

# Tema 5 - Consultas avanzadas

---

- Tema 5 - Consultas avanzadas
  - 5.1 Uso de **LIMIT** y **OFFSET**
  - 5.2 Funciones de cálculo con grupos
  - 5.3 Agrupaciones
    - 5.3.1 Uso de **GROUP BY**
  - 5.4 Condiciones **HAVING**
  - 5.5 Funciones de grupo anidadas
  - 5.6 Subconsultas
    - 5.6.1 Subconsultas simples
    - 5.6.2 Subconsultas que devuelven más de una fila
    - 5.6.3 Subconsultas correlacionadas
    - 5.6.4 Subconsultas insertadas en las cláusulas **FROM** y **JOIN**
    - 5.6.5 Subconsultas escalares
  - 5.7 Consultas con **WITH**
    - 5.7.1 **SELECT** con **WITH**
- Ampliación
- Bibliografía

## 5.1 Uso de **LIMIT** y **OFFSET**

**LIMIT** y **OFFSET** nos permiten obtener solamente una parte de los resultados de una consulta. Tienen la siguiente sintaxis:

```
SELECT select_list
  FROM table_expression
  [ ORDER BY ... ]
  [ LIMIT { number | ALL } ] [ OFFSET number ]
```

Si se proporciona un límite, no se devolverán más filas que esa cantidad, aunque es posible que la consulta en sí devuelva menos.

**OFFSET** indica que se omitan un número determinado de filas antes de comenzar a devolverlas.

Si se usan a la vez, primero se omiten las filas con **OFFSET** y después se limitan los resultados con **LIMIT**.

Cuando se usa **LIMIT** es importante establecer un orden con **ORDER BY**. Si no, los resultados a obtener pueden ser impredecibles.

Las filas omitidas con **OFFSET** son procesadas en el servidor (por ejemplo, si utilizamos funciones de cálculo), con lo que el uso de **OFFSET** con un número elevado de filas puede ser muy ineficiente.

## 5.2 Funciones de cálculo con grupos

Las funciones de cálculo con grupos son las encargadas de realizar cálculos en vertical (usando datos de diferentes filas) en lugar de en horizontal (usando datos procedentes de la misma fila en la que vemos el resultado).

Ya hemos visto que estas funciones son **SUM**, **COUNT**, **AVG**, **MAX** y **MIN**.

Todas ellas requieren trabajar con grupos (más adelante se explica cómo agrupar filas), si no se indican grupos, las funciones trabajan sobre todos los datos de una tabla.

#### MANEJO DE VALORES NULOS

Las funciones de cálculo agrupado ignoran los valores **NULL**. Si no deseamos ignorarlos, deberemos utilizar funciones de nulos como **COALESCE** para manejar apropiadamente el valor nulo.

Sin embargo, la función **COUNT**, se suele usar con asterisco (\*) como objeto de cálculo, lo que hace que cuente filas independientemente de su contenido. Si usamos una expresión, como **COUNT(salario)**, no contaría las filas que tengan un salario nulo en ellas.

Las funciones anteriores admiten que antepongamos el término **DISTINCT** antes de la expresión. De ese forma solo se tienen en cuenta los valores distintos. Por ejemplo en:

```
SELECT COUNT(DISTINCT salario)
```

El resultado es 4 porque solo hay 4 salarios distintos. Lo mismo ocurre con el resto de funciones, **DISTINCT** hace que se ignoren los valores repetidos.

## 5.3 Agrupaciones

Lo normal es utilizar las funciones anteriores, no para una tabla completa sino para grupos de filas en base a un criterio. Esta técnica es la que se conoce como agrupación de filas y provoca que, de cada grupo, se muestre una sola fila.

Para ello se utiliza la cláusula **GROUP BY** que permite indicar en base a qué registros se realiza la agrupación.

