Tu guía intermedia para SQL

A medida que te sientas más a gusto con SQL, podrás realizar consultas incluso más avanzadas. Esta guía detallada te proporcionará una introducción más minuciosa sobre algunas de las funciones de SQL que ya
aprendiste y te brindará algunas herramientas nuevas con las que puedas trabajar. Asegúrate de guardar esta guía para poder consultar fácilmente estos consejos útiles en el futuro.

Estructura de SQL para consultas más complejas

Aprenderás más sobre estas cláusulas y expresiones en las siguientes secciones, pero primero veamos cómo podrías estructurar una consulta más compleja en SQL:

SELECT	Columna que deseas ver
FROM	Tabla donde están los datos
WHERE	Condición determinada de los datos
GROUP BY	Columna que deseas agregar por
HAVING	Condición determinada de la agregación
ORDER BY	Columna en la que deseas ordenar los resultados y en orden ASCendente o DESCendente
LIMIT	El número máximo de columnas que deseas que contengan tus resultados

Diferentes tipos de instrucciones JOIN

La mayoría de los analistas usarán instrucciones **INNER JOIN** o **LEFT JOIN** a lo largo de sus carreras profesionales. Cuando unes tablas, se combinan datos de una tabla con datos de otra tabla conectados por un campo común. Por ejemplo, digamos que tienes los colores favoritos de tus amigos en una tabla y las películas favoritas de tus amigos en otra tabla. Puedes tener sus colores y películas favoritas en una tabla al unir las dos tablas con los nombres de tus amigos, que es el campo que tienen en común. Este es tu campo JOIN. En el área de trabajo, un campo JOIN, por lo general, es alguna forma de identificación, como un id_cliente o id_cuenta.

Vista de la tabla

Colores_Favoritos	Películas_Favoritas
amigo (cadena)	amigo (cadena)
color (cadena)	película (cadena)

Vista de datos

Colores_Favoritos	
amigo	**color**
Rachel DeSantos	azul
Sujin Lee	verde
Najil Okoro	rojo
John Anderson	naranja

Películas_Favoritas	
amigo	**película**
Rachel DeSantos	Vengadores
Sujin Lee	Mi villano favorito
Najil Okoro	Frozen

Entonces, en este ejemplo, quieres usar una instrucción INNER JOIN si únicamente quieres ver información sobre amigos que tienen un color favorito y una película favorita. Esto significa que si John Anderson tiene un color favorito pero no una película favorita, John Anderson no aparecerá en los resultados. Tus amigos deben estar en ambas tablas para aparecer en tus resultados. Por lo tanto, las instrucciones INNER JOIN son útiles cuando se quieren ver los datos en los que la clave JOIN existe en ambas tablas, que es normalmente la razón por la que quieres unir conjuntos de datos en primer lugar. Generalmente, los analistas usarán instrucciones INNER JOIN la mayoría del tiempo.

```
SELECT
      friend,
      color,
      movie
FROM
      Favorite_Colors AS c
INNER JOIN
      Favorite_Movies AS m ON c.friend = m.friend
```

Resultados:

amigo	color	película
Rachel DeSantos	azul	Vengadores
Sujin Lee	verde	Mi villano favorito
Najil Okoro	rojo	Frozen

Como esta consulta utilizó una instrucción INNER JOIN, los resultados solo tienen tres de los cuatro amigos. A modo de recordatorio, esto es así porque las consultas INNER JOIN solo muestran resultados donde el campo JOIN (en este caso “amigo”) existe en ambas tablas. Dado que John Anderson no figura en la tabla de películas favoritas, queda excluido de los resultados de la consulta.

Ahora, digamos que deseas usar una instrucción LEFT JOIN para obtener la información de todos tus amigos en una tabla (por ejemplo, la tabla de colores favoritos) con datos añadidos de la otra tabla (por ejemplo, la tabla de películas favoritas) en caso de existir. Entonces, si John Anderson tiene un color favorito pero no una película favorita, de todos modos aparecerá en tus resultados. Tendrá un campo vacío (que es nulo) para su película favorita. La mayoría de las veces las instrucciones LEFT JOIN se usan si los datos que estás intentando extraer de otra tabla son opcionales. Este es un campo que es bueno tener, pero no es necesario para tu análisis, ya que puedes obtener valores nulos. En el trabajo, encontrarás que los analistas usan instrucciones LEFT JOIN con menor frecuencia que INNER JOIN.

```
SELECT
      friend,
      color,
      movie
FROM
      Favorite_Colors AS c
LEFT JOIN
      Favorite_Movies AS m ON c.friend = m.friend
```

