

Fastbook 08

Tratamiento de Datos (Excel y SQL)

PostgreSQL: vamos a alcanzar
la cima



08. PostgreSQL: vamos a alcanzar la cima

En este fastbook, trabajaremos dos puntos principales: primero, el uso de funciones agregado que nos permitirá (a partir de un set de datos) generar nueva información condensando los datos de la forma que queramos fusionar; segundo, el uso de subqueries o consultas anidadas, que nos permitirá aumentar aún más nuestra capacidad de consulta.

Autores: Carlos Manchón y Breogán Cid

Generar información agregada a partir de una tabla

Comando ANY

Funciones de agregado

Comando ALL

GROUP BY

Un ejemplo más complejo

HAVING

Conclusiones

Introducción a las subqueries

Ejercicios propuestos

Comando EXISTS

Solución a los ejercicios

Generar información agregada a partir de una tabla

X Edix Educación

Si recuerdas la estructura de la **sentencia SELECT**, a la que ya hemos recurrido en el pasado varias veces, podrás observar que hasta ahora no hemos hablado de los puntos 4 y 5.

- 1 `SELECT <columna1> [,<columna2>, ..., <columna N>]`
- 2 `FROM <tabla1>[,<tabla2>], ..., [<tabla N>]`
- 3 `[WHERE <condicion >]`
- 4 `[GROUP BY <columna1>, <columna2>, ..., <columnaN>]`
- 5 `[HAVING <condicion>]`
- 6 `[ORDER BY <columna1>, <columna2>, ..., <columnaN>]`
- 7 `[LIMIT N]`

Ambos comandos (GROUP BY y HAVING) son los que utilizamos cuando queremos **realizar agregaciones de información sobre una o varias tablas**.

¿En qué momento podemos necesitar agregar información? Principalmente cuando necesitemos realizar informes resumen sobre una tabla que contenga información con una granularidad (detalle) muy fina.

Es decir, imaginemos que tenemos recogida en una tabla todas las ventas que ha generado nuestra tienda a lo largo del tiempo con la fecha y hora en la que se han producido esas ventas.

Si queremos generar un informe para analizar cómo evolucionan nuestras ventas, tendremos que agregar la información y mostrarla al **nivel de detalle que sea más oportuno** para el interés del informe (mensual para analizar la evolución durante el año, anual para hacer un análisis estratégico, a largo plazo).

Usaremos el comando GROUP BY para definir los campos por los que vamos a realizar la agregación.

En el caso de no querer agregar por algún campo, no incluiremos en la consulta este comando, usando simplemente las funciones de agregado (que ahora veremos) sobre las columnas que se seleccionan en la consulta.

El comando **HAVING** permite realizar filtrados sobre los datos agregados.

Es decir, no realizaremos el filtrado en la parte del WHERE, sino con el comando HAVING.

Funciones de agregado

X Edix Educación

PostgreSQL ofrece numerosas funciones de agregado. En este fastbook veremos las principales, comunes a otros motores de bases de datos.

Para ver los distintos ejemplos, nos apoyaremos en la siguiente tabla (**tabla_empleados**) que contiene el nombre, edad y salario de una empresa.

nombre	edad	salario
CARLOS	26	2400
LUIS	40	1400
ENRIQUE	36	2600
LAURA	34	1700
IRENE	35	1800
PEDRO	27	2100
JAVIER	33	2100
ELENA	23	1800
MANUELA	36	2300
SILVIA	37	2200
RODRIGO	29	1000
SANDRA	31	2900
MIGUEL	44	2700
LUISA	40	2700

Función MAX()

La función **MAX()** devolverá el valor máximo de la columna sobre la tabla o de la parte de la tabla que hayamos agrupado con el comando GROUP BY (esto último lo veremos más adelante).