Con **GROUP BY** la sintaxis completa instrucción **SELECT** queda de esta forma:

```
SELECT listaDeExpresiones
FROM listaDeTablas
[JOIN tablasRelacionadasYCondicionesDeRelación]
[WHERE condiciones]
[GROUP BY grupos]
[HAVING condicionesDeGrupo]
[ORDER BY listaDeExpresiones];
```

### 5.3.1 Uso de **GROUP BY**

En el apartado **GROUP BY**, se indica el nombre de las columnas (o expresiones más complejas) por las que se agrupa. La función de este apartado es crear una única fila por cada valor distinto en las columnas del

grupo. Si por ejemplo agrupamos en base a las columnas tipo y modelo en una tabla de existencias, se creará un único registro por cada tipo y modelo distintos:

Si la tabla de existencias sin agrupar es:

Tipo	Modelo	N_Almacen	Cantidad
AR	6	1	2500
AR	6	2	5600
AR	6	3	2430
AR	9	1	250
AR	9	2	4000
AR	9	3	678
AR	15	1	5667
AR	20	3	43
BI	10	2	340
BI	10	3	23
BI	38	1	1100
BI	38	2	540
BI	38	3	700

Si ahora ejecutamos la siguiente instrucción:

```
SELECT tipo,modelo  
FROM existencias  
GROUP BY tipo,modelo;
```

Obtendremos este resultado:

Tipo	Modelo
AR	6
AR	9
AR	15
AR	20
BI	10
BI	38

Los datos se resumen. Por cada tipo y modelo distintos se creará un grupo de modo que solo aparece una fila por cada grupo. Los datos n\_almacen y cantidad no están disponibles ya que varían en cada grupo. Solo se podrá mostrar los datos agrupados o datos sobre los que realicemos cálculos.

Es decir, esta consulta es errónea:

```
SELECT tipo,modelo, cantidad  
FROM existencias  
GROUP BY tipo,modelo;
```

```
SELECT tipo, modelo, cantidad
      *
```

La razón es el uso de la columna *cantidad*, en el **SELECT** como no se ha agrupado por ella y no es un cálculo sobre *tipo* y/o *modelo* (las columnas por la que ese está agrupando), se produce el error.

Lo normal es agrupar para obtener cálculos sobre cada grupo. Por ejemplo podemos modificar la consulta anterior de esta forma:

```
SELECT tipo, modelo, cantidad, SUM(Cantidad)
FROM existencias
GROUP BY tipo, modelo;
```

y se obtiene este resultado:

Tipo	Modelo	SUM(Cantidad)
AR	6	10530
AR	9	4928
AR	15	5667
AR	20	43
BI	10	363
BI	38	1740

Se suman las cantidades para cada grupo. Igualmente podríamos realizar cualquier otro tipo de cálculo (**AVG**, **COUNT**, ...)

## 5.4 Condiciones **HAVING**

A veces se desea restringir el resultado de una expresión agrupada, por ejemplo podríamos tener esta idea:

```
SELECT tipo, modelo, cantidad, SUM(Cantidad)
FROM existencias
WHERE SUM(Cantidad) > 500
GROUP BY tipo, modelo;
```

Lo que se espera es que solo aparezcan los grupos cuya suma de la cantidad sea menor de 500. Pero, en su lugar, Postgresql devolvería un error del tipo *función de grupo no permitida en el WHERE*.

La razón reside en el orden en el que se ejecutan las cláusulas de la instrucción **SELECT**. Postgresql calcula primero el **WHERE** y luego los grupos (cláusula **GROUP BY**); por lo que no podemos usar en el **WHERE** la función de cálculo de grupos **SUM**, porque es imposible en ese momento conocer el resultado de la **SUMA** al no haberse establecido aún los grupos.

La solución es utilizar otra cláusula: **HAVING**, que se ejecuta una vez realizados los grupos. Es decir, si queremos ejecutar condiciones sobre las funciones de totales, se debe hacer en la cláusula **HAVING**.