Resultados:

amigo	color	película
Rachel DeSantos	azul	Vengadores
Sujin Lee	verde	Mi villano favorito
Najil Okoro	rojo	Frozen
John Anderson	naranja	nulo

De modo que ahora conoces la diferencia entre las instrucciones INNER JOIN y LEFT JOIN. Sabes que las instrucciones INNER JOIN serán los tipos de instrucción JOIN más usados porque suelen estar alineados con los casos de uso de la empresa.
Otra razón por la que se utilizan las instrucciones INNER JOIN es porque dan como resultado menos datos, ya que la clave de JOIN debe existir en ambas tablas. Esto significa que las consultas realizadas con las instrucciones INNER JOIN tienden a ejecutarse más rápido y usar menos recursos

que las consultas con LEFT JOIN. Esto podría no ser un problema para la mayoría de los analistas, pero si estás trabajando con tablas muy grandes que tienen más de un millón de filas y/o según el dialecto SQL usado, tu consulta podría tardar mucho más en ejecutarse si usas LEFT JOIN en lugar de INNER JOIN.

Básicamente, lo que hay que hacer es usar INNER JOIN lo más posible.

Agregadores como SUM() y COUNT()

Los agregadores resumen las filas en un solo valor. Las funciones **SUM()** y **COUNT()** son ejemplos de agregadores. Los tipos de agregadores que estarán disponibles para ti dependerán del dialecto SQL que utilice tu empresa. Pero los agregadores más usados, como SUM(), COUNT(), MAX() y MIN(), están disponibles en todos los dialectos de SQL, aunque existan algunas pequeñas diferencias. Es fácil comprobar cómo están formateados los agregadores para cualquier dialecto que estés usando. Hay muchos recursos disponibles en línea. Solo abre tu motor de búsqueda favorito y busca la función de agregación que quieres usar y tu dialecto SQL. Por ejemplo, busca “función SUM en SQL Server”. Todos los agregadores trabajan de la misma forma, así que vamos a repasar SUM() y COUNT(). La función SUM() realiza la suma de cualquier columna que escribas entre paréntesis. La función COUNT() cuenta el número de entradas en cualquier columna que escribas dentro del paréntesis. Por ejemplo, supongamos que tienes una tabla de compras con una lista de personas y el número de entradas de cine que han comprado.

La tabla de compras:

nombre	**entradas**
Rachel DeSantos	3
Sujin Lee	2
Najil Okoro	2
John Anderson	1

Consulta:

```
SELECT
      SUM(tickets) AS total_tickets,
      COUNT(tickets) AS number_of_purchases
FROM
      purchases
```

Resultado:

total_tickets	number_of_purchases
8	4

También puedes agregar una cláusula **DISTINCT** dentro de la función. Esto funcionará para la mayoría de los dialectos de SQL, pero siempre es bueno comprobar y confirmar primero que la función acepta el dialecto que usa tu empresa. Agregar una cláusula DISTINCT en la función SUM() o COUNT() te permite hacer una agregación solo en cada valor distinto del campo. Compruébalo en el siguiente ejemplo:

```
SELECT
      SUM(tickets) AS total_tickets,
      SUM(DISTINCT tickets) AS total_distinct_tickets,
      COUNT(tickets) AS number_of_purchases,
      COUNT(DISTINCT tickets) AS
      number_of_distinct_purchases
FROM
      purchases
```

Resultado:

total_tickets	total_distinct_tickets	number_of_purchases	number_of_distinct_purchases
8	6	4	3

Puedes ver que los resultados contienen números más pequeños para las columnas con DISTINCT. Esto se debe a que DISTINCT le indica a SQL que solo agregue valores únicos. Para entenderlo mejor, echa un vistazo a la segunda columna de total_entradas_distintas, que muestra cómo se puede utilizar DISTINCT con una función SUM(). Sin embargo, en este ejemplo, realmente no tiene sentido hacer una suma de valores distintos. Es probable que nunca uses DISTINCT con las funciones SUM(). En cambio, es más probable que uses DISTINCT con las funciones COUNT(), ya que es útil para identificar casos únicos.