Simplemente tendremos que añadir dentro de la función la columna a la que queremos hacer su cálculo agregado. Es decir:

```
SELECT MAX(edad) max_edad  
FROM tabla_empleados
```

Esta consulta devolverá el valor:

max_edad
44

Si queremos calcular la edad y el salario máximos, usamos la función dos veces:

```
SELECT MAX(edad) max_edad,MAX(salario) max_salario  
FROM tabla_empleados
```

Esta consulta nos devolverá:

max_edad	max_salario
44	2900

Cuando usamos funciones de agregado solo podremos indicar en el apartado SELECT columnas a las que se aplique una función de agregado y/o columnas por las que queremos agrupar información (en el comando GROUP BY). En cualquier otro caso, la consulta devolverá un error. Es decir:

```
SELECT nombre,MAX(edad)  
FROM tabla_empleados
```

Este ejemplo devolverá un error, pues la columna nombre no tiene una función de agregado ni hay un comando GROUP BY que la utilice. El error tendría la siguiente forma:

```
ERROR: column "table_empleados.nombre" must appear in the GROUP BY clause  
or be used in an aggregate function LINE 1: SELECT nombre,MAX(edad) ^ SQL  
state: 42803 Character: 8
```

Bastante claro el mensaje, ¿verdad?

Función MIN()

Si la función MAX() devuelve el valor máximo, la función MIN() devuelve el valor mínimo.

Por lo que:

```
SELECT MIN(edad) min_edad  
FROM tabla_empleados
```

Devolverá:

min_edad
23

Un aspecto que no hemos comentado es que estas funciones, según el tipo de la columna a la que se apliquen, **funcionarán de una manera u otra**. Es decir, para columnas de tipo numérico parece claro cómo considerarán el valor máximo/mínimo, pero ¿y para columnas de tipo cadena de texto o de tipo fecha?

Para el caso de **columnas de tipo fecha** también podemos intuir cómo se comportará, puesto que estamos acostumbrados en el día a día a realizar comparaciones entre fechas, y sabemos extraer la fecha máxima y mínima entre dos fechas.

Para el caso de **cadenas de texto**, como ya vimos en los operadores de comparación, sabemos que los caracteres de la cadena tienen un orden. Para calcular la cadena de texto máxima ó mínima sobre un conjunto se evaluará el primer carácter de todas las cadenas, en caso de que haya empate entre varios casos, se mirará el segundo carácter y así sucesivamente.

Por lo tanto:

```
SELECT MAX(nombre) max_nombre, MIN(nombre) min_nombre  
FROM tabla_empleados
```

Devolverá:

max_nombre	min_nombre
SILVIA	CARLOS

Si bien ‘SILVIA’ y ‘SANDRA’ tienen el mismo orden en su primer carácter (‘S’), al comparar el segundo tenemos ‘I’ es > que ‘A’.

Función AVG()

Se trata de la función ‘average’, es decir, **calcula el valor promedio de la columna que introducimos dentro de la función**. Como ya intuirás, solo funciona para columnas de tipo numérico.

Por lo que, si queremos conocer el salario promedio de nuestro listado, haríamos la siguiente consulta:

```
SELECT AVG(salario) salario_promedio  
FROM tabla_empleados
```

Y devuelve:

salario_promedio
2121,428571

Función SUM()

Se trata de la función suma, es decir, **calcula la suma agregada de todos los valores de la columna que introducimos dentro de la función**. De la misma manera que en el caso anterior, solo se puede utilizar para tipos numéricos.

Si quisiéramos saber el gasto total de la empresa en salarios, haríamos la siguiente consulta:

```
SELECT SUM(salario) salario_total  
FROM tabla_empleados
```

Que devolvería:

salario_total
29700

Para ver la siguiente función de agregado vamos a modificar nuestra tabla de consulta y añadiremos lo siguiente:

- Un nuevo empleado del que desconocemos su información de salario.
- Una nueva columna que determine el departamento al que pertenece el empleado.

Algo así:

nombre	departamento	edad	salario
CARLOS	FINANZAS	26	2400
LUIS	FINANZAS	40	1400
ENRIQUE	IT	36	2600
LAURA	IT	34	1700
IRENE	IT	35	1800
PEDRO	IT	27	2100
JAVIER	COMUNICACIÓN	33	2100
ELENA	RRHH	23	1800
MANUELA	RRHH	36	2300
SILVIA	SECRETARÍA	37	2200
RODRIGO	SECRETARÍA	29	1000
SANDRA	SECRETARÍA	31	2900
MIGUEL	DIRECCIÓN	44	2700
LUISA	DIRECCIÓN	40	2700
ÁNGEL	COMUNICACIÓN	35	null

Función COUNT()

Para esta función de agregado veremos **dos modalidades**:

- Count(*) .
- Count(columna) .