La consulta anterior quedaría:

```
SELECT tipo, modelo, cantidad, SUM(Cantidad)
FROM existencias
GROUP BY tipo, modelo
HAVING SUM(Cantidad) > 500;
```

Eso no implica que no se pueda usar **WHERE**. Ésta expresión sí es válida:

```
SELECT tipo, modelo, cantidad, SUM(Cantidad)
FROM existencias
WHERE tipo != 'AR'
GROUP BY tipo, modelo
HAVING SUM(Cantidad) > 500;
```

En definitiva, el orden de ejecución de la consulta marca lo que se puede utilizar con **WHERE** y lo que se puede utilizar con **HAVING**:

- [1] Seleccionar las filas deseadas utilizando **WHERE**. Esta cláusula eliminará columnas en base a la condición indicada
- [2] Se establecen los grupos indicados en la cláusula **GROUP BY**
- [3] Se calculan los valores de las funciones de totales (**COUNT**, **SUM**, **AVG**,...)
- [4] Se filtran los registros que cumplen la cláusula **HAVING**
- [5] El resultado se ordena en base al apartado **ORDER BY**.

## 5.5 Funciones de grupo anidadas

Es posible obtener resultados muy interesantes gracias a la posibilidad de anidar funciones de totales. Así podemos ejecutar esta instrucción:

```
SELECT AVG(SUM(Cantidad))
FROM existencias
GROUP BY tipo, modelo;
```

**Esta expresión, que sí es válida en el estándar SQL, no funciona correctamente en Postgresql, devolviendo el siguiente error:**

```
ERROR: aggregate function calls cannot be nested
LINE 36: SELECT avg(COUNT(id_reserva))
```

Para solventar el problema, se podrá hacer uso de subconsultas (que se explican en el siguiente apartado):

```
SELECT AVG(sumval)
FROM (
    SELECT tipo, modelo, SUM(cantidad) as "sumval"
    FROM existencias
    GROUP BY tipo, modelo;
) datos;
```

Para ello hay que tener en cuenta que la consulta funciona así:

- [1] Se calcula la suma de la cantidad para cada grupo, el resultado es una cantidad por cada tipo y modelo distintos en la tabla
- [2] A partir de ese resultado se calcula la media de las cantidades obtenidas en la subconsulta del punto anterior.

Por lo tanto el resultado es la media de la suma de cantidades de cada grupo.

## 5.6 Subconsultas

El uso de subconsultas es una técnica que permite utilizar el resultado de una tabla **SELECT** en otra consulta **SELECT**. Permite solucionar consultas complejas mediante el uso de resultados previos conseguidos a través de otra consulta.

El **SELECT** que se coloca en el interior de otro **SELECT** se conoce con el término de **SUBSELECT**. Ese **SUBSELECT** se puede colocar dentro de las cláusulas **WHERE**, **HAVING**, **FROM** o **JOIN**.

### 5.6.1 Subconsultas simples

Las subconsultas simples son aquellas que devuelven una única fila. Si además devuelven una única columna, se las llama subconsultas escalares, ya que devuelven un único valor.

La sintaxis es:

```
SELECT listaExpresiones
FROM tabla
WHERE expresión OPERADOR
      (SELECT listaExpresiones
      FROM tabla);
```

El operador puede ser **>**, **<**, **>=**, **<=**, **!=**, **=** o **IN**.

Ejemplo:

```
SELECT nombre_empleado, paga
FROM empleados
WHERE paga <
      (SELECT paga FROM empleados
```

```
WHERE nombre_empleado='Martina')
;
```

Esa consulta muestra el *nombre* y *paga* de los empleados cuya paga es menor que la de la empleada *Martina*. Para que funcione esta consulta, la subconsulta solo puede devolver un valor (solo puede haber una empleada que se llame Martina).

Se pueden usar subconsultas las veces que haga falta:

```
SELECT nombre_empleado, paga
FROM empleados
WHERE paga <
      (SELECT paga FROM empleados
       WHERE nombre_empleado='Martina')
AND paga >
      (SELECT paga FROM empleado
       WHERE nombre_empleado='Luis');
```

En realidad lo primero que hace la base de datos es calcular el resultado de la subconsulta:

```
SELECT nombre_empleado, paga
FROM empleados
WHERE paga < 2500
AND paga < 1870
```

(SELECT paga FROM empleados  
WHERE nombre\_empleado='Martina')

(SELECT paga FROM empleados  
WHERE nombre\_empleado='Luis')

La última consulta obtiene los empleados cuyas pagas estén entre lo que gana Luis (1870 euros) y lo que gana Martina (2500) .

