SQL: Manipulación de datos (cont.)

Consultas Avanzadas

Las consultas avanzadas en SQL representan un nivel superior en la manipulación y análisis de datos dentro de bases de datos relacionales, permitiendo a los usuarios resolver problemas complejos y optimizar el desempeño de las operaciones. Estas consultas incluyen técnicas como subconsultas anidadas, uso de funciones de ventana, operaciones recursivas, combinaciones avanzadas (joins), y manipulación de datos jerárquicos.

Los estándares SQL a partir de SQL-92 permiten que la tabla en la que se realiza una operación SELECT sea una tabla virtual, en lugar de solo una tabla base. Esto significa que un DBMS debe permitir que se utilice una operación SELECT completa (en otras palabras, una **subconsulta**) en una cláusula FROM para preparar la tabla en la que se realizará el resto de la consulta.

Las expresiones que crean tablas para su uso en instrucciones SQL de esta manera se conocen como constructores de tablas. Nota: Cuando se unen tablas en la cláusula FROM, en realidad se está generando una fuente para una consulta “en vuelo” (on the fly). Por ejemplo la siguiente consulta lista los ISBN de los líbros que fueron comprados por los clientes 6 y 10:

```
SELECT isbn, nombre, apellido
FROM volumen v JOIN (SELECT nombre, apellido, id_venta
                     FROM venta t JOIN cliente c
                         ON (t.nro_cliente=c.nro_cliente)
                     WHERE c.nro_cliente = 6
                        OR c.nro_cliente = 10)  x
                ON v.id_venta = x.id_venta;
```

Lo marcado con celeste es la **subconsulta**, es una selección de los registros que forman parte de la cláusula FROM. Esto obliga al procesador de comandos SQL a realizar la subconsulta antes de realizar el JOIN en la consulta externa. Aunque esta consulta podría escribirse de otra manera, el uso de la subconsulta en la cláusula FROM le da a un programador un control adicional sobre el orden en el que se realizan las operaciones.

Una subconsulta (o subselección) es una sentencia SELECT completa incrustada dentro de otra SELECT. El resultado de la sentencia SELECT interna se convierte en datos utilizados por la sentencia SELECT externa.

Las subconsultas tienen otros usos además de evitar JOINs, una consulta que contiene una subconsulta tiene la siguiente forma general:

```
SELECT columna(s)
FROM tabla
WHERE operador (SELECT columna(s))
                FROM tabla
                WHERE …);
```

Existen dos tipos generales de subconsultas. En una **subconsulta no correlacionada**, se completa el procesamiento de la consulta SELECT interna antes de pasar a la externa y en una **subconsulta correlacionada**, no es posible completar la consulta interna sin información de la externa.

Las subconsultas correlacionadas generalmente requieren que la consulta SELECT interna se realice más de una vez y, por lo tanto, pueden ejecutarse con relativa lentitud. No ocurre lo mismo con las subconsultas no correlacionadas, que se pueden utilizar para reemplazar la sintaxis de JOIN y, por lo tanto, pueden producir un rendimiento más rápido.

Operador IN

Por ejemplo si tomemos la siguiente consulta, donde recuperar la fecha de venta y número de cliente de los que compraron el libro cuyo ISBN es 978-1-11111-136-1:

```
SELECT fecha_venta, nro_cliente
FROM venta v JOIN volumen l
    ON (v.id_venta = l.id_venta)
WHERE isbn = '978-1-11111-136-1';
```

Es posible plantearlo con una subconsulta:

```
SELECT fecha_venta, nro_cliente
FROM venta v
WHERE id_venta IN (SELECT id_venta
                   FROM volumen l
                   WHERE ISBN = '978-1-11111-136-1');
```

El SELECT interno recupera datos de la tabla de volumen y produce un conjunto de identificadores de venta. A continuación, el SELECT externo recupera datos de la venta donde el identificador de venta está en el conjunto de valores recuperados por la subconsulta.

