Guía de Subconsultas SQL con el esquema Sakila

Contents

Introducción

En SQL, una **subconsulta** es una consulta anidada dentro de otra consulta más grande. Las subconsultas permiten usar el resultado de una consulta interna para filtrar, comparar o derivar información en la consulta externa. En esta guía utilizaremos la base de datos de ejemplo **Sakila** (un videoclub con tablas como `film`, `actor`, `customer`, etc.) para ilustrar cómo funcionan las subconsultas.

Empezamos por subconsultas sencillas en la cláusula `WHERE`, pasando por subconsultas correlacionadas (que dependen de la fila externa) y subconsultas en la cláusula `FROM` (tablas derivadas) con agregación. Añadiremos explicaciones detalladas y comentarios en el código SQL para aclarar cada ejemplo.

Convención: Para mayor legibilidad, en los listados usaremos mayúsculas para palabras clave SQL (`SELECT`, `WHERE`, etc.), y comentarios precedidos de `-` para explicar partes clave de las consultas.

1 Subconsultas en la cláusula WHERE

Comenzamos con subconsultas *no correlacionadas* en la cláusula `WHERE`. Una subconsulta no correlacionada es aquella que la consulta interna puede ejecutar independientemente, porque no depende de cada fila externa. Estas subconsultas se resuelven primero y su resultado (ya sea un valor escalar o un conjunto de filas) se utiliza luego en la cláusula `WHERE` de la consulta principal. A las consultas que devuelven un único valor, se las denomina **escalares.**

Comparación con un valor escalar devuelto por subconsulta

Una situación común es querer filtrar filas comparando un campo con algún valor calculado mediante otra consulta. Por ejemplo, supongamos que queremos todas las películas cuyo idioma es **inglés**. En la base de datos `Sakila`, la tabla `film` no almacena directamente el nombre del idioma, sino el `language_id` como clave foránea a la tabla `language`. Podemos obtener el `language_id` que corresponde a "English" mediante una subconsulta, y usarlo en el filtro de la consulta principal:

```
SELECT title
FROM film
WHERE language_id = (
  SELECT language_id
  FROM language
  WHERE name = 'English'
);
```

En esta consulta, la subconsulta interna (entre paréntesis) devuelve el `language_id` del idioma cuyo `name` es 'English'. Esa subconsulta produce un único valor (en Sakila, el idioma English tiene `language_id = 1`). La consulta externa entonces selecciona todas las películas (`film.title`) cuyo `language_id` coincide con ese valor. En otras palabras, estamos traduciendo "English" a su `id` numérico con una subconsulta para luego obtener las películas en dicho idioma. Este tipo de subconsulta es escalar (devuelve un valor único) y podemos usar el operador `=` para compararlo.

Es importante asegurar que la subconsulta escalar realmente devuelva solo una fila. Si la subconsulta devolviera más de un valor, la consulta generaría un error (o necesitaríamos

usar operadores como `IN` en lugar de `=`, que veremos más adelante). En nuestro caso, `name` es único en `language`, por lo que la subconsulta de arriba devuelve exactamente un valor.

Otro ejemplo clásico Vamos a comparar contra un valor agregado. Imaginemos que deseamos listar las películas **más largas que la duración media de todas las películas**. Primero podemos calcular la duración promedio usando una subconsulta sobre la propia tabla `film`, y luego filtrar usando ese promedio:

```
SELECT title, length
FROM film
WHERE length > (
  SELECT AVG(length)
  FROM film
);
-- La subconsulta calcula la duración media de todas las películas.
-- La consulta externa filtra las películas cuya duración es mayor a dicha media.
```

Aquí la subconsulta `SELECT AVG(length) FROM film` devuelve un único número (el promedio de la columna `length` en `film`). Supongamos que el promedio es aproximadamente 115 minutos; entonces la consulta externa listará solo las películas con `length` mayor que 115. Esta construcción demuestra cómo integrar funciones de agregación en subconsultas para usarlas en condiciones de filtrado, algo que no podríamos hacer directamente en un `WHERE` sin subconsulta (ya que `WHERE` no puede usar agregados de la misma consulta directamente).

Uso de IN para subconsultas que devuelven conjuntos

Si la subconsulta puede devolver múltiples filas (por ejemplo, una lista de `id`s), debemos usar un operador de conjunto como `IN` (o su negación `NOT IN`) en lugar de un operador de comparación simple. Consideremos el siguiente problema: queremos obtener los nombres de todos los actores que actúan en la película *"ALONE TRIP"*. En Sakila, la tabla `film` nos permite obtener el `film_id` de "ALONE TRIP", y la tabla `film_actor` relaciona `film_id` con `actor_id`. Podemos resolverlo con subconsultas anidadas de esta manera:

```
SELECT first_name, last_name
FROM actor
WHERE actor_id IN (
  SELECT actor_id
  FROM film_actor
  WHERE film_id = (
    SELECT film_id
    FROM film
    WHERE title = 'ALONE TRIP'
)
);
```

Veamos este SQL paso a paso:

- La subconsulta interna `(SELECT film_id FROM film WHERE title = 'ALONE TRIP')` encuentra el `film_id` de la película "ALONE TRIP". Supongamos que devuelve el valor `17` (esto es solo un ejemplo; en Sakila el ID real de "ALONE TRIP" es 17).
- Usando ese valor, la subconsulta del medio `SELECT actor_id FROM film_actor WHERE film_id = 17` obtiene todos los `actor_id` de actores que participaron en la película

con `film_id = 17`. Esta subconsulta probablemente devolverá varios IDs (una lista de actores).

- Finalmente, la consulta externa selecciona los nombres (`first_name`, `last_name`) de la tabla `actor` para aquellos actores cuyo `actor_id` está en el conjunto devuelto por la subconsulta anterior.

Usamos `IN` porque la subconsulta de `film_actor` puede devolver múltiples filas. Si *Alone Trip* tiene, digamos, 5 actores, la condición `actor_id IN (...)` verificará la pertenencia de cada `actor_id` a ese conjunto de 5 valores. Como resultado, obtendremos todos los actores que aparecen en *"ALONE TRIP"*. Esta solución con subconsultas evita tener que escribir un `JOIN` múltiple.

Aunque podríamos haber resuelto la misma pregunta con `JOIN`s entre `film`, `film_actor` y `actor`, el objetivo aquí es practicar la lógica de subconsultas. De hecho, muchas consultas pueden expresarse de varias formas equivalentes; es útil comprender tanto las subconsultas como las combinaciones (`JOIN`) para elegir la más conveniente en cada caso.

