
[ HAVING condición_para_el_grupo ] ]
[ ORDER BY columna [ASC|DESC] [, columna [ASC|DESC] ]
[ FETCH FIRST n ROWS ONLY ];
```

The queries in which the `GROUP BY` appears, show a row per each obtained group. To execute the `GROUP BY` clause, we take the rows of the table which satisfy the constraints specified in the `WHERE` clause, and then they are grouped according to the values in the column (or columns) included in the `GROUP BY` clause. By means of the `HAVING` clause, a restriction is applied on the groups obtained by the `GROUP BY` clause, selecting the groups which fulfill the condition in the `HAVING` clause.

It is important to emphasize that **only the columns that define the grouping and aggregate functions defined on other columns of the table** can appear in the `HAVING` clause. The same holds for the **`SELECT` clause. Only the columns included in the `GROUP BY` clause and aggregate functions defined on other columns of the table can be shown**.

A typical error is produced when the `SELECT` or `HAVING` clauses include some column which neither is specified in the `GROUP BY` clause, nor is included within an aggregate function. This is the reason why, in the previous second example, we must include the `nombre` attribute in the `GROUP BY` clause.

5 Examples

1. *Show the average cost per invoice, without taking into account discounts nor VAT (IVA).*

   Note that the cost of an invoice is calculated adding the cost of all lines.

   The cost of each line is the number of units multiplied by the unit cost (`precio`).

   Given that we want to calculate the average cost, we will add the cost of all invoice lines and, then, we will divide this total cost by the total number of invoices.

   The solution to this exercise is presented below:

   ```
   SELECT ROUND( SUM( cant * precio ) /
          COUNT( DISTINCT codfac ), 2 ) AS importe_medio
   FROM   lineas_fac ;
   ```

```
 importe_medio
---------------
        404.12
```

Note that the result is rounded to two decimals because it is a result in euros.

2. *Show the date of the first invoice of the customer with code 210. Moreover, we want to show the date of the last invoice and the number of days between both invoices.*

   As commented before, some aggregate functions could be applied to the date datatype. In general, `MIN` and `MAX` functions can be used with all datatypes with an order defined.

   In this example, both functions are used to obtain the first and the last invoices. The number of days between them are calculated by subtracting both values, as we did in session 3.

```sql
SELECT MIN( fecha ) AS primera, MAX( fecha ) AS ultima,
       MAX( fecha ) - MIN( fecha ) AS dias
FROM   facturas
WHERE  codcli = 210;
```

```
  primera   |   ultima   | dias
------------+------------+------
 13/09/2000 | 18/12/2004 | 1557
```

3. *We want to show a list of the customers who have more than 5 invoices with 18% VAT, and we also want to include the number of invoices of each of them.*

   First, we must choose the table that should appear in the `FROM` clause. Note that the requested data are included in the `FACTURAS` table. This table has one row for each invoice, while the results should show one row for each customer who meets the restrictions.

   There are two constraints: one on the invoices (18 % VAT) and the other about customers (who must have more than 5 invoices of this type).

   The first restriction must appear in the `WHERE` clause, because this clause chooses the rows of the table in the `FROM` clause which fulfill the condition.

   The second restriction must be applied on the customers, after we have counted how many invoices each customer has. For that, we have to group invoices, and since we want to count the invoices of each customer, the grouping must be done on the `codcli` column.

   Once the groups of invoices have been made we can apply the second restriction, which should be included in the `HAVING` clause. Since the table which is in the `FROM` clause contains one row for each invoice, and we want to count the number of invoices per customer, the `COUNT( * )` function can be used.

   Finally, we show in the `SELECT` clause, for each customer, her/his code and the number of invoices she/he has.