La única diferencia es que en lugar de colocar el conjunto de valores entre paréntesis como literales, el conjunto se genera mediante un SELECT. Al procesar esta consulta, el DBMS nunca une las dos tablas. Primero realiza el SELECT interno y luego utiliza la tabla de resultados de esa consulta al procesar el SELECT externo. En

el caso en el que las dos tablas sean muy grandes, esto puede acelerar significativamente el procesamiento de la consulta.

Operador ANY

Al igual que IN, el operador ANY busca un conjunto de valores. En su forma más simple, ANY es equivalente a IN:

```
SELECT fecha_venta, nro_cliente
FROM venta v
WHERE id_venta = ANY (SELECT id_venta
                      FROM volumen l
                      WHERE ISBN = '978-1-11111-136-1');
```

Esta sintaxis le indica al DBMS que recupere registros de la tabla venta donde el id de venta sea "igual a cualquiera" de los recuperados por el SELECT en la subconsulta.

Lo que diferencia a ANY de IN es que el = se puede reemplazar con cualquier otro operador de relación (por ejemplo, < o >).

Operador UNION

Existen operaciones adicionales que es posible realizar en las tablas que, por ejemplo, responden a preguntas como "muéstrame los datos que no son…" o "muéstrame la combinación de datos que son…".

La unión es una de las pocas operaciones de álgebra relacional cuyo nombre se puede usar en una consulta SQL. Cuando desea utilizar una unión, escriba dos instrucciones SELECT individuales, unidas por la palabra clave UNION:

```
SELECT columna(s)
FROM tabla(s)
WHERE predicado
UNION
SELECT columna(s)
FROM tabla(s)
WHERE predicado;
```

Las columnas recuperadas por los dos SELECT deben tener los mismos tipos de datos y tamaños y estar en el mismo orden. Esto anterior se denomina "ser unión compatible". Las tablas de origen de los dos SELECT no necesitan ser las mismas, ni las columnas necesitan tener los mismos nombres.

```sql
SELECT nombre, apellido
FROM cliente c JOIN venta v ON (c.nro_cliente = v.nro_cliente)
    JOIN volumen l ON (l.id_venta = v.id_venta)
WHERE isbn = '978-1-11111-128-1'
UNION
SELECT nombre, apellido
FROM cliente c JOIN venta v ON (c.nro_cliente = v.nro_cliente)
    JOIN volumen l ON (l.id_venta = v.id_venta)
WHERE isbn = '978-1-11111-143-1';
```

Un uso típico de UNION es el reemplazo de un predicado por un OR. El DBMS procesa la consulta realizando dos SELECT. Luego combina las dos tablas de resultados individuales en una, eliminando los registros duplicados. Para eliminar los duplicados, el DBMS ordena la tabla de resultados por cada columna de la misma y luego la escanea en busca de registros coincidentes ubicados uno debajo del otro. (Es por eso que los registros del resultado están en orden alfabético por el nombre del autor).

Operador EXCEPT

Entre las consultas de bases de datos más potentes se encuentran aquellas formuladas en forma negativa, como por ejemplo "*muéstrame todos los clientes que no han realizado ninguna compra durante el año pasado*".

Este tipo de consulta es particularmente complicada porque de todos los clientes debe "restarle" los clientes que sí hicieron compras el año pasado, entonces el DBMS debería utilizar una operación de diferencia.

Una forma de realizar una consulta que requiere una diferencia es utilizar la sintaxis de subconsulta con el operador NOT IN. Por ejemplo considere una consulta que recupere todos los libros que no están en stock:

```sql
SELECT isbn, titulo, id_editorial, edicion
FROM libro l JOIN titulo t ON (l.nro_trabajo = t.nro_trabajo)
WHERE isbn NOT IN (SELECT isbn
                   FROM volumen
                   WHERE id_venta IS NULL);
```

La consulta externa recupera una lista de todas las cosas de interés; la subconsulta recupera aquellas que cumplen los criterios necesarios. El operador NOT IN actúa entonces para incluir todos aquellos de la lista de todas las cosas que no están en el conjunto de valores devueltos por la subconsulta.