Uso de GROUP BY con agregadores

Anteriormente, cuando aprendiste sobre SUM() y COUNT() con las compras de entradas de cine, resumiste los datos para obtener el número total de entradas compradas y el número total de compras realizadas con SUM() y COUNT(). Cuando se utilizan funciones de agregación como SUM() y COUNT() con una cláusula **GROUP BY,** los grupos se resumen en partes especificadas por la cláusula GROUP BY.

Por ejemplo, supongamos que tu tabla de compras es como la que se muestra a continuación, donde la transacción de cada persona fue para una ocasión particular. Deberías usar una cláusula GROUP BY si deseas obtener el número total de entradas vendidas y el número total de compras realizadas por el tipo de ocasión. Verás que si quieres agregar por algo (por ejemplo, la ocasión), puedes utilizar la cláusula GROUP BY. De este modo, SQL es bastante intuitivo.

La nueva tabla de compras:

ocasión	nombre	entradas
diversión	Rachel DeSantos	5
fecha	Sujin Lee	2
fecha	Najil Okoro	2
diversión	John Anderson	3

Consulta:

```
SELECT
      occasion,
      SUM(tickets) AS total_tickets,
      COUNT(tickets) AS number_of_purchases
FROM
      purchases
GROUP BY
      occasion
```

Resultados:

occasion	total_tickets	number_of_purchases
diversión	8	2
fecha	4	2

¡Excelente! Ahora ya sabes cómo utilizar la cláusula GROUP BY y cuándo querrás utilizarla. Aquí hay otra cosa interesante que debes saber: puedes usar el número de columna en la cláusula GROUP BY para especificar por qué quieres agrupar en lugar de usar los nombres de las columnas. En el último ejemplo, quisiste agrupar por ocasión. Ocasión es la primera columna escrita en la consulta SQL. Eso significa que es posible escribir GROUP BY 1 en lugar de GROUP BY ocasión. Si ocasión fuera la segunda columna de la cláusula SELECT, entonces escribirías GROUP BY 2. Ver a continuación:

Consulta:

```
SELECT
      occasion,
      SUM(tickets) AS total_tickets,
      COUNT(tickets) AS number_of_purchases
FROM
      purchases
GROUP BY
      occasion
```

Es lo mismo que:

```
SELECT
      occasion,
      SUM(tickets) AS total_tickets,
      COUNT(tickets) AS number_of_purchases
FROM
      purchases
GROUP BY
      1
```

Conocer este atajo puede ahorrarte tiempo al escribir tus consultas SQL y cuando estés agrupando por múltiples campos. En ese caso, solo hay que separarlos con comas (por ejemplo, GROUP BY 1, 2, 3, 4).

Cuándo usar HAVING

La cláusula HAVING es similar a la cláusula WHERE, ya que filtra los datos en función de determinadas condiciones. Pero estas cláusulas se usan en situaciones distintas. La cláusula **WHERE** se usa para crear filtros en la tabla, como un filtro para determinados intervalos de fechas o países específicos. La cláusula **HAVING** se usa para hacer filtros en tus agregaciones y tiene que estar emparejada con una cláusula GROUP BY.

La tabla de compras:

ocasión	**nombre**	**entradas**
diversión	Rachel DeSantos	5
fecha	Sujin Lee	2
fecha	Najil Okoro	2
diversión	John Anderson	3

En este ejemplo, verás que puedes poner capas en la cláusula HAVING si deseas establecer límites en tu agregación, o la suma y el recuento en este caso:

Consulta:

```
SELECT
      occasion,
      SUM(tickets) AS total_tickets,
      COUNT(tickets) AS number_of_purchases
FROM
      purchases
GROUP BY
      occasion
HAVING
      SUM(tickets) > 5
```

Resultados:

occasion	total_tickets	number_of_purchases
diversión	8	2

Es importante tener en cuenta que tus resultados ya no contienen la ocasión 'fecha'. Esto es porque tu cláusula HAVING filtra las sumas que son mayores que 5. La ocasión 'fecha' solo tenía 4 entradas en total, que son menos de 5, por lo que la ocasión 'fecha' no aparece en tus resultados.