Un detalle importante: si la subconsulta no encuentra ninguna fila, la condición `IN` simplemente no coincidirá con ningún valor (es decir, actuará como conjunción vacía). Por ejemplo, si buscamos `actor_id IN (...)` y la subconsulta interna devolviera cero filas, entonces ningún actor cumpliría la condición. En cambio, una subconsulta escalar usada con `=` que no devuelve filas produciría un valor `NULL` implícitamente, haciendo que la condición falle para todas las filas (a menos que usemos tratamiento especial de `NULL`).

Subconsultas con ANY, ALL y comparaciones avanzadas

SQL proporciona los predicados `ANY` (o su sinónimo `SOME`) y `ALL` para realizar comparaciones respecto a conjuntos de valores devueltos por una subconsulta. Estos predicados se usan junto con operadores de comparación (`=, >, <, >=, <=, <>`) cuando la subconsulta retorna varias filas. Por ejemplo, supongamos que queremos listar las películas cuya duración es **mayor que la de alguna película** de la categoría `Sports`. En otras palabras, buscamos películas que tengan más minutos que al menos una película de la categoría deportes (lo cual prácticamente incluiría a casi todas las películas excepto quizá las más cortas, ya que "> ANY" significa "> al mínimo", y es una condición relativamente fácil de cumplir). Por ilustración, escribiremos esta consulta así:

```
SELECT title, length
FROM film
WHERE length > ANY (
  SELECT f2.length
  FROM film AS f2
    JOIN film_category AS fc ON f2.film_id = fc.film_id
    JOIN category AS c ON fc.category_id = c.category_id
  WHERE c.name = 'Sports'
);
```

La subconsulta devuelve las longitudes de todas las películas en la categoría `Sports`. El predicado `> ANY` compara `film.length` (de la fila externa actual) con cada valor de ese conjunto y será verdadero si la longitud de la película externa es mayor que *al menos uno* de los valores devueltos. En términos matemáticos, "L > ANY(conjunto)" significa $L > \min(conjunto)$. Como comentamos, esta condición será cierta para prácticamente todas las películas que tengan una duración mayor que la película deportiva más corta. Más interesante es usar `ALL`. Por ejemplo, para encontrar las películas más largas que **todas** las

películas de `Sports` (es decir, más largas que la duración máxima de la categoría Sports), usaríamos `> ALL`:

```
SELECT title, length
FROM film
WHERE length > ALL (
  SELECT f2.length
  FROM film AS f2
    JOIN film_category AS fc ON f2.film_id = fc.film_id
    JOIN category AS c ON fc.category_id = c.category_id
  WHERE c.name = 'Sports'
);
```

Con `> ALL`, la condición solo será verdadera para aquellas películas cuya duración supere a cada una de las duraciones en la lista de películas Sports. Esto equivale a decir que su duración es mayor que la máxima duración de las películas Sports. En la práctica, obtendremos las películas más largas del inventario, siempre que las más largas no pertenezcan también a Sports (si la película más larga del catálogo fuera de Sports, entonces ninguna película podría ser mayor que todas las de Sports y el resultado sería vacío). En resumen, `ANY` y `ALL` nos permiten refinar comparaciones con subconsultas multivalor:

- **`condición > ANY(subconsulta)`**: Verdadero si la condición se cumple con al menos un valor del conjunto devuelto.
- **`condición > ALL(subconsulta)`**: Verdadero solo si se cumple con todos los valores del conjunto (es la condición más estricta).

Naturalmente, podemos usar estos predicados con cualquier operador de comparación (por ejemplo, `= ALL` significaría igual a todos los valores del conjunto, lo que generalmente solo ocurre si el conjunto tiene un único valor o varios repetidos iguales). Finalmente, mencionar que `NOT IN`, `NOT EXISTS` (que veremos enseguida) y construcciones como `<> ALL` se utilizan para formular condiciones de exclusión (por ejemplo, `<> ALL(...)` significa "distinto de todos los valores de..", es decir, "ninguno de esos valores coincide").

2 Subconsultas correlacionadas (EXISTS y NOT EXISTS)

Hasta ahora, las subconsultas que utilizamos no dependían de la fila actual de la consulta externa; podían ejecutarse por sí solas.

Subconsultas correlacionadas: Ahora veremos las **subconsultas correlacionadas**, que sí dependen de la fila externa. En una subconsulta correlacionada, la consulta interna hace referencia a columnas de la consulta externa, formando un vínculo entre ambas.

Esto significa que la subconsulta se reevaluará para cada fila candidata de la consulta externa. Un indicador típico de subconsulta correlacionada es el uso de **`EXISTS`** o **`NOT EXISTS`**, aunque también podemos tener correlación utilizando comparaciones normales. La estructura general es:

```
SELECT ...
%
FROM TablaExterna AS te
WHERE [NOT] EXISTS (
  SELECT 1
```

```
  FROM OtraTabla AS t
  WHERE t.algo = te.algo ...);
```

Aquí la subconsulta del `WHERE` puede acceder a `te.algo`, un valor de la fila actual de `TablaExterna`, y comprobar alguna condición relacionada en `OtraTabla`. `EXISTS` devuelve verdadero si la subconsulta interna retorna al menos una fila para esa combinación, y `NOT EXISTS` devuelve verdadero si la subconsulta interna no devuelve ninguna fila. Veamos ejemplos para clarificar.

Encontrar existencia de registros relacionados con EXISTS

Supongamos que queremos listar los clientes (`customer`) que han realizado al menos un alquiler. Podríamos resolver esto con un `JOIN` entre `customer` y `rental`, pero usando subconsulta correlacionada resulta muy instructivo. La idea es: para cada cliente de la tabla `customer`, comprobamos si **existe** al menos un registro en `rental` que corresponda a ese cliente. En SQL:

```
SELECT first_name, last_name
FROM customer AS c
WHERE EXISTS (
  SELECT 1
  FROM rental AS r
  WHERE r.customer_id = c.customer_id
);
```

Observemos la lógica:

- La consulta externa itera por cada cliente `c` de `customer`.
- La subconsulta interna busca `SELECT 1 FROM rental AS r WHERE r.customer_id = c.customer_id`. Aquí, `c.customer_id` se refiere al cliente actual de la consulta externa. La subconsulta selecciona simplemente un `1` (valor constante) porque solo nos importa la existencia, no necesitamos ningún dato en sí (tradicionalmente se pone `SELECT *` o `SELECT 1` ya que `EXISTS` solo comprueba si hay filas, ignorando los valores seleccionados).
- Si para un cliente dado existe al menos una fila de `rental` que coincide (es decir, ese cliente tiene algún alquiler), la subconsulta devuelve una o más filas, lo que hace que `EXISTS(...)` sea verdadero y por tanto ese cliente pase el filtro.
- Si un cliente no tiene ningún alquiler asociado, la subconsulta no encuentra filas y `EXISTS` resulta falso, filtrando fuera a ese cliente.