El estándar SQL-92 agregó un operador, EXCEPT, que realiza una operación de diferencia directamente entre dos tablas que son **unión compatible**. Las consultas que utilizan EXCEPT se parecen mucho a un UNION:

```sql
SELECT isbn, titulo, id_editorial, edicion
FROM libro l JOIN titulo t ON (l.nro_trabajo = t.nro_trabajo)
EXCEPT
SELECT l.isbn, titulo, id_editorial, edicion
FROM libro l JOIN titulo t ON (l.nro_trabajo = t.nro_trabajo)
            JOIN volumen v ON (l.isbn = v.isbn)
WHERE id_venta IS NULL;
```

En el ejemplo anterior se incluyen dos sentencias SELECT completas unidas por la palabra clave EXCEPT; cada consult debe devolver tablas que sean unión compatibles. El primer SELECT recupera una lista de todas las cosas (en este ejemplo, todos los libros); el segundo recupera las cosas que son (en este ejemplo, los libros que hay en stock, o sea que no se han vendido - id_venta nulo). Luego, el operador EXCEPT elimina todos los registros de la primera tabla que aparecen en la segunda.

Operador EXISTS

El operador EXISTS comprueba la cantidad de registros devueltos por una subconsulta. Si la subconsulta contiene uno o más registros, el resultado es verdadero y se coloca un registro en la tabla de resultados; de lo contrario, el resultado es falso entonces no se agrega ningun registro a la tabla de resultados.

Por ejemplo, supongamos que la librería en línea desea ver los títulos de los libros que se han vendido. Para escribir la consulta utilizando EXISTS, debe utilizar:

```sql
SELECT titulo
FROM libro l JOIN titulo t ON (l.nro_trabajo = t.nro_trabajo)
WHERE EXISTS (SELECT 1
              FROM volumen v
              WHERE l.isbn = v.isbn
                AND id_venta IS NOT NULL);
```

La anterior es una **subconsulta correlacionada**. En lugar de completar toda la subconsulta y luego pasar a la consulta externa, el DBMS procesa la consulta de la siguiente manera:

1. Lee un registro de la tabla libro.
2. Utiliza el ISBN de ese registro en la cláusula WHERE de la subconsulta.
3. Si la subconsulta encuentra al menos un registro en la tabla volume con el mismo ISBN vendido (con un valor en id_venta), coloca el registro en la tabla de resultados intermedios; sino, no hace nada.
4. Repite los pasos 1 a 3 para todos los registros en la tabla libro.
5. Hace el JOIN con la tabla de resultados intermedios.
6. Proyecta la columna titulo.

Lo importante que hay que reconocer aquí es que el DBMS **repite la subconsulta para cada registro** de la tabla libro. Lo que hace que sea una **subconsulta correlacionada** es la condición del WHERE que utiliza el valor del campo ISBN de la consulta externa en el WHERE de la consulta interna.

Como puede ver la subconsulta interna tiene un **SELECT 1**, ésto es porque cuando se utiliza el operador EXISTS, no importa lo que siga a SELECT en la subconsulta, EXISTS simplemente comprueba si hay registros presentes

en la tabla de resultados de la subconsulta. Por lo tanto, es más fácil simplemente utilizar cualquier caracter en lugar de especificar columnas individuales. ¿Cómo funcionará esta consulta? Probablemente funcionará mejor que una consulta que une libro y volume, especialmente si las dos tablas son grandes. Si escribiera la consulta utilizando una subconsulta IN:

```
SELECT titulo
FROM libro l JOIN titulo t ON (l.nro_trabajo = t.nro_trabajo)
WHERE isbn IN (SELECT isbn
               FROM volumen v
               WHERE id_venta IS NOT NULL);
```

En la consulta anterior se está utilizando una subconsulta **no correlacionada** que devuelve un conjunto de ISBN que la consulta externa busca. Cuanto más registros devuelve la subconsulta no correlacionada la performance del EXISTS es similar a la del IN. Sin embargo, si la subconsulta no correlacionada devuelve solo algunos registros, probablemente tendrá un mejor rendimiento que la consulta que contiene la subconsulta correlacionada.