¡Excelente trabajo! Ahora ya sabes cómo y cuándo usar la cláusula HAVING. Como analista de datos, utilizarás muchas cláusulas WHERE y solo algunas cláusulas HAVING. Esto se debe al caso de uso del negocio, pero también a los recursos, al igual que INNER JOIN vs. LEFT JOIN. Si tu consulta contiene una cláusula HAVING, tardará más tiempo en ejecutarse y consumirá más recursos porque SQL necesita filtrar después de ejecutar los cálculos SUM() y COUNT(). Por eso, es una buena idea intentar minimizar el uso de la cláusula HAVING siempre que sea posible. Pero, si tienes que usar HAVING, intenta hacerlo en tablas temporales.

Uso de ORDER BY para organizar los resultados

La cláusula **ORDER BY** te ayuda a organizar los resultados. Va al final de la consulta SQL y es la última cláusula que se debe usar, a menos que tengas una cláusula LIMIT.

Una versión ligeramente modificada de la tabla de compras:

nombre	entradas
Rachel DeSantos	3
Sujin Lee	5
Najil Okoro	2
John Anderson	4

Digamos que queremos que todas las personas de esta tabla estén organizadas por el número de entradas que compraron de mayor a menor, o en orden descendente.

```
SELECT
      name,
      tickets
FROM
      purchases
ORDER BY
      tickets DESC
```

Resultado:

nombre	entradas
Sujin Lee	5
John Anderson	4
Rachel DeSantos	3
Najil Okoro	2

Si quieres mostrar primero a la persona con la menor cantidad de entradas, deberías ordenar tus resultados en orden ASCendente. Para hacer eso en SQL, puedes usar ASC o dejarlo en blanco, ya que SQL ordena las columnas en orden ASCendente por defecto. Pero, la mejor práctica es escribir ASC o DESC para que esta cláusula sea clara para todos los que lean tu consulta.

Cuándo usar LIMIT

La cláusula **LIMIT** es útil cuando solo se quiere trabajar con un número selecto de filas. Generalmente se usa en dos situaciones.

En la primera situación, digamos que quieres el número X más alto de casos. En el ejemplo de las entradas de cine, digamos que solo quieres las 3 compras más grandes. Podrías usar una cláusula LIMIT como la siguiente.

Consulta:

```
SELECT
      name,
      tickets
FROM
      purchases
ORDER BY
      tickets DESC
LIMIT 3 --top 3 results only
```

Resultado:

nombre	entradas
Sujin Lee	5
John Anderson	4
Rachel DeSantos	3

En la segunda situación, digamos que quieres trabajar con todo el conjunto de datos antes de escribir la consulta. En ese caso, usarías una cláusula LIMIT para no malgastar recursos extrayendo cada una de las filas.

Consulta:

```
SELECT
      name,
      tickets
FROM
      purchases
ORDER BY
      tickets DESC
LIMIT 20 --top 20 results only
```

Resultado:

nombre	entradas
Rachel DeSantos	3
Sujin Lee	5
Najil Okoro	2
John Anderson	4

Habrás notado que solo tienes cuatro filas de datos en nuestros resultados a pesar de haber establecido un límite de
20. Esto es así porque la tabla de compras solo tiene cuatro filas de datos. La cláusula LIMIT es el número máximo de filas a mostrar, y si la tabla de compras contuviera un millón de filas, solo se mostrarían 20 filas. Sin embargo, dado que la tabla de compras contiene menos de 20 filas, entonces, se muestran todos los datos.

Expresiones condicionales como CASE, IF y COALESCE()

Las **expresiones condicionales,** como las instrucciones CASE y la función COALESCE(), se usan cuando se desea cambiar la presentación de los resultados. Aunque estás estableciendo condiciones en expresiones condicionales, las expresiones condicionales difieren del tipo de condiciones que incluyes en una cláusula WHERE. Las condiciones puestas en las cláusulas WHERE se aplican a toda la consulta, mientras que las expresiones condicionales se aplican solo a

ese campo en particular. Además, puedes cambiar la forma en que se presentan los resultados con expresiones condicionales, algo que no se puede hacer con las instrucciones WHERE. Veamos las tres expresiones condicionales más comunes: CASE, IF y COALESCE().