El resultado de esta consulta será la lista de todos los clientes que han realizado al menos un alquiler. En la base `Sakila`, probablemente esto incluya a casi todos los clientes, ya que es una base de ejemplo activa; pero si existieran clientes sin ninguna renta, éstos quedarían excluidos. (*Si quisiéramos, con `NOT EXISTS` podríamos obtener precisamente esos clientes sin alquileres, como veremos luego*).

NOTA: La condición `SELECT 1` se utiliza por costumbre, pero podríamos poner cualquier cosa (p.ej., `SELECT *`). El motor de SQL optimiza la subconsulta de `EXISTS` para que no importe qué columnas se seleccionan, solo importa si hay al menos una fila que cumpla el `WHERE` interno. En pseudocódigo, `EXISTS(subconsulta)` es como preguntar "¿existe al menos una fila en el resultado de subconsulta?".

Veamos ahora un ejemplo con `NOT EXISTS`. Supongamos que quisiéramos listar las películas que **nunca han sido alquiladas**. Para saber si una película ha sido alquilada, debemos ver si existe al menos un registro de `rental` asociado a alguna de sus copias (`inventory`). La tabla `inventory` nos dice qué copias físicas hay de cada película en cada tienda (`inventory.film_id` es la película de la copia). La tabla `rental` se vincula a `inventory` por `inventory_id`. Por lo tanto, una película no ha sido alquilada si *ninguna* de sus copias en inventario aparece en la tabla de rentals. Traduciendo esto a subconsulta correlacionada:

```
SELECT title
FROM film AS f
WHERE NOT EXISTS (
  SELECT 1
  FROM inventory AS i
    JOIN rental AS r ON r.inventory_id = i.inventory_id
  WHERE i.film_id = f.film_id
);
```

Aquí:

- La consulta externa recorre cada película `f` en la tabla `film`.
- La subconsulta correlacionada hace un `JOIN` entre `inventory` (`i`) y `rental` (`r`) para encontrar cualquier alquiler `r` que corresponda a una copia `i` de la película `f` actual (`i.film_id = f.film_id`). Si encuentra al menos uno, significaría que esa película sí fue alquilada.
- Usamos `NOT EXISTS` para filtrar solo las películas donde *no* exista tal registro combinado en `inventory-rental`. Es decir, la subconsulta interna no devolvió filas para ese `f.film_id` (ninguna copia de esa película tuvo alquileres).

El resultado será la lista de títulos de las películas nunca rentadas. En la práctica, es posible que esta lista esté vacía o muy reducida en la base Sakila si se asumió que todas las películas fueron alquiladas al menos una vez. Pero en problemas reales (o si Sakila estuviera actualizada con nuevos filmes sin alquilar) esta consulta identifica las películas "inmovibles". Aunque hubiéramos podido resolver esto con un `LEFT JOIN` y buscando `NULL` en la parte de `rental`, la solución con `NOT EXISTS` es declarativa y clara: "dame las películas para las que **no existe** ningún alquiler correspondiente".

Otro uso típico de subconsultas correlacionadas con `NOT EXISTS` es para preguntas del tipo "para cada X, que cumpla una condición con **todos** los Y relacionados". Por ejemplo, "actores que han participado en *todas* las categorías de películas" se resolvería comprobando que no existe ninguna categoría en la que el actor no tenga al menos una película. Esa consulta sería más compleja y no la desarrollaremos completa aquí por brevedad, pero conceptualmente:

```
SELECT actor_id, first_name, last_name
FROM actor AS a
WHERE NOT EXISTS (
  SELECT 1 FROM category AS c
  WHERE NOT EXISTS (
    SELECT 1
    FROM film_actor fa
      JOIN film_category fc ON fa.film_id = fc.film_id
    WHERE fa.actor_id = a.actor_id AND fc.category_id = c.category_id
    )
);
```

Lo importante a notar es la doble negación: "no existe una categoría tal que no exista una película de ese actor en dicha categoría", que equivale a "el actor tiene al menos una película en cada categoría". Este tipo de patrón doble `NOT EXISTS` permite expresar condiciones "para todos", aunque son consultas más avanzadas.

Mencionamos este ejemplo para dar una idea del poder de las subconsultas correlacionadas; si bien no es necesario que el estudiante lo entienda a la perfección de inmediato, muestra hasta dónde se puede llegar con estas técnicas.

2.1 Algunos ejemplos más

Ejemplo 1: Películas de la categoría "Children" (camino y pasos). Camino de tablas: `category` → `film_category` → `film`.

Paso 1: obtener `category_id` de "Children".

```
SELECT category_id
FROM category
WHERE name = 'Children';
```

Paso 2: con ese `category_id`, obtener `film_id` en `film_category`.

```
SELECT film_id
FROM film_category
WHERE category_id = 3; -- ejemplo: Children = 3
```

Paso 3: filtrar `film` con `IN` (lista de `film_id`).

```
SELECT title
FROM film
WHERE film_id IN (
    SELECT film_id
    FROM film_category
    WHERE category_id = (
        SELECT category_id
        FROM category
        WHERE name = 'Children'
    )
);
```

Resultado esperado: una lista de 60 títulos (no se reproducen aquí los 60 por brevedad).

Subconsultas correlacionadas con `EXISTS`

A veces, la condición de filtrado requiere verificar la existencia o ausencia de filas relacionadas para cada fila de la consulta externa. En tales casos, se usan subconsultas correlacionadas con `EXISTS` o `NOT EXISTS`.

Ejemplo 2: Películas nunca alquiladas (camino y pasos). Camino de tablas: `film` → (anti-join) `inventory` → `rental`.

Paso 1: consulta exterior.

```
SELECT f.film_id, f.title
FROM film AS f;
```

Paso 2: subconsulta sin correlación (estructura mínima).

```sql
SELECT 1
FROM inventory AS i
JOIN rental AS r ON i.inventory_id = r.inventory_id;
```

Paso 3: correlacionar con la fila exterior (`i.film_id = f.film_id`).

```sql
SELECT 1
FROM inventory AS i
JOIN rental AS r ON i.inventory_id = r.inventory_id
WHERE i.film_id = f.film_id;
```

Paso 4: usar `NOT EXISTS`.

```sql
SELECT f.film_id, f.title
FROM film AS f
WHERE NOT EXISTS (
    SELECT 1
    FROM inventory AS i
    JOIN rental AS r ON i.inventory_id = r.inventory_id
    WHERE i.film_id = f.film_id
)
ORDER BY f.title;
```

Resultado esperado: pocas o ninguna fila (en Sakila casi todas las películas han sido alquiladas).