Operador INTERSECT

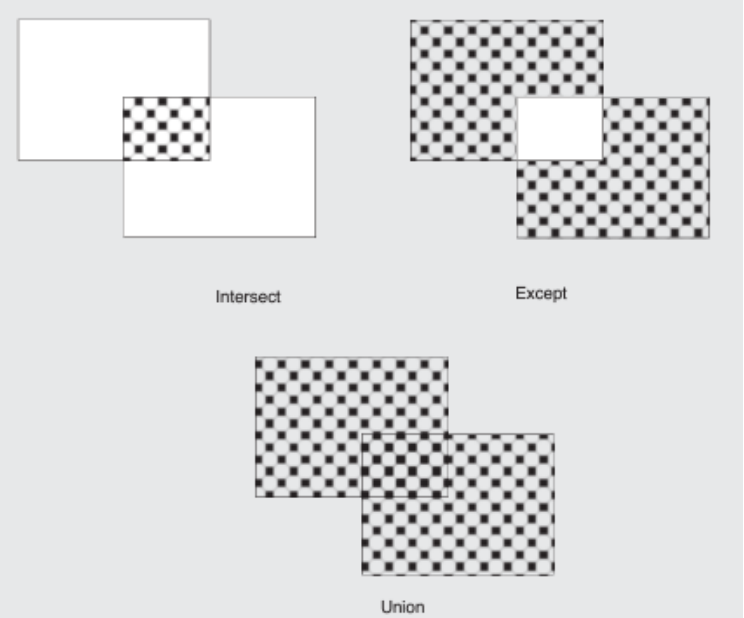
INTERSECT devuelve todos los registros que las dos consultas tienen en común, al igual que los operadores UNION y EXCEPT los resultados de cada consulta deben ser una unión compatible. En la mayoría de los casos, las dos tablas de origen se generan mediante una operación SELECT.

Como primer ejemplo, preparemos una consulta que incluya todos los clientes de la librería, excepto aquellos que hayan realizado compras con un costo total de más de $500. Una forma de escribir esta consulta es:

```
SELECT nro_cliente, nombre, apellido
FROM cliente
EXCEPT
SELECT nro_cliente, nombre, apellido
FROM cliente c JOIN venta v
      ON (c.nro_cliente = v.nro_cliente)
WHERE monto_total_venta > 500;
```

Aquí la consulta devuelve los no hayan realizado compras superiores a $500. En cambio si realizamos las mismas consultas con INTERSECT obtendremos aquellos clientes que sí realizaron compras superiores a $500.

```sql
SELECT nro_cliente, nombre, apellido
FROM cliente
INTERSECT
SELECT c.nro_cliente, nombre, apellido
FROM cliente c JOIN venta v
      ON (c.nro_cliente = v.nro_cliente)
WHERE monto_total_venta > 500;
```



Operando con caracteres

Concatenación

El estándar básico de SQL contiene un operador y varias funciones para manejar cadenas de caracteres, el operador de concatenación —||— pega una cadena al final de otra. Puede utilizarse para formatear una salida, por ejemplo, la librería podría obtener una lista alfabética de nombres de clientes formateada:

```sql
SELECT nombre || ', ' ||
       apellido AS Nombre_Apellido
FROM cliente
ORDER BY apellido, nombre;
```

nombre_apellido
Helen, Brown
Mary, Collins
Peter, Collins
Jane, Doe
John, Doe
Edna, Hayes
Franklin, Hayes
Helen, Jerry
Peter, Johnson
Peter, Johnson
Janice, Jones
Jon, Jones
Jane, Smith
Janice, Smith
John, Smith

Como se observa la concatenación incluye una cadena literal para colocar la coma y el espacio entre el apellido y el nombrey simplemente coloca una cadena al final de otra.

Mayúsculas y minúsculas

Cuando un DBMS evalúa una cadena literal contra datos almacenados, realiza una búsqueda que distingue entre mayúsculas y minúsculas. Esto significa que las letras mayúsculas y minúsculas son diferentes: "JONES" no es lo mismo que "Jones".