Instrucciones CASE

Las instrucciones CASE son las más usadas como etiquetas dentro del conjunto de datos. Puedes usar las instrucciones CASE para etiquetar las filas que cumplen una determinada condición como X y las filas que cumplen otra condición como Y. Por eso, en general, encontrarás que esta instrucción se usa con funciones de agregación cuando quieres agrupar elementos por categorías. Este es un ejemplo en el que se utiliza una tabla de películas que se proyectan en el cine local:

Tabla PelículaCine:

genre	movie_title
terror	El silencio de los inocentes
comedia	Jumanji
familia	Frozen 2
documental	13º

Digamos que quieres agrupar estas películas en las que verás y las que no verás, y contar el número de películas que entran en cada categoría. Tu consulta sería:

```
SELECT
CASE
      WHEN genre = 'horror' THEN 'will not watch'
      ELSE 'will watch'
      END AS watch_category, --creating your own category
COUNT(movie_title) AS number_of_movies
FROM
      MovieTheater
GROUP BY
1 --when grouping by CASE, use position numbers or type
entire CASE statement here
```

Resultados:

watch_category	number_of_movies
No veré	1
Veré	3

Puedes ver que agregaste tus propias etiquetas al conjunto de datos, lo que puedes hacer con las instrucciones CASE. Pero se debe tener en cuenta que esta función no está en todos los dialectos de SQL, incluido BigQuery. Si te

gustaría aprender más, repasa la documentación de COUNT() o SUM() de tu dialecto SQL concreto y comprueba cómo se pueden usar las instrucciones CASE.

También hay otra forma de usar las instrucciones CASE en BigQuery (de nuevo, esto podría no aplicarse a todos los dialectos de SQL). Si sus condiciones coinciden, como en el ejemplo anterior, entonces podrías escribir tu instrucción CASE como (compara las líneas 2-3):

```
SELECT
      CASE genre
            WHEN 'horror' THEN 'will not watch'
            ELSE 'will watch'
            END AS watch_category
      COUNT(movie_title) AS number_of_movies
FROM
      MovieTheater
GROUP BY
      1
```

Esto producirá los mismos resultados, pero no se recomienda porque se limita a las condiciones que coinciden (por ejemplo, género = 'terror'). En comparación, la versión anterior con WHEN género = 'terror' es flexible y puede tomar otros tipos de condiciones, como mayor que (>), menor que (<), no igual a (<> o !=), etc.

Instrucciones IF

Lo que sigue son las instrucciones IF. Las **instrucciones IF** son similares a las instrucciones CASE, pero tienen una diferencia clave: Las instrucciones CASE pueden tomar múltiples condiciones, mientras que las instrucciones IF no. En el ejemplo anterior, solo tenías una condición (por ejemplo, WHEN género = 'terror'), por lo que también podrías haber utilizado una instrucción IF como:

```
SELECT
      IF(genre='horror', 'will not watch', 'will watch')
      AS watch_category,
      COUNT(movie_title) AS number_of_movies
FROM
      MovieTheater
GROUP BY
      1
```

Pero, si tienes múltiples condiciones, querrás usar una instrucción CASE, por ejemplo:

```
SELECT
      IF(genre='horror', 'will not watch', 'will watch')
      AS watch_category,
      COUNT(movie_title) AS number_of_movies
FROM
      MovieTheater
GROUP BY
      1
```

Resultados:

watch_category	number_of_movies
No veré	1
Veré solo	1
Veré con otros	2

Función COALESCE()

Por último, está la función **COALESCE()**. Esta función se usa para devolver la primera expresión no nula en el orden especificado en la función. Es útil cuando los datos están repartidos en varias columnas. Por ejemplo, digamos que tienes una tabla de películas como filas, columnas como meses y valores como 1 si la película se estrenó en ese mes o nulo si no lo hizo. Ver la tabla PelículaEstrenos a continuación:

movie_title	Jan_2030	Feb_2030	Mar_2030
Vengadores X	1	*nulo*	*nulo*
Frozen V	*nulo*	1	*nulo*
El Rey León IV	*nulo*	*nulo*	*nulo*

Si quieres saber si estas películas se estrenaron entre enero y marzo de 2030, puedes usar COALESCE. Por ejemplo:

```
SELECT
      movie_title,
      COALESCE(Jan_2030, Feb_2030, Mar_2030) AS
      launched_indicator
FROM
      MovieLaunches
```

Resultados:

movie_title	launched_indicator
Vengadores X	1
Frozen V	1
El Rey León IV	*nulo*

Puedes ver que dos de las tres películas tenían valores no nulos en los campos especificados (Ene_2030, Feb_2030, Mar_2030). Este ejemplo muestra cómo funciona la función COALESCE. Buscará en cada columna que especifique dentro de la función e intentará devolver un valor no nulo si lo encuentra.