```sql
SELECT codcli, COUNT( * ) AS facturas
FROM   facturas
WHERE  iva = 18
GROUP  BY codcli
HAVING COUNT( * ) > 5;
```

4. *We want to show the number of invoices issued each year, such that the year in which the number of invoices is higher appears first. Furthermore, the number of different customers, salespeople and dates of the invoices must be shown for each year.*

```
SELECT EXTRACT( year FROM fecha ) AS año,
       COUNT( * ) AS nfacturas,
       COUNT( DISTINCT codcli ) AS nclientes,
       COUNT( DISTINCT codven ) AS nvendedores,
       COUNT( DISTINCT fecha ) AS ndias
FROM   facturas
GROUP  BY EXTRACT( year FROM fecha )
ORDER  BY 2 DESC;
```

5. *For the customers whose code is between 240 and 250, we want to show the customer name and the number of invoices that the customer has of each VAT type.*

```
SELECT c.codcli, c.nombre, COALESCE( f.iva, 0 ),
       COUNT( * ) AS facturas
FROM   facturas AS f JOIN clientes AS c USING ( codcli )
WHERE  c.codcli BETWEEN 240 AND 250
GROUP  BY c.codcli, c.nombre, COALESCE( f.iva, 0 )
```

Note that the invoices have been grouped by two criteria, thus one group contains the invoices of a customer (`c.codcli, c.nombre`) where the VAT type is the same (`COALESCE( f.iva, 0 )`). Since we must consider the null VAT value as zero, we have used the `COALESCE` function. Otherwise, the invoices of each customer with a VAT value equal to null would be included in a separate group, because the `GROUP BY` clause treats null values as if they were all the same value.

If you execute this query, you will see if the order in which the columns appear in the `GROUP BY` clause determines the ordering of the results. Here, we can deduce if the groups are formed by sorting. As explained in section 4.2, sometimes we can avoid using the `ORDER BY` clause if we use the `GROUP BY` clause.

6. *We want to show the gross amount invoiced in each month. For each month, we want to show its number and the amount invoiced in euros with a nice format. To make the interpretation easier, we also want to show the amount of each month as a bar with a star character ('*') per each thousand euros invoiced.*

For instance, the output to be obtained is the following one:

```
 mes | facturacion |          un_*_por_cada_mil_euros
-----+-------------+-------------------------------------------
   1 |   32,914.32 | *********************************
   2 |   34,490.64 | **********************************
   3 |   26,541.67 | **************************
   4 |   42,992.37 | ******************************************
   5 |   39,719.26 | ****************************************
   6 |   32,575.98 | *********************************
   7 |   33,725.54 | *********************************
   8 |   28,440.20 | ****************************
   9 |   35,768.02 | ************************************
  10 |   38,407.27 | *************************************
  11 |   29,522.02 | *****************************
  12 |   29,024.10 | *****************************
(12 rows)
```

As can be easily deducted, the data must be grouped by the invoice month. The `to_char` function is used to show numbers with the required format, and the `lpad` function is used to

print one '*' per each thousand euros. Finally, the `cast( x as integer)` function is used to convert the result of the division to integer.

```
SELECT extract( month from f.fecha ) as mes,
       to_char( sum( l.cant * l.precio ), '999,999.99' )
       as facturacion,
       lpad( '',
             cast( ( sum( l.cant * l.precio ) / 1000 ) as integer),
             '*' )
       as "un_*_por_cada_mil_euros"
FROM   facturas as f join lineas_fac as l using ( codfac )
GROUP  BY extract( month from f.fecha )
ORDER  BY mes ;
```

6 Subqueries

In the previous section, we have practised the way to make a column calculation. For example, it is possible to obtain the cost of the most expensive article using the following query:

```
SELECT MAX( precio )
FROM   articulos;
```

However, how could we obtain the information related to this article? (description, cost, ...)? The answer is easy: using *subqueries*.

The query below shows the information of the most expensive article using a subquery:

```
SELECT *
FROM   articulos
WHERE  precio = ( SELECT MAX( precio ) FROM articulos );
```

First, the query in brackets (the subquery) is executed. Next, it is replaced by the returned value. Finally, the main query is executed. To avoid execution errors in this kind of queries, **it is important that the subquery returns a single value**.

7 Session exercises

As usual, the exercises are to be solved individually. Results can be found on the course website, so that you can check if your queries return the expected results.

1. How many towns are there in the Comunidad Valenciana? Remember that the province codes are 03 (Alicante), 12 (Castellón) and 46 (Valencia).

2. Obtain the total cost of the articles in the store. The stock column indicates the number of units of each article in the store. Therefore, the stored value can be obtained multiplying the stock by the cost per unit.

3. In how many towns, with postal code starting with 12, do customers reside?

4. Obtain the maximum and minimum value of stock for articles whose price is between 1 and 50 euros. Show the average value of both. Note that, when working in euros, the value is rounded to euro cents.

5. What is the average cost of articles whose stock is higher than 10 units? Note that, when working in euros, the value is rounded to euro cents.

6. Taking into account only towns with customers, what is average number of customers per town?

7. What is the number of articles out of stock?

8. Show the information of the most recent invoices, that is, the information of the invoices done the last day an invoice was made.

   Note: You may have to use a subquery to solve this exercise.

9. Show the code of the articles and how many units of each have been sold. Order the result by the code of the article.

10. Modify the previous query to show only the articles which are sold in more than 12 invoices.

11. Modify the previous query to show also the description of the article. Order the result by the number of articles sold.

12. Show the name of the salespeople with more than 4 invoices in the same month.

13. Show the minimum and maximum retail prices of the articles whose code begins with "R". Show also the average discount applied to the article (show two decimal places).

    The result must be ordered such that the articles whose difference between the maximum and the minimum of the retail prices is lower are shown first.

14. For each customer who has invoices with 18% VAT or without discount from more than 5 different salespeople, show the code and the name of the customer, the number of invoices and the number of salespeople with whom the customer has worked.

15. Show the number of invoices made in each month of the last year by the salespeople whose code is equal to 255, 355 or 455.

16. Modify the previous query to show also the number of invoices made by each salesperson each month. The result must be ordered by the month and, within the same month, by the salesperson.

17. In this exercise you should describe in natural language what the `SQL` queries does.

```
SELECT a.codart, a.descrip, MAX( l.precio ) as precio
FROM   lineas_fac AS l JOIN articulos AS a USING( codart )
WHERE  UPPER( a.codart ) LIKE 'U%'
GROUP  BY a.codart, a.descrip
HAVING MAX( l.precio ) = MIN( l.precio );
```

    Would it have made sense to include the following `HAVING` clause? Explain your answer.

```
HAVING MAX( a.precio ) = MIN( a.precio );
```

    And what about the clause below?

```
HAVING MAX( l.precio ) = MIN( a.precio );
```

8 What you don't have to forget

- When aggregate functions are used without grouping, the query returns only one row.
- Aggregate functions are only used in `SELECT` or `HAVING` clause (never in the `WHERE` clause).
- The query has two clauses to define restrictions: `WHERE` and `HAVING`. It is very important to know where to put the conditions: **the restrictions applied on each row must be included in the `WHERE` clause**; **while the restrictions applied on each group of rows** (which usually involve aggregate functions) **must be included in the `HAVING` clause**.
- Since the grouping is internally made through an ordering, sometimes it is possible to avoid ordering the results by correctly sorting the columns in the `GROUP BY` clause. **But beware**, it will depend on the DBMS implementation of the grouping, and therefore, it will be necessary to consult the manual first.
- It will be necessary to include the `DISTINCT` clause in a query with a `GROUP BY` clause, only if the columns in the `SELECT` clause are a subset of the columns in the `GROUP BY` clause.
- In the groups made by using the `GROUP BY` clause (remember that these are groups of rows), we can only see the value of the columns used in the grouping (because the value of these columns is the same in all the rows of each group). Therefore, only these columns can be directly written in the `SELECT` and `HAVING` clauses. Additionally, we can include in these clauses some aggregate function, which is usually applied to the columns which are not included in the `GROUP BY` clause.