Las subconsultas siempre se deben encerrar entre paréntesis y se deberían (aunque no es obligatorio, sí altamente recomendable) colocar a la derecha del operador relacional.

*\*operadores*

Una subconsulta que utilice los valores **>, <, >=, ...** tiene que devolver un único valor, de otro modo ocurre un error.

Además **tienen que devolver el mismo tipo y número de datos** para relacionar la subconsulta con la consulta que la utiliza (no puede ocurrir que la subconsulta tenga dos columnas y ese resultado se compare usando una sola columna en la consulta general).

### 5.6.2 Subconsultas que devuelven más de una fila

En el apartado anterior se comentaba que las subconsultas sólo pueden devolver una fila. Pero a veces se necesitan consultas del tipo: mostrar el sueldo y nombre de los empleados cuyo sueldo supera al de cualquier empleado del departamento de ventas.

La subconsulta necesaria para ese resultado mostraría todos los sueldos del departamento de ventas. Pero no podremos utilizar un operador de comparación directamente ya que esa subconsulta devuelve más de una fila. La solución a esto es utilizar instrucciones especiales entre el operador y la consulta, que permiten el uso de subconsultas de varias filas.

Esas instrucciones son:

- **ANY O SOME**: Compara con cualquier registro de la subconsulta. La instrucción es válida si hay un registro en la subconsulta que permite que la comparación sea cierta. Se suele utilizar la palabra **ANY** (**SOME** es un sinónimo).
- **ALL**: Compara con todos los registros de la consulta. La instrucción resulta cierta si es cierta toda comparación con los registros de la subconsulta.
- **IN**: No usa comparador, ya que sirve para comprobar si un valor se encuentra en el resultado de la subconsulta.
- **NOT IN**: Comprueba si un valor no se encuentra en una subconsulta.

Ejemplo:

```
SELECT nombre, sueldo
FROM empleados
WHERE sueldo >= ALL (SELECT sueldo FROM empleados);
```

La consulta anterior obtiene el empleado que más cobra. Otro ejemplo:

```
SELECT nombre FROM empleados
WHERE dni IN (SELECT dni FROM directivos);
```

En ese caso se obtienen los nombres de los empleados cuyos dni están en la tabla de directivos.

Si se necesita comparar dos columnas en una consulta IN, se hace de esta forma:

```
SELECT nombre FROM empleados
WHERE (cod1,cod2) IN (SELECT cod1,cod2 FROM directivos);
```

### 5.6.3 Subconsultas correlacionadas

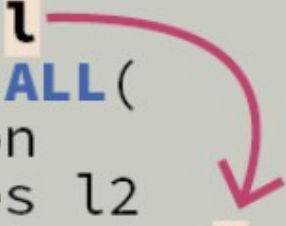


En las subconsultas a veces se puede desear poder utilizar datos procedentes de la consulta principal. Eso es posible utilizando el alias de la tabla que queremos usar de la consulta principal.

Por ejemplo, supongamos que deseamos obtener de una base de datos geográfica, el nombre y la población de las localidades que sean las más pobladas de su provincia. Es decir, las localidades cuya población es la mayor de su provincia. Para ello necesitamos comparar la población de cada localidad con la de todas las localidades de su provincia. Supongamos que la tabla de las localidades almacena el nombre, población y el número de la provincia a la que pertenecen.

La consulta sería:

```
SELECT l.nombre, poblacion
FROM localidades l
WHERE poblacion >= ALL(
  SELECT poblacion
  FROM localidades l2
  WHERE l2.n_provincia = l.n_provincia
)a
```



En el código anterior se observa que dentro de la subconsulta usamos el alias / correspondiente a la tabla de localidades de la consulta principal (por eso se le ha puesto como alias /2 a la tabla localidades en la subconsulta).

Si quieres profundizar con otro ejemplo, puedes echar un vistazo al apartado de subconsultas correlacionadas en Tutoriales YA:

<https://www.tutorialesprogramacionya.com/postgresqlya/temarios/descripcion.php?inicio=50&cod=220&punto=62>

#### 5.6.4 Subconsultas insertadas en las cláusulas FROM y JOIN

El resultado de una operación de tipo **SELECT** es una vista (aunque sea temporal, ya que no se almacena de forma permanente). Y las vistas pueden ser utilizadas dentro de otras vistas (al igual que las tablas).