En el área de trabajo, COALESCE se usa frecuentemente para asegurarse de que los campos no contengan valores nulos. De esta manera, una instrucción COALESCE podría ser: COALESCE(intenta_este_campo, luego_este_campo, 0) para indicar a SQL que compruebe los dos primeros campos en orden para un valor no nulo. Si no existe ninguno en esos campos, entonces asigna un cero en lugar de un valor nulo. En BigQuery, esto es lo mismo que usar la función IFNULL() (encontrarás más información sobre esto aquí). Es posible que otros dialectos de SQL no dispongan de la función IFNULL(), en cuyo caso se utilizará COALESCE() en su lugar.

Crear y eliminar tablas

Los datos en SQL se almacenan en tablas. En esta guía se ha hecho referencia a tablas pequeñas como las tablas de estrenos de películas y la de cines. Son tablas que ya existen porque no las has creado.

Crear tablas

La situación ideal para crear una tabla es que se cumplan las tres condiciones a continuación:

1. Consulta compleja con múltiples instrucciones JOIN
2. El resultado es una tabla
3. Tienes que ejecutar la consulta con frecuencia o de forma regular

Si se cumplen estas condiciones, es una gran idea crear una tabla de informes. Pero es una práctica recomendada consultar con tu superior o tus compañeros de equipo antes de hacerlo en caso de que necesites acceder a los permisos para crear una tabla de informes.

La sintaxis para crear tablas cambiará dependiendo del dialecto SQL y de la plataforma SQL que utilices. Aquí se repasa cómo crear tablas en BigQuery, pero si tu empresa usa un dialecto SQL diferente, es una buena idea buscar en Internet cómo crear tablas en ese dialecto SQL (por ejemplo, “Crear tablas en PostgreSQL”). O mejor aún, pide ayuda a tu superior o a un compañero de equipo.

En general, cuando se crean tablas, hay que asegurarse de que la misma tabla no exista con anterioridad. Esto se debe a que si se intenta crear una tabla que ya existe, la consulta devolverá un error.
La comprobación de las tablas existentes diferirá entre los dialectos de SQL, pero siempre es bueno comprobar para
evitar errores innecesarios en tu trabajo.

La forma de comprobar las tablas preexistentes en BigQuery es:

```
CREATE TABLE IF NOT EXISTS mydataset.FavoriteColorAndMovie
AS
SELECT
      friend,
      color,
      movie
FROM
      Favorite_Colors AS c
INNER JOIN
      Favorite_Movies AS m ON c.friend = m.friend
```

¡Creaste la tabla ColorFavoritoYPelícula! Puedes consultar esta tabla única para encontrar el color favorito y/o la película favorita de cada amigo sin tener que unir las dos tablas separadas, Colores_Favoritos y Películas_Favoritas, cada vez.

El método CREATE TABLE IF NOT EXISTS es mejor si las tablas de tu consulta (por ejemplo, Colores_Favoritos y Películas_Favoritas) no se actualizan continuamente. CREATE TABLE IF NOT EXISTS no hará nada si la tabla ya existe porque no se actualizará. Por lo tanto, si las tablas de origen se actualizan continuamente (por ejemplo, se añaden continuamente nuevos amigos con sus colores y películas favoritas) entonces, es mejor optar por un método diferente de creación de tablas.

Este otro método de creación de tablas es el método CREATE OR REPLACE TABLE. Así es como funciona:

```
CREATE OR REPLACE TABLE mydataset.FavoriteColorAndMovie
AS
SELECT
      friend,
      color,
      movie
FROM
      Favorite_Colors AS c
INNER JOIN
```

```
Favorite_Movies AS m ON c.friend = m.friend
```

Puedes ver que la única diferencia entre las dos formas de crear tablas es la primera línea de la consulta. Esto le dice a SQL qué hacer si la tabla ya existe. CREATE TABLE IF NOT EXISTS solo creará una tabla si todavía no existe. Si existe, se ejecutará la consulta pero no hará nada. Se trata de un mecanismo de seguridad para no sobrescribir por accidente una tabla potencialmente importante. Como alternativa, si necesitas sobrescribir una tabla, puedes usar CREATE OR REPLACE TABLE.