En el primer caso, esta función nos contabilizará el número de filas existentes. Para el segundo caso, contará el número de filas para los que la columna indicada tiene un valor distinto a nulo.

Por lo que, ¿qué salida esperarías para la siguiente consulta?

```
SELECT COUNT(*),COUNT(edad),COUNT(salario)
FROM tabla_empleados
```

- a)

15	15	15
----	----	----
- b)

15	14	14
----	----	----
- c)

15	14	15
----	----	----
- d)

15	15	14
----	----	----

Solución

Efectivamente, la respuesta correcta es la 'd'. El número de filas de la tabla es 15, el número de valores no nulos del campo edad es 15 y el número de valores no nulos del campo salario es 14.

Dentro del apartado de esta función vamos a mencionar un comando muy interesante de SQL que, en combinación con COUNT(), puede llegar a ser muy útil.

Nos referimos al comando DISTINCT que aplicado sobre una columna devolverá todos sus valores sin repetir. Si lo aplicamos sobre dos columnas devolverá todas las combinaciones únicas de estas dos columnas. Si sobre estas combinaciones únicas aplicamos la función de agregado COUNT() podremos saber el número de combinaciones únicas.

Veamos varios ejemplos para entender su funcionamiento.

1

Edades distintas de los empleados:

```
SELECT DISTINCT edad edades  
FROM tabla_empleados
```

Que devuelve:

edades
44
40
26
37
29
34
35
36
31
33
27
23

Si observas, solo devuelve 12 valores, siendo 15 el número de filas de la tabla, por lo que habrá 3 casos en los que el valor de edad se encuentra repetido (35, 36 y 40).

2

Combinaciones únicas departamento-salario:

```
SELECT DISTINCT departamento, salario  
FROM tabla_empleados
```

Que devuelve:

departamento	salario
SECRETARÍA	2200
RRHH	1800
SECRETARÍA	2900
IT	1700
IT	2600
FINANZAS	2400
IT	1800
SECRETARÍA	1000
COMUNICACIÓN	
IT	2100
FINANZAS	1400
COMUNICACIÓN	2100
RRHH	2300
DIRECCIÓN	2700

En este caso, devuelve 14 combinaciones, por lo que existe una que se repite (DIRECCIÓN, 2700).

3

Número de salarios distintos:

```
SELECT COUNT(DISTINCT salario) combinaciones  
FROM tabla_empleados
```

Que devuelve:

combinaciones
11

Si queremos saber el **número de combinaciones únicas departamento-salario** (antes hicimos las combinaciones únicas, ahora queremos saber el número), vamos a tener que hacer algo ligeramente diferente, porque COUNT(DISTINCT()) solo funciona para una única columna dentro de DISTINCT.

Es decir, esta query devuelve error:

```
SELECT COUNT(DISTINCT departamento, salario) combinaciones  
FROM tabla_empleados
```

Tenemos dos opciones, para nuestro interés solo vamos a ver la primera: hacer uso de una subquery, que explicaremos más adelante en este fastbook, por lo que este ejemplo servirá como introducción.

```
SELECT COUNT(*)  
FROM (  
    SELECT DISTINCT departamento, salario combinaciones  
    FROM tabla_empleados  
) subquery
```

Observa que, a la hora de hacer subqueries en la cláusula FROM, PostgreSQL obliga a darle un alias a la subconsulta. Así, lo que se encuentra en el ejemplo entre paréntesis se le añade al final el alias (nombre) con el que se le va a identificar. En nuestro caso ‘subquery’.

El uso de subqueries es una herramienta muy potente a la hora de poder realizar consultas sobre tablas, pues nos proporciona un extra a la hora de expresarnos. La filosofía consiste básicamente en consultar sobre una tabla que es el resultado de otra consulta. Y esto lo podemos hacer todas las veces que queramos.

GROUP BY

X Edix Educación

Las funciones de agregado que hemos visto resultan muy útiles, pero se quedan muy cortas si no hacemos uso del comando GROUP BY.