Es posible evitar estos problemas utilizando las funciones UPPER y LOWER para convertir los datos almacenados a un solo caso. Por ejemplo, suponga que alguien en la librería no está seguro de cómo se hn gusrdado los nombres de los clientes. Para realizar una búsqueda que no distinga entre mayúsculas y minúsculas de los clientes con un apellido específico, la persona podría utilizar:

```
SELECT nombre || ', ' ||
       apellido AS Nombre_Apellido
FROM cliente
WHERE UPPER(apellido) = 'SMITH';
```

Es posible lograr el mismo resultado utilizando LOWER en lugar de UPPER

Otras funciones de caracteres (TRIM y SUBSTRING)

La función TRIM elimina los caracteres iniciales y/o finales de una cadena. Es posible colocar TRIM en cualquier expresión que contenga una cadena. La función SUBSTRING extrae partes de una cadena. En https://www.postgresql.org/docs/9.1/functions-string.html es posible ampliar muchas otras funcions para el manejo de caracteres.

Operando con fechas y horas

Los DBMS de SQL proporcionan tipos de datos para fechas y horas. Cuando se almacenan datos utilizando estos tipos, es posible que SQL realice operaciones cronológicas sobre esos valores. Puede, por ejemplo, restar

dos fechas para averiguar el número de días entre ellas o agregar un intervalo a una fecha para adelantarla una cantidad específica de días. El SQL estándar especifica cuatro tipos de datos que se relacionan con fechas y horas:

- DATE: fecha únicamente
- TIME: hora solamente
- TIMESTAMP: Combinación de fecha y hora
- INTERVAL: un intervalo entre dos tipos de datos (cualquiera de los anteriores)

SQL también permite recuperar la fecha y la hora actual con las siguientes funciones:

- CURRENT_DATE: Devuelve la fecha actual
- CURRENT_TIME: Devuelve la hora actual
- CURRENT_TIMESTAMP: Devuelve la fecha y la hora actual

El operador EXTRACT extrae una parte de una fecha y/o hora. Tiene el siguiente formato general:

EXTRACT (datetime_field FROM datetime_value)

Por ejemplo, la consulta

SELECT EXTRACT (YEAR FROM CURRENT_DATE);

devuelve el año actual. Además de los campos de fecha y hora, EXTRACT también puede proporcionar el día de la semana (DOW) y el día del año (DOY).

Para ampliar las particularidades del manejo de fechas y horas en Postgresql ver https://www.postgresql.org/docs/16/functions-datetime.html

Otros operadores

La expresión SQL CASE, al igual que un CASE en un lenguaje de programación de propósito general, permite que una instrucción SQL elija entre una variedad de acciones en función de la veracidad de las expresiones lógicas. Al igual que las operaciones aritméticas y de cadenas, la instrucción CASE genera un valor que se mostrará y, por lo tanto, forma parte de la cláusula SELECT. La expresión CASE tiene la siguiente sintaxis general:

```
CASE
    WHEN logical condition THEN action
    WHEN logical condition THEN action
    :
    :
    ELSE default action
END
```

El CASE no necesariamente necesita ser el último elemento en la cláusula SELECT. La palabra clave END puede ir seguida de una coma y otras columnas o cantidades calculadas.

Como ejemplo, si la librería desease ofrecer descuentos a los usuarios en función del precio de un libro. Cuanto mayor sea el precio de venta del libro, mayor será el descuento. Para incluir el precio con descuento en el resultado de una consulta, es posible utilizar:

```
SELECT isbn, precio_venta,
       CASE
           WHEN precio_venta < 50
                THEN precio_venta * .95
           WHEN precio_venta < 75
               THEN precio_venta * .9
           WHEN precio_venta < 100
               THEN precio_venta * .8
           ELSE precio_venta * .75
        END AS precio_con_descuento
FROM volumen;
```