En resumen, se debe utilizar CREATE TABLE IF NOT EXISTS si estás creando una tabla estática que no necesita ser actualizada. Si tu tabla debe actualizarse continuamente, deberás usar CREATE OR REPLACE TABLE en su lugar.

Eliminar tablas

Ahora hablemos sobre la eliminación de tablas. Esto no pasa casi nunca, pero es importante saberlo porque, una vez que se eliminan las tablas, los datos contenidos en ellas pueden perderse. Y, dado que los datos son propiedad de la empresa para la que trabajas, eliminar una tabla podría significar deshacerse de algo que no es tuyo.

Piensa en cómo tratarías una cuenta de las redes sociales que no fuera tuya. Por ejemplo, si tu empresa te dio acceso a su cuenta. Puedes mirar las publicaciones y tal vez (con permiso) crear tu propia publicación para ellos, pero no eliminarías una publicación en las redes sociales porque es la cuenta del propietario, no la tuya.

Si alguna vez te encuentras contemplando la posibilidad de pulsar el botón de eliminación de una tabla que no creaste, asegúrate de verificar las razones por las que estás pulsando el botón de eliminación y vuelve a consultarlo con tu superior por las dudas.

Pero, si alguna vez necesitas eliminar una tabla, especialmente si es una que creaste y ya no necesitas, puedes eliminarla en BigQuery utilizando esta consulta:

```
DROP TABLE IF EXISTS mydataset.FavoriteColorAndMovie
```

DROP TABLE le dice a SQL que elimine la tabla, y la parte IF EXISTS se asegura de que no se produzca un error si la tabla no existe. Es un mecanismo de seguridad, de modo que si la tabla existe, será eliminada. Si la tabla no existe y ejecutas esta consulta, no pasará nada. Así que, de cualquier manera, la seguridad funciona a tu favor. La práctica recomendada es agregar IF EXISTS.

Tablas temporales

Hasta ahora, has aprendido cómo crear tablas y cuándo querrías crearlas. No dudes en repasar eso en la sección anterior si alguna vez necesitas un repaso. Las tablas que se crean con el método CREATE TABLE IF NOT EXISTS o CREATE OR REPLACE TABLE son tablas permanentes. Pueden ser compartidas y vistas por otras personas, y se puede acceder a ellas más tarde.

Sin embargo, puede haber situaciones en las que no sea necesario crear tablas permanentes. Recuerda: el almacenamiento de datos en SQL cuesta dinero y recursos a la empresa. Si no necesitas una tabla permanente, puedes crear tablas temporales en su lugar. Las tablas temporales solo existen dentro de tu sesión (o hasta 24 horas dependiendo de tu plataforma de SQL) y no se pueden compartir con terceros; tampoco son visibles para estos. Las tablas temporales solo existen para ti durante tu sesión. Piensa en las tablas temporales como en un bloc de notas donde puedes garabatear tus cálculos antes de escribir la respuesta final.

Comencemos por explicar cuándo querrás crear una tabla permanente en lugar de una tabla temporal.

Estas son las tres condiciones cuando quieras crear una tabla permanente. Se deben cumplir las tres condiciones.

1. Consulta compleja con múltiples instrucciones JOIN
2. El resultado es una tabla
3. Tienes que ejecutar la consulta con frecuencia o de forma regular

Por otra parte, las **tablas temporales** se usan para dividir las consultas complejas en incrementos más pequeños. Estas consultas complejas pueden contener múltiples instrucciones JOIN, pero no es necesario que así sea. Es posible que desees usar tablas temporales si se dan una o más de las siguientes condiciones:

- Consulta de ejecución lenta con múltiples instrucciones JOIN y WHERE
- Consulta de ejecución lenta que contiene GROUP BY y HAVING
- Consultas anidadas (es decir, una consulta dentro de otra)
- Si necesitas hacer un cálculo sobre otro cálculo (por ejemplo, sumar por día y, luego, promediar las sumas de todos los días)

Si se cumple alguna de las condiciones anteriores, el uso de una tabla temporal puede acelerar tu consulta, lo que te facilitará su escritura, así como la resolución de problemas si algo va mal.