Pensemos en la tabla de empleados, ¿cómo podríamos calcular el salario mínimo por departamento? Muy fácil, usando GROUP BY:

```
SELECT departamento, MIN(salario) salario_minimo  
FROM tabla_empleados  
GROUP BY departamento
```

departamento	salario_minimo
RRHH	1800
COMUNICACIÓN	2100
SECRETARÍA	1000
FINANZAS	1400
IT	1700
DIRECCIÓN	2700

Podemos agrupar por tantas columnas como queramos, incluso podemos agrupar sobre nuevas columnas que generemos al vuelo aplicando funciones sobre las columnas originales.

Practiquemos un poco con esta nueva funcionalidad:

1

¿Cuántos empleados existen por departamento?

```
SELECT departamento, count(*) empleados  
FROM tabla_empleados  
GROUP BY departamento
```

2

Queremos saber cuál es el departamento que mejor paga en promedio. Táctica para enfocarlo: construimos una tabla en la que, por departamento, tenemos el salario promedio; sobre esa tabla ordenamos por salario promedio de mayor a menor y limitamos el resultado de la consulta a 1.

```
SELECT departamento, AVG(salario) salario_promedio  
FROM tabla_empleados  
GROUP BY departamento  
ORDER BY salario_promedio DESC  
LIMIT 1
```

departamento	salario_promedio
DIRECCIÓN	2700

HAVING

X Edix Educación

Como ya hemos comentado antes, usaremos la cláusula HAVING cuando queramos realizar filtrados sobre los agregados. Es muy importante no confundirnos y hacerlo en el comando WHERE.

Vamos a realizar varios ejemplos para comprender en qué momento usar WHERE y en qué momento usar HAVING:

1

Listar empleados cuyo salario sea menor a 1500€.

```
SELECT nombre  
FROM tabla_empleados  
WHERE salario < 1500
```

nombre
LUIS
RODRIGO

2

Listar los departamentos cuyo salario medio sea menor a 2000€.

```
SELECT departamento  
FROM tabla_empleados  
GROUP BY departamento  
HAVING AVG(salario) < 2000
```

departamento
FINANZAS

3

Queremos calcular el salario medio por departamento y quedarnos solo con los departamentos que tengan un salario medio mayor a 2200€, pero no queremos tener en cuenta en el cálculo a los empleados del departamento de SECRETARÍA ni a los empleados de la empresa que tengan una edad menor a 25 años.

```
SELECT departamento, AVG(salario) salario_medio
FROM tabla_empleados
WHERE departamento <> 'SECRETARIA' AND edad > 25
GROUP BY departamento
HAVING AVG(salario) > 2200
```

Recuerda que la cláusula WHERE se ejecuta antes que GROUP BY y HAVING, por lo que si no queremos tener en cuenta para el cálculo de la agregación ciertos valores (en este caso, queremos filtrar departamento y por edad), el lugar en el que tenemos que indicarlo es en el WHERE.

departamento	salario_medio
RRHH	2300
DIRECCIÓN	2700

Introducción a las subqueries

X Edix Educación

Ya hemos visto antes un pequeño ejemplo del uso de las subqueries, hagamos una introducción oficial de estas.

Básicamente, haremos uso de las subqueries cuando necesitemos construir nuestra consulta en varias etapas o pasos. ¿Cómo hacerlo? Si partimos de la base que una query devuelve como resultado una tabla, ¿por qué no hacer consultas sobre esa consulta?, ¿y si lo hacemos una vez, dos, tres o las veces que hagan falta?

Al final como resultado tendremos un conjunto de subconsultas (o subqueries, o consultas anidadas). Su estructura 'tipo' es la siguiente:

```
SELECT #operaciones nivel 1
FROM (SELECT #operaciones nivel 2
      FROM (SELECT #operaciones nivel 3
            FROM ( SELECT #operaciones nivel 4
                  FROM ( SELECT #operaciones nivel 5
                        FROM mitabla
                      ) query_nivel5
                    ) query_nivel4
                  )query_nivel3
                ) query_nivel2
```

Esta estructura la podemos hacer aún más compleja, haciendo uso de cruces de tablas (JOINS) con subconsultas, etc. En la estructura tipo hacemos referencia a subqueries en el FROM, pero también podemos hacer uso de estas dentro del WHERE, como valor para una comparación o como un listado de valores dentro del comando IN.

Veamos una serie de ejemplos para entender su uso.