Ejemplo 3: Actores con alguna película de `length > 120` (camino y pasos). Camino de tablas: `actor` → `film_actor` → `film`.

Paso 1: exterior.

```sql
SELECT a.actor_id, a.first_name, a.last_name
FROM actor AS a;
```

Paso 2: subconsulta sin correlación (estructura mínima).

```sql
SELECT 1
FROM film_actor AS fa
JOIN film AS f ON f.film_id = fa.film_id
WHERE f.length > 120;
```

Paso 3: correlacionar con la fila exterior.

```sql
SELECT 1
FROM film_actor AS fa
JOIN film AS f ON f.film_id = fa.film_id
WHERE fa.actor_id = a.actor_id
  AND f.length > 120;
```

Paso 4: usar `EXISTS`.

```sql
SELECT a.actor_id, a.first_name, a.last_name
FROM actor AS a
WHERE EXISTS (
  SELECT 1
  FROM film_actor AS fa
  JOIN film AS f ON f.film_id = fa.film_id
```

```
  WHERE fa.actor_id = a.actor_id
    AND f.length > 120
)
ORDER BY a.last_name, a.first_name;
```

Subconsultas en la lista SELECT (campos calculados por fila)

Otra forma de subconsulta correlacionada es colocarla en la lista `SELECT`, es decir, como parte de las columnas calculadas en la salida. Estas subconsultas deben ser de tipo escalar (devuelven un único valor por cada fila externa), y a menudo se usan para traer información agregada relacionada sin hacer un `JOIN GROUP BY` complicado. Por ejemplo, podríamos querer mostrar cada película junto con el número total de veces que ha sido alquilada. Podemos lograrlo con una subconsulta en la lista `SELECT` que cuente los alquileres (`rental`) de esa película:

```
SELECT
f.title,
(
  SELECT COUNT(*)
  FROM rental AS r
    JOIN inventory AS i ON r.inventory_id = i.inventory_id
  WHERE i.film_id = f.film_id
) AS total_rentals
FROM film AS f;
```

Aquí hemos puesto una subconsulta dentro del `SELECT`, etiquetándola como `total_rentals`. Esta subconsulta correlacionada se ejecuta para cada película `f`: realiza un conteo de cuántos registros de `rental` existen para la película actual (`f.film_id`) usando la relación `inventory` (porque los alquileres referencian copias, no directamente la película). El resultado será una lista de todas las películas con un número asociado de alquileres totales. Por ejemplo, puede mostrar:

TITLE	*total_rentals*
ACADEMY DINOSAUR	23
ACE GOLDFINGER	7
ADAPTATION HOLES	12
...	

Este enfoque de subconsulta escalar en el `SELECT` es equivalente a hacer un `JOIN` con `rental` e `inventory` seguido de un `GROUP BY film`, pero a veces la subconsulta resulta más sencilla de entender por filas: "para esta película, calcula este dato agregado". Hay que tener presente que este tipo de subconsulta puede ser menos eficiente que un `JOIN` agregando, especialmente si la tabla externa tiene muchas filas, porque la subconsulta se evalúa repetidamente.

En nuestro ejemplo, la base `film` tiene 1000 filas, así que la subconsulta contará alquileres 1000 veces (una por película). Sin embargo, para nuestros propósitos pedagógicos es más importante la claridad que la optimización. Otro ejemplo de subconsulta en el `SELECT`: podríamos listar cada cliente con el monto total que ha pagado en todos sus alquileres:

```
SELECT
  c.first_name,
  c.last_name,
  (
```

```
    SELECT SUM(p.amount)
    FROM payment AS p
    WHERE p.customer_id = c.customer_id
  ) AS total_spent
FROM customer AS c;
```

Esta consulta correlacionada suma los pagos (`payment.amount`) de cada cliente `c` y los muestra junto al nombre. Nuevamente, esto se podría hacer con un `JOIN` y `GROUP BY`, pero la subconsulta ofrece una alternativa útil cuando queremos calcular columnas agregadas particulares por fila.

En resumen, las subconsultas correlacionadas pueden emplearse en la cláusula `WHERE` (usando `EXISTS`, `NOT EXISTS`, `IN`, etc. con referencias a la fila externa) o en la lista `SELECT` como subconsultas escalares para crear columnas calculadas.

Son una herramienta potente para expresar condiciones complejas de forma declarativa.

3 Subconsultas en la cláusula FROM (tablas derivadas)

Las subconsultas también pueden usarse dentro de la cláusula `FROM`, lo que nos permite crear "tablas derivadas" o vistas temporales sobre las cuales realizar la consulta externa. Esto es útil cuando queremos estructurar la consulta en pasos: primero obtener un resultado intermedio con su propia `SELECT`, y luego consultarlo o combinarlo con otras tablas. Al poner una subconsulta en el `FROM`, **es obligatorio asignarle un alias**, igual que si fuera una tabla (porque la consulta externa debe referirse a ese resultado). Además, podemos renombrar las columnas de la subconsulta para usarlas cómodamente fuera.

Otro ejemplo: Veamos un caso práctico: supongamos que queremos encontrar las categorías de películas que tienen **más películas que el número promedio de películas por categoría**. Dicho de otro modo, queremos averiguar qué categorías están "por encima del promedio" en cuanto a cantidad de películas. Resolver esto directamente con agregaciones requiere dos niveles de agrupamiento: primero contar películas por categoría, luego comparar cada conteo con el promedio de todos los conteos. SQL no permite usar una agregación de agregación en un solo paso (no podemos calcular directamente en un `HAVING` el promedio de un `COUNT(*)` a menos que usemos una subconsulta). Así que las subconsultas en `FROM` vienen perfectas para este tipo de problema. Dividiremos la solución en pasos lógicos:

1. Obtener el número de películas por categoría.
2. Calcular el promedio de esos números.
3. Seleccionar las categorías cuyo número está por encima de ese promedio.