La consulta anterior muestra el ISBN y el precio de venta del libro. Luego evalúa la primera expresión CASE después de WHEN. Si esa condición es verdadera, la consulta realiza el cálculo, muestra el precio con descuento y sale de CASE. Si la primera condición es falsa, la consulta pasa al segundo WHEN, y así sucesivamente. Si ninguna de las condiciones es verdadera, la consulta ejecuta la acción después de ELSE. (El ELSE es opcional). La columna calculada es renombrada como precio_con_descuento.

isbn	precio_venta	precio_con_descuento
978-1-11111-137-1	80.00	64.000
978-1-11111-137-1	50.00	45.000
978-1-11111-136-1	75.00	60.000
978-1-11111-136-1	50.00	45.000
978-1-11111-121-1	110.00	82.5000
978-1-11111-124-1	75.00	60.000
978-1-11111-141-1	24.95	23.7025
978-1-11111-141-1	24.95	23.7025
978-1-11111-141-1	24.95	23.7025
978-1-11111-145-1	27.95	26.5525
978-1-11111-145-1	27.95	26.5525
978-1-11111-145-1	27.95	26.5525
978-1-11111-130-1	50.00	45.000
978-1-11111-132-1	17.95	17.0525
978-1-11111-138-1	75.95	60.760
978-1-11111-138-1	75.95	60.760
978-1-11111-131-1	15.95	15.1525
978-1-11111-140-1	25.95	24.6525
978-1-11111-123-1	24.95	23.7025
978-1-11111-127-1	27.95	26.5525

Operando con Grupos de Registros

Las consultas que se han visto hasta ahora en su mayoría operan de a un registro a la vez. Sin embargo, SQL también incluye una variedad de palabras clave y funciones que funcionan en grupos de registros, ya sea una tabla completa o un subconjunto de una tabla.

Funciones de Agregación

Las funciones de conjunto o de agregación básicas de SQL calculan una variedad de medidas basadas en valores de una columna en varios registros. El resultado de usar una de estas funciones de conjunto da como resultado una columna calculada (que aparece solo en el resultado).

La sintaxis básica para una función de conjunto es:

Nombre_función (argumento_entrada)

Se debe colocar la llamada de función después de SELECT, tal como lo haría con un cálculo aritmético. Lo que use como argumento de entrada depende de la función que esté usando. En la mayoría de los casos, es posible que el DBMS admita COUNT, SUM, AVG, MIN y MAX. Además, muchos DBMS proporcionan funciones agregadas adicionales para medidas como la desviación estándar y la varianza.

Información adicional es posible encontrarla en la documentación de Postgresql en:

https://www.postgresql.org/docs/16/functions-aggregate.html

COUNT

La función COUNT es algo diferente de otras funciones de conjunto de SQL en que, en lugar de realizar cálculos basados en valores de datos, cuenta el número de registros de una tabla. Para utilizarla, se coloca COUNT (*) en la consulta. Para éste caso el argumento de entrada de COUNT es un asterisco:

```
SELECT COUNT (*)
FROM volumen;
```

Por ejemplo la siguiente consulta:

```
SELECT COUNT (*)
FROM volumen
WHERE isbn = '978-1-11111-141-1';
```

muestra que la librería ha tenido o tiene en existencia 7 libros cuyo ISBN es 978-1-11111-141-1. Si quisiéramos saber cuántos libros hay en existencia del mismo ISBN deberíamos agregar:

```
SELECT COUNT (*)
FROM volumen
WHERE isbn = '978-1-11111-141-1'
AND id_venta IS NULL;
```

Cuando se utiliza * como parámetro de entrada para la función COUNT, el DBMS incluye todas los registros. Sin embargo, si desea excluir registros que tienen valores nulos en una columna en particular, puede utilizar el nombre de la columna como parámetro de entrada. Por ejemplo sabemos que hay 71 libros en la tabla volumen (algunos ya vendidos), entonces para averiguar cuántos volúmenes se han vendido la librería podría hacer la siguiente consulta:

```
SELECT COUNT (id_venta)
FROM volumen;
```

Entonces se ve que se han vendido 42 libros, ¿ahora bien, en cuantas ventas se han distribuído esos libros?, se debería colocar en cuantas ventas diferents, entonces

```
SELECT COUNT (DISTINCT id_venta)
FROM volumen;
```

Se debe utilizar COUNT colocando la palabra clave DISTINCT delante del nombre de la columna utilizada como parámetro de entrada, vemos que se han realizado 20 ventas diferentes.