Subquery en el FROM

Imaginemos la siguiente tabla (tabla_ventas) en la que tenemos las ventas de una cadena de supermercados a nivel municipal.

municipio	provincia	ventas
mun-1	prov-1	3000
mun-2	prov-1	4000
mun-3	prov-2	5000
mun-4	prov-2	1000

Queremos saber las ventas promedio a nivel nacional, es decir, primero deberemos calcular las ventas a nivel provincial para posteriormente calcular el promedio. **Esto tendríamos que realizarlo en dos niveles puesto que no es posible aplicar dos funciones de agregado a la vez.**

```
SELECT AVG(ventas)
FROM (
    SELECT provincia, SUM(ventas) ventas
    FROM tabla_ventas
    GROUP BY provincia
) ventas_por_provincia
```

Subquery en el WHERE

Para la misma tabla anterior, queremos obtener los municipios cuyas ventas sean mayores que las ventas promedio a nivel municipal. Tendríamos que realizar la consulta otra vez a dos niveles: primero debemos calcular la venta promedio a nivel municipal y una vez obtengamos ese valor lo usaremos como filtro.

```
SELECT municipio
FROM tabla_ventas
WHERE ventas > (SELECT AVG(ventas)
                  FROM tabla_ventas
                 )
```

Subquery en el WHERE (versión en IN)

Aparte de la tabla anterior, tenemos una tabla más (tabla_clientes) en la que tenemos recogida toda la información de los clientes.

cliente	correo	edad	municipio
cli-1	cli1@correo.com	31	mun-1
cli-2	cli2@correo.com	45	mun-1
cli-3	cli3@correo.com	23	mun-2
cli-4	cli4@correo.com	19	mun-3
cli-5	cli5@correo.com	25	mun-4

Queremos obtener toda la información disponible de los clientes cuyo municipio haya tenido ventas por encima de 2000€.

```
SELECT *
FROM table_clientes
WHERE municipio IN ( SELECT municipio
                      FROM table_ventas
                      WHERE ventas > 2000
                  )
```

Comando EXISTS



El comando EXISTS se utiliza como **filtro dentro del WHERE** y hace uso también de subqueries. Su funcionamiento consiste en testar si la subquery devuelve filas o no para cada uno de los casos de la query principal. En caso de existir filas, **EXISTS devuelve 1** y por lo tanto el filtro se cumple; en caso de no existir filas, **EXISTS devuelve 0** y por lo tanto el filtro no se cumple.

Como lo importante es comprobar que la subquery devuelve o no datos, lo que devuelva da un poco igual. Por normal general, se coloca en el SELECT de la subquery el valor 1 que es el valor más simple para devolver.

Para entenderlo, realizaremos el mismo ejemplo que hemos visto para las subqueries dentro del comando IN.

Iremos recorriendo la tabla de clientes uno a uno y comprobaremos en la subquery (la tabla de ventas) que el departamento del cliente ha vendido más de 2000€ en la tabla de ventas.

La clave en esta query es cómo se relacionan los campos municipios de ambas tablas.

Como último detalle de este comando, hay que comentar que también existe la posibilidad de utilizar la versión negada, esto es, NOT EXISTS.

Comando ANY

X Edix Educación

El comando ANY es otra opción para trabajar con subqueries. Para poder usarlo necesitamos cumplir una serie de requisitos:

- La subquery solo puede devolver una columna.
- El comando ANY debe ir precedido antes por un operador de comparación, es decir, `=, <=, >=, >, < o <>`.
- Si al menos uno de los elementos de la subquery cumple con la condición del operador de comparación, el elemento de la query principal pasará el filtro; en caso contrario, no lo pasará y se filtrará.

Veámoslo con un ejemplo. Volviendo a la tabla de empleados -departamentos y salario, queremos obtener los empleados que ganan igual o más que el salario máximo de al menos un departamento.

```
SELECT *
FROM tabla_empleados
WHERE salario >= ANY (SELECT MAX(salario)
                      FROM tabla_empleados
                      GROUP BY departamento
                     )
```

Como última nota de este comando: cuando usemos el comando ANY con un '`=`' previo, es decir, `= ANY()`, el funcionamiento es equivalente a `IN()`.