Primero, construyamos una subconsulta que calcule el total de películas en cada categoría. Las tablas relevantes son `film_category` (que relaciona películas con categorías) y `category` (para obtener el nombre de la categoría si queremos mostrarlo):

```
SELECT category_id, COUNT(*) AS film_count
FROM film_category
GROUP BY category_id
```

Esta subconsulta produce una tabla con dos columnas: el `category_id` y la cantidad de películas (`film_count`) en esa categoría. Ahora la usaremos dentro de la cláusula `FROM` de la

consulta principal, aliándola, y también incorporaremos el cálculo del promedio. Podemos calcular el promedio de `film_count` con otra subconsulta separada, o incluso usando la misma subconsulta de conteo dentro de la `WHERE`. Para mantenerlo claro, haremos una subconsulta derivada principal y luego otra anidada para el promedio:

```
SELECT
  c.name AS category_name,
  category_counts.film_count
FROM (
  SELECT category_id, COUNT() AS film_count
  FROM film_category
  GROUP BY category_id
  ) AS category_counts
    JOIN
  category AS c ON c.category_id = category_counts.category_id
WHERE category_counts.film_count > (
  SELECT AVG(film_count)
  FROM (
    SELECT category_id, COUNT() AS film_count
    FROM film_category
    GROUP BY category_id
    ) AS category_counts2
);
```

Analicemos esta consulta compleja:

- La parte dentro de `FROM (...) AS category_counts` es la subconsulta derivada que saca el total de películas por categoría, tal como definimos antes. Ahora ese resultado se comporta como si fuera una tabla llamada `category_counts`, con columnas `category_id` y `film_count`.
- Hacemos un `JOIN` de `category_counts` con la tabla `category` real para obtener el `name` (nombre de la categoría) correspondiente a cada `category_id`. Así, podemos mostrar el nombre en vez del número de ID.
- En la cláusula `WHERE`, comparamos `category_counts.film_count` con el promedio de `film_count` de todas las categorías. Ese promedio se obtiene mediante otra subconsulta:
  - `SELECT AVG(film_count) FROM (...), AS, category_counts2`. Aquí hemos reutilizado la misma subconsulta de conteo dentro para calcular su promedio global. El alias `category_counts2` es necesario pero no lo usamos fuera; solo sirve para que la sintaxis sea correcta.
  - En esencia, `AVG(film_count)` tomará todos los conteos por categoría y hallará el promedio.
- Solo se seleccionarán aquellas filas donde `film_count` sea mayor que ese promedio. Es decir, categorías cuyo número de películas excede la media general.

La consulta final lista el nombre de cada categoría y su conteo de películas, para las categorías sobre la media. Si ejecutamos esto en Sakila, obtendríamos un resultado parecido a:

category_name	*film_count*
Sports	74
Foreign	73
Games	64

...

(asumiendo que la media esté alrededor de 62.5, estas categorías podrían salir por encima; el resultado exacto puede variar ligeramente según la distribución).

Nota: Esta solución es perfectamente válida, pero nos ha obligado a repetir la subconsulta de conteo dos veces (una para usar los conteos en sí, otra para calcular la media).

En SQL a veces existen formas alternativas. Por ejemplo, podríamos haber calculado el total de películas (N) y el número de categorías (K) para computar la media $\frac{N}{K}$ en una sola subconsulta. O incluso usar una `WITH` (Common Table Expression) para definir la subconsulta de conteo y referenciarla dos veces.

Otra ilustración más sencilla de subconsulta en `FROM` es usarla para lograr un "paso intermedio". Por ejemplo, podríamos reescribir la consulta de "películas con duración superior a la media" (que resolvimos con subconsulta escalar en `WHERE`) de la siguiente manera con una tabla derivada:

```
SELECT f.title, f.length
FROM film AS f
JOIN (
  SELECT AVG(length) AS avg_length FROM film) AS stats
WHERE f.length > stats.avg_length;
```

Aquí la subconsulta `(SELECT AVG(length) FROM film)` produce una relación de una sola fila con una columna `avg_length`. Al hacer un `JOIN` sin especificar condición (`JOIN` implícitamente es un `CROSS JOIN`[1] cuando no hay `ON`), esa única fila se "adhiere" a cada película `f`, permitiéndonos comparar `f.length` con `stats.avg_length` fácilmente.

Esta forma puede parecer redundante para este caso trivial, pero demuestra que las subconsultas en `FROM` pueden ser útiles para almacenar valores globales (como una media) o resultados agregados que luego usamos en comparaciones con cada fila. En general, las subconsultas en `FROM` (derivadas) son muy útiles para:

- Descomponer consultas complejas en subpasos comprensibles.
- Evitar limitaciones del SQL básico, como la imposibilidad de usar una agregación de una consulta externa directamente en otra agregación.
- Reutilizar resultados intermedios (aunque ojo, en nuestra solución tuvimos que repetir la subconsulta; con otras técnicas podríamos evitar repeticiones).

Vale la pena mencionar que cualquier subconsulta en `FROM` podría equivalerse a definir una **vista** o usar una expresión de tabla común (`WITH`), pero esas herramientas avanzadas escapan al alcance actual de nuestros supuestos (no estamos creando vistas permanentes, solo consultas en el momento).

Conclusiones

En esta guía hemos explorado las subconsultas SQL aplicadas a la base de datos Sakila, y las hemos ordenado desde los usos más básicos hasta construcciones más complejas:

[1] Un `CROSS JOIN` es como poner `FROM tabla1,tabla2`

- Empezamos con subconsultas simples en la cláusula `WHERE`, útiles para comparar contra un valor calculado o filtrar por pertenencia a un conjunto (`IN`/`NOT IN`).
- Introdujimos los predicados `ANY` y `ALL` para comparaciones con conjuntos de valores devueltos por subconsulta.
- Vimos subconsultas correlacionadas, que dependen de la fila externa, usando `EXISTS`/`NOT EXISTS` para verificar existencia o ausencia de registros relacionados. Estas nos permitieron plantear condiciones como "existe al menos uno" o "no existe ninguno", muy útiles para expresiones complejas (incluso anidando `NOT EXISTS` para condiciones de "para todos").
- También utilizamos subconsultas correlacionadas como subconsultas escalares en la lista `SELECT` para obtener columnas derivadas por fila (por ejemplo, contar elementos relacionados sin `GROUP BY` externo).
- Finalmente, exploramos las subconsultas en la cláusula `FROM`, creando tablas derivadas. Vimos cómo esta técnica permite estructurar consultas en pasos y comparar agregados de manera flexible, a costa de introducir alias adicionales y, en algunos casos, subconsultas anidadas adicionales.

A lo largo de los ejemplos, añadimos comentarios y explicaciones para aclarar cómo trabaja cada subconsulta. Es importante practicar estos conceptos con variaciones:

- Convertir `JOIN`s en subconsultas y viceversa, para entender la equivalencia lógica.
- Usar `NOT IN` o `NOT EXISTS` para formular consultas "¿quiénes no...?".
- Intentar escribir consultas con dos o más niveles de anidamiento, como las de "para todos" que requieren pensar cuidadosamente la lógica de negación.