SUM

Si alguien en la librería quisiera saber el monto total de una venta para poder insertar ese valor en la tabla de ventas, entonces la forma más fácil de obtener dicho valor es sumar los valores en la columna precio_vendido, por ejemplo para la venta cuyo id es 6:

```sql
SELECT SUM (precio_vendido)
FROM volumen
WHERE id_venta = 6;
```

AVG

La función AVG calcula el valor promedio de una columna. Por ejemplo, para encontrar el precio de venta promedio de los libros en la librería, se podría usar una consulta como:

```sql
SELECT AVG(precio_venta)
FROM volumen;
```

MIN y MAX

Las funciones MIN y MAX devuelven los valores mínimo y máximo de una columna o expresión. Por ejemplo, para ver el precio máximo de un libro (el más caro) o mínimo (el más barato), alguien en la librería podría usar una consulta como:

```sql
SELECT MIN (precio_venta) as minimo,
       MAX(precio_venta) as maximo
FROM volumen;
```

Funciones de Agregación en Predicados

Las funciones de conjunto también se pueden utilizar en predicados WHERE para generar valores con los que se puedan comparar los datos almacenados. Supongamos, por ejemplo, que alguien en la librería quiere ver los títulos y el precio al que se vendieron los libros que costaron más que el promedio de precios vendidos de todos los libros. La estrategia para preparar esta consulta es utilizar una subconsulta que devuelva promedio de precios vendidos de todos los libros y lo compare con el precio al que fue vendido cada cada libro de la tabla de volumen:

```sql
SELECT titulo, precio_vendido
FROM titulo t JOIN libro l ON (t.nro_trabajo = l.nro_trabajo)
    JOIN volumen v ON (l.isbn = v.isbn)
WHERE precio_vendido > (SELECT AVG (precio_vendido)
                        FROM volumen);
```

La subconsulta marcada en celeste devuelve el promedio (68.2313953488372093), el cual es comparado con cada uno de los valores que posee cada registro en la columna precio_vendido.

Consultas de Agrupación

SQL puede agrupar registros en función de los valores coincidentes en columnas específicas y valores resumidos para cada grupo. Cuando estas consultas de agrupación se combinan con las funciones de conjunto, SQL puede proporcionar informes simples sin necesidad de programación extra.

Formando Grupos

Para formar un grupo, se agrega una cláusula GROUP BY a una declaración SELECT, seguida de las columnas cuyos valores se utilizarán para formar los grupos.

Todas los registros cuyos valores coincidan con esas columnas se colocarán en el mismo grupo. Por ejemplo, si alguien en la librería quiere ver cuántas copias de cada edición del libro se han vendido, puede utilizar la siguiente consulta:

```
SELECT isbn, COUNT(*)
FROM volumen
GROUP BY isbn
ORDER BY isbn;
```

isbn	count
978-1-11111-111-1	1
978-1-11111-115-1	1
978-1-11111-121-1	3
978-1-11111-122-1	1
978-1-11111-123-1	2
978-1-11111-124-1	1
978-1-11111-125-1	1
978-1-11111-126-1	3
978-1-11111-127-1	5
978-1-11111-128-1	1

La consulta forma grupos haciendo coincidir los ISBN. Muestra el ISBN y el número de registros en cada grupo. Hay una restricción importante que debe observar con una consulta de agrupación: **puede mostrar valores solo de las columnas que se utilizan para formar los grupos**. Como ejemplo, si alguien en la librería desea ver el número de copias de cada título que se han vendido. Alguien tentado podrí escribir la siguiente consulta:

```
SELECT titulo, COUNT (*)
FROM volumen v JOIN libro l ON (v.isbn = l.isbn)
   JOIN titulo t ON (l.nro_trabajo = t.nro_trabajo)
GROUP BY titulo
ORDER BY titulo;
```