Comando ALL

X Edix Educación

Podemos decir que el comando **ALL** es equivalente al comando **ANY**, con la única salvedad que ANY devolvía el filtro como VERDADERO si al menos un elemento de la subquery cumplía con la comparación. En ALL, todos los elementos de la subquery deben cumplir con la comparación.

El ejemplo equivalente para el comando ALL respecto el comando ANY sería obtener los empleados que ganan igual o más que el salario máximo de todos los departamentos.

```
SELECT *
FROM tabla_empleados
WHERE salario >= ALL (SELECT MAX(salario)
                      FROM tabla_empleados
                      GROUP BY departamento
                     )
```

Un ejemplo más complejo

X Edix Educación

El uso de subqueries es básico para poder **realizar consultas complejas sobre unos datos**. Si esto lo unimos a los conocimientos que hemos ido adquiriendo en fastbooks anteriores el resultado es que podemos llegar a construir queries realmente potentes que nos generan información muy útil.

Veamos un ejemplo donde la complejidad sea mayor.

Desde el departamento de RR. HH. quieren realizar un análisis y conocer por empleado cuánto dinero está ganando de más o de menos respecto del promedio de su departamento, y también del promedio de la empresa.

Enfoque

1

Primero pensemos qué formato de salida va a tener nuestra tabla deseada. Sería algo así, ¿verdad? (Es un ejemplo que no se ajusta a los datos que tenemos).

empleado	diferencia_vs_departamento	diferencia_vs_empresa
LUIS	300	-100

Leeremos dicha tabla como que Luis gana 300€ más que el promedio de su departamento; sin embargo, gana 100€ menos que el promedio de la empresa.

Eso nos indica que el departamento al que pertenece Luis está mal pagado en comparación al resto de departamentos; sin embargo, Luis dentro de su departamento es el mejor pagado o uno de los mejores.

2

Previamente, para poder hacer el cálculo de un empleado respecto a su departamento y a la empresa (salario empleado-salario promedio), necesitaremos tener esa información a nivel de todas las filas. Algo así:

empleado	departamento	salario	salario_promedio_departamento	salario_promedio_empresa

3

A lo mejor, en este momento no ves forma de sacar esa información unida, pero, ¿y separada? Seguro que somos capaces de construir estas tres tablas:

- Tabla empleado-departamento-salario.
- Tabla departamento-salario promedio.
- Tabla salario promedio de la empresa.

La construcción de la primera tabla es trivial.

```
SELECT nombre, departamento, salario  
FROM tabla_empleados
```

Acabamos de aprender a construir la segunda tabla, usando agregados.

```
SELECT departamento, AVG(salario) salario_promedio_departamento  
FROM tabla_empleados  
GROUP BY departamento
```

La construcción de la tercera tabla es prácticamente igual a la anterior, entendiendo bien cómo queremos agregarlo.

```
SELECT AVG(salario) salario_promedio_empresa  
FROM tabla_empleados
```

Como queremos realizar el agregado sobre toda la tabla de empleados (lo que representa a la empresa), no colocamos ningún elemento en el GROUP BY.

4

Llegados a este punto, tenemos las siguientes tres tablas:

empleado	departamento	salario

departamento	salario_promedio

salario_promedio_empresa

5

¡Tenemos prácticamente el ejercicio resuelto! Solo nos queda saber cómo juntarlas. Lo vamos a realizar en dos pasos, para que comprendas bien la lógica que hay, pero normalmente lo podríamos hacer todo junto en un único paso.

6

Unamos la primera tabla con la segunda mediante una operación JOIN (no nos interesa hacer un LEFT o RIGHT JOIN) a través de la columna común ‘DEPARTAMENTO’ y generamos una tabla (la llamaremos tabla4) con el siguiente formato:

empleado	departamento	salario	salario_promedio

```
SELECT nombre,departamento,salario,salario_promedio_departamento
FROM tabla1
JOIN tabla2
ON tabla1.departamento = tabla2.departamento
```

Para simplificar, hemos llamado a las tablas ‘tabla1’ y ‘tabla2’, posteriormente, cuando fusionemos todos estos pasos en una única query, haremos la sustitución de estos nombres por la subquery a la que representa.

7

De la misma manera que en el punto anterior, uniremos la tabla resultante en el punto 6, con la 3^a tabla que generamos en el punto 4 (la que contiene solo una fila con el valor del salario promedio de la empresa).