Así una consulta como esta:

```
SELECT tipo, modelo, SUM(cantidad) suma_cantidad
FROM existencias
GROUP BY tipo, modelo
```

Muestra los tipos y modelos de piezas en los almacenes y la suma de cantidades que poseen sumando la de cada almacén. El resultado es una vista de tres columnas. Lo interesante es que puede ser un inicio para una nueva consulta.

Por ejemplo:

```
SELECT tipo, COUNT(modelo), SUM(suma_cantidad)
FROM (
  SELECT tipo, modelo, SUM(cantidad) suma_cantidad
  FROM existencias
  GROUP BY tipo, modelo
) datos
GROUP BY tipo;
```

Postgresql nos fuerza a que la subconsulta del **FROM** tenga dentro un alias.

La posibilidad de usar así las subconsultas se extiende a la cláusula JOIN:

```
SELECT tipo, modelo, precio_venta
FROM piezas P1
JOIN (
  SELECT MAX(precio_venta) max_precio_venta
  FROM piezas
) P2 ON P1.precio_venta=P2.max_precio_venta;
```

Así, esta consulta nos muestra el tipo y modelo de las piezas que tiene el precio de venta más alto. No es la única forma de resolver esta consulta, pero nos permite observar la capacidad de usar subconsultas de forma muy avanzada.

A esta técnica se le llama usar vistas en línea (en inglés *inline views*). Ahora bien, para que eso sea posible las columnas de la subconsulta deben usar alias obligatorios para que nos e repita el nombre de la columna y especialmente en las columnas con datos calculados.

Por ejemplo supongamos que deseamos saber el nombre de los empleados que tienen un jefe que gana más de 2000 euros. Para ello primero haremos un consulta que obtenga el nombre e identificador de los empleados que ganan más de 2000 euros; luego bastará con combinar esta consulta y la tabla de empleados de modo que el jefe del empleado esté en la lista de empleados que ganan más de 200 euros.

En formato SQL 92 sería:

```
SELECT e.nombre
FROM empleados e,
  (SELECT nombre, id_empleado FROM empleados WHERE salario>2000) e2
WHERE e.id_jefe=e2.id_empleado;
```

Usando SQL 99:

```
SELECT e.nombre
FROM empleados e,
JOIN (SELECT nombre, id_empleado FROM empleados WHERE salario>2000) e2
  ON(e.id_jefe=e2.id_empleado);
```

## 5.6.5 Subconsultas escalares

Las subconsultas escalares son aquellas que devuelven un único resultado. En definitiva son **SELECT** que devuelven una única consulta y un único valor.

Estas subconsultas son muy útiles porque se pueden utilizar en muchas partes del lenguaje **SELECT**. Concretamente:

- En cualquier cláusula de la instrucción **SELECT** (excepto **GROUP BY**)
- En las funciones **CASE** y **DECODE**
- En cualquier cláusula **WHERE** de una instrucción **DML** (**INSERT**, **UPDATE**, **DELETE**)
- En la cláusula **SET** de la instrucción **UPDATE**

Desgraciadamente (y la tentación es grande), no se pueden utilizar en las restricciones **CHECK** para validar datos. Esta necesidad sólo la pueden resolver los triggers.

Ejemplo de uso de subconsulta escalar:

```
SELECT nombre,  
  (SELECT COUNT(*)  
   FROM comunidades  
   JOIN provincias USING (ID_COMUNIDAD)  
   where ID_comunidad=c.id_comunidad  
   GROUP BY ID_COMUNIDAD  
  ) AS numero_provincias  
FROM comunidades c;
```

Esta consulta obtiene el nombre de cada comunidad autónoma, seguida del número de provincias que tiene (habría otras formas más sencillas de conseguirla).