Con esta base, el estudiante debería estar más cómodo abordando problemas SQL que involucren múltiples pasos. Las subconsultas son una herramienta muy poderosa en SQL; dominarlas amplía significativamente el rango de consultas que podemos resolver directamente en la base de datos, incluso sin recurrir a programación procedimental o temporales.

Por último, hay que señalar que la legibilidad de una consulta SQL es tan importante como su corrección. A veces una subconsulta puede hacer la solución más intuitiva, y otras veces un `JOIN` con `GROUP BY` es más natural. La experiencia y la práctica ayudarán a decidir cuál emplear en cada caso.

4 Ejercicios de práctica (con resultados esperados)

Ejercicio 1: Películas por encima de la duración promedio

Camino: `film`.
Enunciado: listar **título** y **duración** (`length`) de las películas cuya duración es superior a la media global.

Resultado esperado (extracto):

Título	length
AFRICAN EGG	130
AGENT TRUMAN	169
AIRPLANE SIERRA	194
AMERICAN CIRCUS	181
ANTITRUST TOMATOES	179
...	...

Consulta:

```
SELECT title, length
FROM film
WHERE length > (SELECT AVG(length) FROM film)
ORDER BY length DESC, title;
```

Ejercicio 2: Actores en "Action" y en "Comedy"

Camino: `actor` → `film_actor` → `film_category` → `category`.
Enunciado: actores con al menos una película en `Action` y al menos una en `Comedy`.

Resultado esperado (extracto):

first_name	last_name
GINA	DEGENERES
KENNETH	TORN
WALTER	TORN
...	...

Posible solución (EXISTS doble):

```
SELECT a.first_name, a.last_name
FROM actor AS a
WHERE EXISTS (
  SELECT 1
  FROM film_actor AS fa
  JOIN film_category AS fc ON fc.film_id = fa.film_id
  JOIN category AS c ON c.category_id = fc.category_id
  WHERE fa.actor_id = a.actor_id AND c.name = 'Action'
)
AND EXISTS (
  SELECT 1
  FROM film_actor AS fa
  JOIN film_category AS fc ON fc.film_id = fa.film_id
  JOIN category AS c ON c.category_id = fc.category_id
  WHERE fa.actor_id = a.actor_id AND c.name = 'Comedy'
)
ORDER BY a.last_name, a.first_name;
```

Ejercicio 3: Cliente con más alquileres

Camino: `rental` → (derivada por `customer_id`) → `customer`.
Enunciado: `customer_id`, **nombre** y **apellido** del cliente con mayor número de alquileres.

Resultado esperado:

customer_id	first_name	last_name
526	ELEANOR	HUNT

Consulta (una opción):

```
SELECT c.customer_id, c.first_name, c.last_name
FROM customer AS c
JOIN (
  SELECT r.customer_id, COUNT(*) AS rentals
  FROM rental AS r
  GROUP BY r.customer_id
) AS t ON t.customer_id = c.customer_id
WHERE t.rentals = (
  SELECT MAX(x.rentals)
  FROM (
    SELECT r2.customer_id, COUNT(*) AS rentals
    FROM rental AS r2
    GROUP BY r2.customer_id
  ) AS x
);
```

Nota: puede ser que haya formas más concisas de operar esta consulta, pero la resolvemos así para mostrar cómo pueden funcionar las subconsultas.

Ejercicio 4: Películas de "Sports" no alquiladas en 2005

Camino: `film` → `film_category` → `category` y anti-join `inventory` → `rental`.
Enunciado: títulos de películas de `Sports` sin alquileres durante 2005.

Resultado esperado:

Título (*Sports* sin alquileres en 2005)
JAWBREAKER BROOKLYN
ROOF CHAMPION

Consulta (NOT EXISTS):

```
SELECT f.title
FROM film AS f
JOIN film_category AS fc ON fc.film_id = f.film_id
JOIN category AS c ON c.category_id = fc.category_id
WHERE c.name = 'Sports'
  AND NOT EXISTS (
    SELECT 1
    FROM inventory AS i
    JOIN rental AS r ON r.inventory_id = i.inventory_id
    WHERE i.film_id = f.film_id
      AND YEAR(r.rental_date) = 2005
  )
ORDER BY f.title;
```

Como cierre, recuerda:

- **Derivadas** en `FROM`: siempre con alias y útiles para encadenar etapas (`HAVING` no sustituye casos multi-etapa).

- **Escalares** en `SELECT`: un valor por fila (asegura que no devuelvan múltiples filas).
- **Correlacionadas** con `EXISTS`/`NOT EXISTS`: expresan existencia/ausencia sin duplicar filas.
- Practica ejecutando primero la subconsulta *sola*, luego intégrala paso a paso.

5 Ejercicios resueltos

5.1 Consultas derivadas, correlacionadas y escalares

En este apartado se muestran diferentes tipos de subconsultas en SQL, todas ellas basadas en la base de datos `sakila`. El objetivo es comprender cuándo conviene usar una *subconsulta derivada*, una *subconsulta correlacionada* o una *subconsulta escalar*, así como interpretar su resultado.

Derivada — Películas por idioma

Enunciado: Queremos conocer cuántas películas hay registradas en la base de datos por cada idioma. Para ello agrupamos las películas según su `language_id`, contamos cuántas hay en cada grupo, y posteriormente relacionamos ese resultado con la tabla `language` para mostrar el nombre del idioma.

Tipo de subconsulta: derivada (tabla intermedia agregada).

Camino: `film → (derivada por language_id) → language`

Salida: `language_id, language_name, films_in_language (desc)`

Resultado esperado:

language_id	language_name	films_in_language
1	English	1000

```
SELECT l.language_id, l.name AS language_name, t.films_in_language
FROM (
  SELECT f.language_id, COUNT(*) AS films_in_language
  FROM film AS f
  GROUP BY f.language_id
) AS t
JOIN language AS l ON l.language_id = t.language_id
ORDER BY t.films_in_language DESC, l.name;
```

Interpretación: La subconsulta crea una tabla temporal (`t`) que contiene la cantidad de películas por idioma. Posteriormente se une con `language` para mostrar el nombre de cada idioma. El resultado demuestra que todas las películas están en inglés.

Derivada — Idiomas con longitud media superior a 110 minutos

Enunciado: Deseamos saber qué idiomas tienen películas con una duración media superior a 110 minutos. Se calcula la media de la longitud (`length`) de las películas agrupadas por idioma y se filtran los casos que superan ese umbral.

Tipo de subconsulta: derivada con función de agregación y filtro exterior.