¿Pero cómo lo unimos? En el punto 6 estaba claro que la unión debía hacerse por el campo DEPARTAMENTO, ¿y ahora? Puesto que necesitamos que esa información se pegue en todas las filas que representan a nuestros empleados, sin especificar ningún tipo de condición, ¿qué operación conocemos que realice esto?

Efectivamente, tenemos que hacer un CROSS JOIN:

```
SELECT nombre,departamento,salario,salario_promedio_departamento,  
salario_promedio_empresa  
FROM tabla4,tabla3
```

8

Agrupemos entonces todas las queries en una sola:

```
SELECT  
nombre,  
departamento,  
salario - salario_promedio_departamento salario_vs_departamento,  
salario - salario_promedio_empresa salario_vs_empresa  
FROM (  
    SELECT nombre,departamento,salario,salario_promedio_departamento  
    FROM tabla_empleados  
    JOIN ( SELECT departamento, AVG(salario) salario_promedio_departame  
          FROM tabla_empleados  
          GROUP BY departamento  
        ) tabla2  
    ON tabla_2.departamento = tabla_empleados.departamento  
) tabla4, (SELECT AVG(salario) salario_promedio_empresa  
            FROM tabla_empleados) tabla3
```

Seguro que en estos momentos te cuesta hasta leer la consulta. Crea la tabla de prueba, copia esta query y juega con ella: ejecútala, separa las subqueries y ejecútalas individualmente. Comprender cómo funciona esta query te va a permitir abrir tu mente.

Conclusiones

 Edix Educación

En este fastbook hemos seguido trabajando y conociendo el lenguaje DML, en concreto con las agregaciones de datos y las subqueries.

Entre lo más relevante que hemos aprendido...

- Hemos visto las principales funciones de agregado que podemos usar en SQL: MAX, MIN, AVG, SUM y COUNT.
- Hemos conocido el comando GROUP BY y las posibilidades que nos ofrece al poder realizar las agrupaciones por cualquier columna de nuestras tablas.
- Hemos utilizado el comando HAVING para realizar los filtrados sobre los conjuntos agregados, y hemos visto las similitudes entre este comando y WHERE.
- También hemos repasado las ventajas que nos ofrece el uso de subqueries para poder obtener resultados de mayor complejidad, así como las distintas formas en las que las podemos usar, por ejemplo, EXISTS, ANY y ALL.

Después de esto, ya estamos listos para empezar nuestra aventura con el uso de SQL, ya que hemos visto lo principal y lo más usado en nuestro día a día. Existen otros comandos o funcionalidades, como el uso de cálculos de ventana, la creación de funciones personalizadas o el uso de extensiones, pero este contenido queda fuera del temario.

Ejercicios propuestos

 Edix Educación

Te propongo los siguientes ejercicios resueltos (código + resultado) para que practiques.

Ejercicio 1

¿Cuál es el nombre, tipología y provincia de la tienda que vende más?

Ejercicio 2

¿Cuántas tiendas tienen unas ventas mayores a 5500€?

Nota. Hay que dar dos tablas como respuesta, la primera el listado de tiendas (id_tienda) que cumplen ese requisito ordenado por importe de venta (mayor a menor) y la segundo el número de tiendas.

Ejercicio 3

Lista por tipo de tarjeta, la fecha de creación más antigua y la más reciente.

Ejercicio 4

Ventas acumuladas y número de tickets por provincia para tiendas de tipo Hipermercado, considerando solo ventas que se hayan pagado con tarjeta de crédito.

Ejercicio 5

¿Existe alguna caja de una tienda que haya registrado en un mismo día tres tickets?, ¿y dos tickets? Responde con el número de casos. Si existe algún caso, listar id_tienda, caja y fecha ordenado por id_tienda, fecha.

Ejercicio 6

Queremos saber cuántas tiendas no tienen venta en la fecha “2020-10-30”. Usa el comando EXISTS (pista: realmente NOT EXISTS).