A continuación se explica cómo crear una tabla temporal:

```
CREATE TEMP TABLE ExampleTable
AS
SELECT
      colors
FROM
      Favorite_Colors
```

Ahora, volvamos a la tabla permanente que creaste antes:

```
CREATE TABLE IF NOT EXISTS mydataset.FavoriteColorAndMovie
AS
SELECT
      friend,
      color,
      movie
FROM
      Favorite_Colors AS c
INNER JOIN
      Favorite_Movies AS m ON c.friend = m.friend
```

Esta consulta funciona para una tabla permanente porque se cumplen las tres condiciones que hemos mencionado anteriormente. Sin embargo, no es una buena idea usarla para tablas temporales. Estás trabajando con múltiples instrucciones JOIN, pero no hay instrucciones WHERE, y la consulta se ejecuta realmente rápido ya que las tablas Colores_Favoritos y Películas_Favoritas son relativamente pequeñas (filas de <100,000).

Consideremos un escenario diferente en el que deseas utilizar tablas temporales. Anteriormente aprendiste sobre GROUP BY y HAVING. Si tu consulta contiene ambas cláusulas, como la que se muestra a continuación, es posible que desees utilizar tablas temporales si tu consulta se ejecuta lentamente.

Consulta de la tabla temporal (si tu consulta se ejecuta lentamente):

```
SELECT
      occasion,
      SUM(tickets) AS total_tickets,
      COUNT(tickets) AS number_of_purchases
FROM
      purchases
GROUP BY
      occasion
HAVING
      SUM(tickets) > 5
```

En la consulta anterior, SQL tiene que realizar tres acciones. Primero, agrupará tu tabla por ocasión. Segundo, realizará las operaciones de SUM() y COUNT() de la columna de entradas. Tercero, solo mostrará las ocasiones con una SUM() de entradas superior a cinco. Si la tabla de compras fuera mucho más grande (más de un millón de filas) y también tuviera instrucciones JOIN en esta tabla, entonces, tu consulta probablemente se ejecutaría lentamente. Sin embargo, esto se puede evitar usando la cláusula HAVING y acelerando tu consulta al dividir la consulta en dos pasos utilizando tablas temporales.

Primero, puedes hacer las agregaciones con GROUP BY:

```
CREATE TEMP TABLE TicketsByOccasion
AS
SELECT
      occasion,
      SUM(tickets) AS total_tickets,
      COUNT(tickets) AS number_of_purchases
FROM
      purchases
GROUP BY
      occasion;
```

Luego, puedes ejecutar la limitación HAVING como una condición WHERE:

```
SELECT
      occasion,
      total_tickets,
      number_of_purchases
FROM
      TicketsByOccasion
WHERE
      total_tickets > 5
```

Hay tres puntos clave para tener en cuenta aquí:

1. Si estás ejecutando dos consultas al mismo tiempo, lo que tienes que hacer con tablas temporales (solo disponibles durante esa sesión), entonces necesitas un punto y coma para separar cada consulta.
2. En la primera consulta creas la tabla temporal. La segunda consulta hace referencia a esta tabla temporal y a sus campos. Es por eso por lo que puedes acceder a la suma de entradas como total_entradas en la segunda consulta.
3. Al crear los nombres de las tablas, no uses espacios o la consulta devolverá un error. La práctica recomendada es usar mayúsculas y minúsculas cuando se nombra la tabla que se está creando. El **estilo de escritura CamelCase** significa que se escribe en mayúscula el comienzo de cada palabra sin espacios intermedios, como la joroba de un camello. La tabla EntradasPorOcasión usa el estilo de escritura CamelCase.

En conclusión, no es necesario usar tablas temporales, pero pueden ser una herramienta realmente útil para ayudar a dividir las consultas complejas o complicadas en pasos más pequeños y manejables

Conclusión

Esta guía abarca muchos conceptos, pero puedes volver a ella una y otra vez mientras continúas escribiendo consultas SQL por tu cuenta. Tal como has aprendido a lo largo de este curso, la práctica es una parte importante del proceso de aprendizaje, y cuanto más practiques trabajar en SQL, más descubrimientos harás. Puedes guardar esta guía para poder repasar y consultar estas funciones y conceptos cuando lo necesites.