Camino: `film → (derivada por language_id) → language`
Salida: `language_id, language_name, avg_length`

Resultado esperado:

language_id	language_name	avg_length
1	English	115.2720

```
SELECT l.language_id, l.name AS language_name, s.avg_length
FROM (
  SELECT f.language_id, AVG(f.length) AS avg_length
  FROM film AS f
  GROUP BY f.language_id
) AS s
JOIN language AS l ON l.language_id = s.language_id
WHERE s.avg_length > 110
ORDER BY s.avg_length DESC, l.name;
```

Interpretación: La consulta interna agrupa las películas por idioma y calcula su duración media. El filtro en la consulta principal permite mostrar solo los idiomas cuya media supera los 110 minutos. En la base de datos `sakila`, solo el idioma inglés cumple la condición.

Derivada — Máximo y mínimo `replacement_cost` por idioma

Enunciado: Queremos conocer el rango de precios de reemplazo de las películas por cada idioma, es decir, el coste mínimo y máximo. La subconsulta calcula ambos valores por idioma y la consulta exterior añade el nombre del idioma.

Camino: `film → (derivada por language_id) → language`
Salida: `language_id, language_name, max_replacement_cost, min_replacement_cost`

Resultado esperado:

language_id	language_name	max_replacement_cost	min_replacement_cost
1	English	29.99	9.99

```
SELECT l.language_id, l.name AS language_name, m.max_replacement_cost, m.
    min_replacement_cost
FROM (
  SELECT f.language_id,
         MAX(f.replacement_cost) AS max_replacement_cost,
         MIN(f.replacement_cost) AS min_replacement_cost
  FROM film AS f
  GROUP BY f.language_id
) AS m
JOIN language AS l ON l.language_id = m.language_id
ORDER BY l.language_id;
```

Interpretación: Esta subconsulta derivada sirve para calcular valores agregados múltiples (máximo y mínimo) por cada grupo. Permite obtener un resumen de los extremos de precios sin necesidad de usar funciones escalares múltiples.

Correlacionada (EXISTS) — Idiomas con al menos una película `rating='R'`

Enunciado: Queremos saber qué idiomas tienen al menos una película clasificada con la calificación `'R'`. Utilizamos una subconsulta correlacionada, ya que cada fila del exterior (`language`) evalúa la condición en la tabla `film`.

Camino: `language → film(rating='R')`
Salida: `language_id, language_name`

Resultado esperado:

language_id	language_name
1	English

```
SELECT l.language_id, l.name AS language_name
FROM language AS l
WHERE EXISTS (
  SELECT 1
  FROM film AS f
  WHERE f.language_id = l.language_id
    AND f.rating = 'R'
)
ORDER BY l.language_id;
```

Interpretación: `EXISTS` evalúa la existencia de al menos una fila que cumpla la condición interna. Es una consulta correlacionada porque depende del valor actual de `l.language_id`. Solo el idioma inglés contiene películas clasificadas como `'R'`.

Escalar — Número total de idiomas distintos presentes en `film`

Enunciado: Queremos obtener un único valor: cuántos idiomas diferentes existen en la tabla `film`. Al tratarse de un solo resultado, se utiliza una subconsulta escalar dentro del `SELECT`.

Camino: `film`
Salida: `film_languages`

Resultado esperado:

film_languages
1

```
SELECT (
  SELECT COUNT(DISTINCT f.language_id)
  FROM film AS f
) AS film_languages;
```

Interpretación: El resultado es una única columna con un único valor escalar: el número de idiomas diferentes en la tabla `film`. Este tipo de subconsulta se usa cuando queremos integrar un valor agregado en una consulta más grande.

Escalar — Número de películas con calificación 'R'

Enunciado: Queremos saber cuántas películas tienen calificación 'R'. La subconsulta se ejecuta de forma independiente y devuelve un valor único, que se muestra en una columna.

Camino: film
Salida: films_rating_r

Resultado esperado:

films_rating_r
195

```
SELECT (
  SELECT COUNT(*)
  FROM film AS f
  WHERE f.rating = 'R'
) AS films_rating_r;
```

Interpretación: Una subconsulta escalar devuelve un solo valor que puede usarse en una expresión o mostrar directamente como resultado final. En este caso, el número total de películas con calificación 'R' es 195.

CTE (WITH) — Actores con al menos 30 películas

Enunciado: Queremos listar los actores que han participado en 30 o más películas. Usamos una *CTE* (*Common Table Expression*) que genera una tabla temporal con el número de películas por actor, y luego la consultamos como si fuera una tabla normal.

Camino: film_actor → (CTE por actor) → actor
Salida: actor_id, first_name, last_name, total_films

```
WITH film_count AS (
  SELECT fa.actor_id, COUNT(*) AS total_films
  FROM film_actor AS fa
  GROUP BY fa.actor_id
)
SELECT a.actor_id, a.first_name, a.last_name, fc.total_films
FROM film_count AS fc
JOIN actor AS a ON a.actor_id = fc.actor_id
WHERE fc.total_films >= 30
ORDER BY fc.total_films DESC, a.last_name, a.first_name;
```

Interpretación: Las CTEs son una alternativa moderna y más legible a las subconsultas derivadas. Permiten reutilizar resultados intermedios con nombres temporales y facilitan la lectura de consultas complejas.

6 Para practicar

Formato de cada ejercicio: Cada consulta incluye: **tipo**, **camino de tablas**, **resultado esperado** (tabla verificable) y **SQL**.

6.1 Derivada — Películas por idioma

Camino: `film` → (derivada por `language_id`) → `language`

Table 1: Resultado esperado

language_id	language_name	films_in_language
1	English	1000

```
SELECT l.language_id, l.name AS language_name, t.films_in_language
FROM (
  SELECT f.language_id, COUNT(*) AS films_in_language
  FROM film AS f
  GROUP BY f.language_id
) AS t
JOIN language AS l ON l.language_id = t.language_id
ORDER BY t.films_in_language DESC, l.name;
```

6.2 Derivada — Idiomas con AVG(length) > 110

Camino: `film` → (derivada por `language_id`) → `language`

Table 2: Resultado esperado

language_id	language_name	avg_length
1	English	115.2720

```
SELECT l.language_id, l.name AS language_name, s.avg_length
FROM (
  SELECT f.language_id, AVG(f.length) AS avg_length
  FROM film AS f
  GROUP BY f.language_id
) AS s
JOIN language AS l ON l.language_id = s.language_id
WHERE s.avg_length > 110
ORDER BY s.avg_length DESC, l.name;
```