Solución a los ejercicios

X Edix Educación

Ejercicio 1

Solución:

```
SELECT nombre, provincia, tipología, importe_tienda
FROM tienda
JOIN (
    SELECT id_tienda, SUM(importe) importe_tienda
    FROM ticket
    GROUP BY id_tienda
) importes_tienda
ON tienda.id = importes_tienda.id_tienda
ORDER BY importe_tienda DESC
LIMIT 1
```

Resultado:

nombre character varying (60)	provincia character varying (60)	tipología character varying (50)	importe_tienda double precision
Sevilla I	Sevilla	Hipermercado	6101.37

Ejercicio 2

Respuestas:

```
SELECT id_tienda,SUM(importe) importe_tienda
FROM ticket
GROUP BY id_tienda
HAVING SUM(importe) > 5500
ORDER BY SUM(importe) DESC
```

```
SELECT count(*)
FROM (
    SELECT id_tienda,SUM(importe) importe_tienda
    FROM ticket
    GROUP BY id_tienda
    HAVING SUM(importe) > 5500
) tiendas_por_importe
```

Resultado:

id_tienda integer	importe_tienda double precision	count bigint	5
30001	6101.37		
30036	5973.85		
30079	5742.88		
30064	5696.45		
30071	5683.3		

Ejercicio 3

Respuesta:

```
SELECT tipo_tarjeta,MIN(fecha_creacion),MAX(fecha_creacion)
FROM tarjeta
GROUP BY tipo_tarjeta
```

Resultado:

tipo_tarjeta character varying (10)	min date	max date
DEBITO	2019-01-01	2020-12-30
CREDITO	2019-01-01	2020-12-30

Ejercicio 4

Respuesta:

```
SELECT provincia,SUM(importe) venta_total,COUNT(*) numero_tickets
FROM ticket t
JOIN tienda ti ON ti.id = t.id_tienda
JOIN tarjeta ta ON ta.id = t.id_tarjeta
WHERE tipologia = 'Hipermercado'
AND tipo_tarjeta = 'CREDITO'
GROUP BY provincia
ORDER BY provincia
```

Resultado:

	provincia character varying (60)	venta_total double precision	numero_tickets bigint
1	Baleares	1190.99	18
2	Barcelona	1602.8	27
3	Granada	1255.46	23
4	Huelva	3459.7	64
5	Madrid	1022.19	18
6	Málaga	5669.38	95
7	Segovia	2491.31	45
8	Sevilla	3086.24	49
9	Tarragona	2977.67	51

Ejercicio 5

Respuestas:

```
SELECT COUNT(*)
FROM (
    SELECT id_tienda, fecha, caja
    FROM ticket
    GROUP BY id_tienda, fecha, caja
    HAVING COUNT(*) = 3
) a
```

```
SELECT COUNT(*)
FROM (
    SELECT id_tienda, fecha, caja
    FROM ticket
    GROUP BY id_tienda, fecha, caja
    HAVING COUNT(*) = 2
) a
```

Resultados:

Tres tickets:

	count bigint
1	0

Dos tickets:

	count bigint
1	30

	id_tienda integer	fecha date	caja integer
1	30001	2020-07-18	4
2	30002	2020-12-01	16
3	30004	2020-09-18	18
4	30007	2020-09-05	8
5	30013	2020-06-11	20
6	30015	2020-08-26	15
7	30018	2020-06-21	2
8	30019	2020-10-11	10
9	30024	2020-12-12	17
10	30024	2020-12-25	5
11	30025	2020-05-13	20
12	30029	2020-02-01	19
13	30033	2020-03-30	16
14	30043	2020-01-09	6
15	30043	2020-02-21	10
16	30046	2020-03-13	18
17	30049	2020-06-16	19
18	30052	2020-05-10	5
19	30057	2020-05-24	11
20	30058	2020-11-08	3
21	30059	2020-08-24	16
22	30059	2020-12-27	13
23	30062	2020-11-03	18
24	30065	2020-10-24	2
25	30067	2020-03-21	8
26	30068	2020-04-03	2
27	30070	2020-04-01	19
28	30071	2020-08-15	5
29	30075	2020-01-18	15
30	30077	2020-04-29	10

Ejercicio 6

Respuesta:

```
SELECT COUNT(*)
FROM tienda
WHERE NOT EXISTS(
    SELECT 1
    FROM ticket
    WHERE
        tienda.id = ticket.id_tienda AND
        fecha = '2020-10-30'
    GROUP BY id_tienda
)
```

Enhorabuena, ¡fastbook superado!

edix

Creamos Digital Workers