Con imaginación, podemos conseguir consultas tan espectaculares como esta:

```
SELECT nombre,  
  DECODE(  
    (SELECT COUNT(*)  
     FROM comunidades  
     JOIN provincias USING (id_comunidad)  
     WHERE id_comunidad=c.id_comunidad  
     GROUP BY id_comunidad  
    ),  
    1, 'Uniprovincial', 'Multiprovincial'  
  )  
FROM comunidades c;
```

Basada en la anterior, ahora obtenemos el nombre de cada comunidad y un texto que dice si es *uniprovincial* o *multiprovincial*.

## 5.7 Consultas con WITH

**WITH** es un operador que nos permite escribir consultas *auxiliares* que podemos utilizar en una consulta más larga y compleja. Esas sentencias, a las cuales se les conoce en ocasiones como CTEs (*Common Table Expressions* o Expresiones comunes de tabla) se puede considerar como la definición de tablas (vistas) temporales que existen solo para una consulta.

Cada instrucción auxiliar en una cláusula **WITH** puede ser **SELECT**, **INSERT**, **UPDATE** o **DELETE**; y la cláusula **WITH** en sí se adjunta a una declaración principal que también puede ser **SELECT**, **INSERT**, **UPDATE** o **DELETE**.

Por ahora lo haremos solamente con consultas **SELECT**.

### 5.7.1 SELECT con WITH

La misión principal de una sentencia **WITH** asociada a una sentencia **SELECT** es **dividir un problema complejo en pequeños problemas más simples**.

Por ejemplo:

```
WITH regional_sales AS (  
    SELECT region, SUM(amount) AS total_sales  
    FROM orders  
    GROUP BY region  
) , top_regions AS (  
    SELECT region  
    FROM regional_sales  
    WHERE total_sales > (SELECT SUM(total_sales)/10 FROM regional_sales)  
)  
SELECT region,  
       product,  
       SUM(quantity) AS product_units,  
       SUM(amount) AS product_sales  
FROM orders  
WHERE region IN (SELECT region FROM top_regions)  
GROUP BY region, product;
```

Esta consulta muestra las ventas totales por producto, pero solamente en las regiones con mayores ventas. La cláusula **WITH** define dos consultas auxiliares llamadas **regional\_sales** y **top\_regions**, donde la salida de **regional\_sales** se utiliza en **top\_regions**, y la salida de esta última se utiliza en la consulta **SELECT** principal.

Este ejemplo se podría haber hecho sin **WITH**, pero necesitaríamos dos niveles de subconsultas. ¿Te atreves a hacerlo?

Una propiedad útil de las cláusulas **WITH** es que normalmente se evalúan una sola vez aunque se haga referencia a ellas más de una vez desde alguna subconsulta o la consulta **SELECT** principal. Esto hace que la ejecución de la consulta sea más eficiente.

# Ampliación

---

Dentro de este tema no estamos trabajando algunos conceptos que quizás son importantes, pero para los cuales no tenemos mucho tiempo. Te dejo una lista y algunos enlaces para que los puedas trabajar por tu cuenta.

- **GROUPING SET**, **ROLLUP**, **CUBE**: <https://www.postgresql.org/docs/13/queries-table-expressions.html#QUERIES-GROUPING-SETS>
- **UNION**, **INTERSECT**, **MINUS**: <https://www.postgresql.org/docs/13/queries-union.html>
- Uso de **EXISTS** en subconsultas: <https://www.postgresqltutorial.com/postgresql-exists/>
- Consultas recursivas o jerárquicas: <https://www.2ndquadrant.com/en/blog/oracle-to-postgresql-start-with-connect-by/>

## Bibliografía

---

1. Apuntes de Gestión de Bases de Datos de Jorge Sánchez. (Consultas con agrupamiento)  
<https://jorgesanchez.net/manuales/sql/select-totales-sql2016.html>
2. Documentación de Postgresql: <https://www.postgresql.org/docs/13/index.html>
3. Apuntes de Gestión de Bases de Datos de Jorge Sánchez. (Subconsultas)  
<https://jorgesanchez.net/manuales/sql/select-subconsultas-sql2016.html>
4. Consultas con **WITH**. <https://www.postgresql.org/docs/13/queries-with.html>