6.3 Derivada — Máx/Mín replacement_cost por idioma

Camino: `film` → (derivada por `language_id`) → `language`

Table 3: Resultado esperado

language_id	language_name	max_replacement_cost	min_replacement_cost
1	English	29.99	9.99

```sql
SELECT l.language_id, l.name AS language_name, m.max_replacement_cost, m.
    min_replacement_cost
FROM (
  SELECT f.language_id,
         MAX(f.replacement_cost) AS max_replacement_cost,
         MIN(f.replacement_cost) AS min_replacement_cost
  FROM film AS f
  GROUP BY f.language_id
) AS m
JOIN language AS l ON l.language_id = m.language_id;
```

6.4 Correlacionada (EXISTS) — Idiomas con películas rating='R'

Camino: `language` → `film(rating='R')`

Table 4: Resultado esperado

language_id	language_name
1	English

```sql
SELECT l.language_id, l.name AS language_name
FROM language AS l
WHERE EXISTS (
  SELECT 1
  FROM film AS f
  WHERE f.language_id = l.language_id
    AND f.rating = 'R'
);
```

6.5 Escalar — Nº de idiomas distintos en film

Camino: `film`

Table 5: Resultado esperado

film_languages
1

```sql
SELECT (SELECT COUNT(DISTINCT f.language_id) FROM film AS f) AS film_languages;
```

6.6 Escalar — Nº de películas rating='R'

Camino: `film`

```sql
SELECT (SELECT COUNT(*) FROM film AS f WHERE f.rating = 'R') AS films_rating_r;
```

Table 6: Resultado esperado

films_rating_r
195

Table 7: Resultado esperado

language_name	films_in_language
English	1000

6.7 Derivada — Idioma con más películas (TOP-1)

Camino: `film` → (derivada por `language_id`) → `language`

```
SELECT l.name AS language_name, t.films_in_language
FROM (
  SELECT f.language_id, COUNT(*) AS films_in_language
  FROM film AS f
  GROUP BY f.language_id
) AS t
JOIN language AS l ON l.language_id = t.language_id
ORDER BY t.films_in_language DESC, l.name
LIMIT 1;
```

6.8 Derivada — Idioma con mayor AVG(length) (TOP-1)

Camino: `film` → (derivada por `language_id`) → `language`

Table 8: Resultado esperado

language_name	avg_length
English	115.2720

```
SELECT l.name AS language_name, s.avg_length
FROM (
  SELECT f.language_id, AVG(f.length) AS avg_length
  FROM film AS f
  GROUP BY f.language_id
) AS s
JOIN language AS l ON l.language_id = s.language_id
ORDER BY s.avg_length DESC, l.name
LIMIT 1;
```

6.9 Escalar — Media global de length

Camino: `film`

```
SELECT (SELECT AVG(f.length) FROM film AS f) AS avg_length_global;
```

Table 9: Resultado esperado

avg_length_global
115.2720

6.10 Correlacionada (NOT EXISTS) — Idiomas de `language` sin películas

Camino: `language` → (anti) `film`

Table 10: Resultado esperado

language_id	language_name
2	Italian
3	Japanese
4	Mandarin
5	French
6	German

```
SELECT l.language_id, l.name AS language_name
FROM language AS l
WHERE NOT EXISTS (
  SELECT 1
  FROM film AS f
  WHERE f.language_id = l.language_id
)
ORDER BY l.language_id;
```

6.11 Escalar — Películas no R (cálculo a partir de totales)

Camino: `film`

Table 11: Resultado esperado

films_not_r
805

```
SELECT
  (SELECT COUNT(*) FROM film) -
  (SELECT COUNT(*) FROM film WHERE rating='R')
  AS films_not_r;
```

6.12 Escalar — Proporción de R sobre el total (4 decimales)

Camino: `film`

Table 12: Resultado esperado

ratio_r
0.1950

```
SELECT ROUND(
  (SELECT COUNT(*) FROM film WHERE rating='R') /
  (SELECT COUNT(*) FROM film) , 4
) AS ratio_r;
```

6.13 Correlacionada (EXISTS) — ¿Algún idioma distinto de English con películas?

Camino: `language`(≠English) → (*¿existe?*) `film`

Table 13: Resultado esperado: tabla vacía

language_id	name
(sin filas)	

```
SELECT l.language_id, l.name
FROM language AS l
WHERE l.name <> 'English'
  AND EXISTS (SELECT 1 FROM film AS f WHERE f.language_id = l.language_id);
```

6.14 Escalar — ¿Cuántos idiomas hay en `language`?

Camino: `language`

Table 14: Resultado esperado

langs_in_language
6

```
SELECT (SELECT COUNT(*) FROM language) AS langs_in_language;
```

6.15 Correlacionada (EXISTS) — Idiomas con al menos una película

Camino: `language` → `film`

```
SELECT l.name AS language_name
FROM language AS l
WHERE EXISTS (SELECT 1 FROM film AS f WHERE f.language_id = l.language_id);
```

Table 15: Resultado esperado

language_name
English

Table 16: Resultado esperado

language_name
English

6.16 Derivada — Idiomas con MAX(replacement_cost) = 29.99

Camino: `film` → (derivada MAX por `language_id`) → `language`

```
SELECT l.name AS language_name
FROM (
  SELECT f.language_id, MAX(f.replacement_cost) AS mx
  FROM film AS f
  GROUP BY f.language_id
) AS t
JOIN language AS l ON l.language_id = t.language_id
WHERE t.mx = 29.99;
```

6.17 Derivada — Idiomas con MIN(replacement_cost) = 9.99

Camino: `film` → (derivada MIN por `language_id`) → `language`

Table 17: Resultado esperado

language_name
English

```
SELECT l.name AS language_name
FROM (
  SELECT f.language_id, MIN(f.replacement_cost) AS mn
  FROM film AS f
  GROUP BY f.language_id
) AS t
JOIN language AS l ON l.language_id = t.language_id
WHERE t.mn = 9.99;
```

6.18 Correlacionada (EXISTS) — Idiomas con películas no 'R'

Camino: `language` → `film`($\neq$'R')

```
SELECT l.name AS language_name
FROM language AS l
WHERE EXISTS (
  SELECT 1 FROM film AS f
```

Table 18: Resultado esperado

language_name
English

```
  WHERE f.language_id = l.language_id
    AND f.rating <> 'R'
);
```

6.19 Derivada — ¿Cuántas películas totales hay? (derivada trivial)

Camino: `film` → (derivada de conteo)

Table 19: Resultado esperado

total_films
1000

```
SELECT t.total_films
FROM (SELECT COUNT(*) AS total_films FROM film) AS t;
```

6.20 Escalar — Media global de length (redondeada a 4 decimales)

Camino: `film`

Table 20: Resultado esperado

avg_len_4d
115.2720

```
SELECT ROUND((SELECT AVG(f.length) FROM film AS f), 4) AS avg_len_4d;
```