Pero los títulos pueden duplicarse (se duplican) entonces vemos que los títulos no se repiten pero son diferentes libros (distintos ISBN), se debe agrupar por el ISBN y título. El DBMS formará entonces grupos que tengan los mismos valores en ambas columnas. Solo hay un título por número de obra, además de obtener la capacidad de mostrar el título al agrupar por el ISBN. La consulta debería escribirse:

```
SELECT l.isbn, titulo, COUNT (*)
FROM volumen v JOIN libro l ON (v.isbn = l.isbn)
   JOIN titulo t ON (l.nro_trabajo = t.nro_trabajo)
GROUP BY l.isbn, titulo
ORDER BY titulo;
```

A veces es necesario utilizar varias columnas para crear grupos anidados. Por ejemplo, si se desea ver el costo total de las compras realizadas por cada cliente por día, la consulta podría escribirse:

```
SELECT c.nro_cliente,
       fecha_venta,
       SUM(monto_total_venta)
FROM cliente c JOIN venta v
    ON (c.nro_cliente = v.nro_cliente)
GROUP BY c.nro_cliente, fecha_venta;
```

Debido a que la columna nro_cliente se incluye en primer lugar en la cláusula GROUP BY, sus valores se utilizan para crear las agrupaciones externas. A continuación, el DBMS agrupa los pedidos por fecha dentro de los números de cliente.

Si bien el resultado es algo difícil de interpretar porque las agrupaciones externas (las de cliente) no están en orden, si se le agrega la cláusula ORDER BY para ordenar la salida por número de cliente, es posible ver el orden por fecha dentro de cada cliente.

nro_cliente	fecha_venta	sum
2	2013-07-25	130.00
11	2013-07-10	125.00
4	2013-06-30	110.00
8	2013-07-07	50.00
12	2013-07-05	505.00
1	2013-05-29	510.00
6	2013-09-01	95.00
2	2013-07-10	25.95
11	2013-07-12	75.00
2	2013-09-01	75.00
5	2013-08-22	100.00
8	2013-07-05	80.00
1	2013-06-05	125.00
10	2013-07-10	200.00
6	2013-07-10	80.00
1	2013-07-25	100.00
6	2013-06-30	110.00
5	2013-07-07	90.00
1	2013-06-15	58.00
9	2013-07-10	200.00

nro_cliente	fecha_venta	sum
1	2013-05-29	510.00
1	2013-06-05	125.00
1	2013-06-15	58.00
1	2013-07-25	100.00
2	2013-07-10	25.95
2	2013-07-25	130.00
2	2013-09-01	75.00
4	2013-06-30	110.00
5	2013-07-07	90.00
5	2013-08-22	100.00
6	2013-06-30	110.00
6	2013-07-10	80.00
6	2013-09-01	95.00
8	2013-07-05	80.00
8	2013-07-07	50.00
9	2013-07-10	200.00
10	2013-07-10	200.00
11	2013-07-10	125.00
11	2013-07-12	75.00
12	2013-07-05	505.00

Restringiedo Grupos

Las consultas de agrupación que se han presentado hasta aquí incluyen todos los registros del rupo. Sin embargo, puede restringirse los registros que se incluyen en la salida agrupada utilizando una de dos estrategias:

- Restringir las registros antes de que se formen los grupos (con la cláusula WHERE)
- Permitir que se formen todos los grupos y luego restringir los grupos (con la cláusula HAVING)

La cláusula HAVING contiene un predicado que se aplica a los grupos después de que se forman.

Por ejemplo si en lugar de ver todas las compras realizadas por cada cliente por día, sólo deseamos listar las que sean iguales o superiores a $100

```sql
SELECT c.nro_cliente,
       fecha_venta,
       SUM(monto_total_venta)
FROM cliente c JOIN venta v
    ON (c.nro_cliente = v.nro_cliente)
GROUP BY c.nro_cliente, fecha_venta
HAVING SUM(monto_total_venta) >= 100;
```