Ayudantía 3 · Consultas avanzadas: anidación, agregación y nulos

Clases 04, 05 y 06 · IIC2413

Benjamin Saavedra

Nuestra base de datos: tienda de videojuegos (recordatorio)

```
desarrolladores(id_desarrollador, nombre, pais)                                    ar = 3
juegos(id_juego, titulo, id_desarrollador, genero, plataforma, precio, anio_lanzamiento)    ar = 7
compras(id_compra, id_juego, nombre_comprador, fecha_compra)                       ar = 4
```

Mismo esquema de la ayudantía pasada: `juegos.id_desarrollador` referencia a `desarrolladores`, `compras.id_juego` referencia a `juegos`.

Hoy la vamos a **ampliar un poco**, justo para poder hablar de nulos y de outer-joins.

La ampliamos para hoy

```
desarrolladores(id_desarrollador, nombre, pais)                                    ar = 3
juegos(id_juego, titulo, id_desarrollador, genero, plataforma, precio, anio_lanzamiento, rating)    ar = 8
compras(id_compra, id_juego, nombre_comprador, fecha_compra)                       ar = 4
```

Tres cambios:

1. `juegos` tiene una columna nueva: **`rating`** (puntaje 0-100). Algunos juegos recién agregados a la tienda **todavía no tienen rating** → `rating = NULL`.
2. Agregamos un desarrollador nuevo, **`Bethesda Game Studios`**, que aún **no tiene ningún juego** cargado en la tienda.
3. Agregamos un juego nuevo, **`Undertale`**, al que **todavía no le asignamos desarrollador** → `id_desarrollador = NULL`.

Con esto podemos hacer preguntas del tipo "¿qué desarrolladores no tienen juegos?" o "¿qué juegos no tienen rating?".

Repaso

Agregación (Clase 05)

Agregación en SQL

```
SELECT AGG(<atributo>)
FROM <lista-relaciones>
WHERE <condicion>
```

Una consulta con `AGG` no devuelve filas individuales: devuelve **un resumen** de un grupo de filas (por defecto, un solo grupo con toda la tabla filtrada).

Funciones de agregación vistas en clase

Función	Qué hace
`COUNT(<atr>)`	cuenta los valores **no** `NULL` de `<atr>`
`COUNT(*)`	cuenta **filas** (a diferencia de `COUNT(<atr>)`, sí cuenta filas con `NULL`)
`SUM(<atr>)`	suma los valores (ignora los `NULL`)
`MIN(<atr>)` / `MAX(<atr>)`	mínimo / máximo (ignora los `NULL`)
`AVG(<atr>)`	promedio (los `NULL` no entran ni al numerador ni al denominador)

`AGG(DISTINCT <atr>)` (ej. `COUNT(DISTINCT genero)`) agrega sin contar valores repetidos.

Otras funciones de agregación (SQLite, no vistas en clase)

Función	Qué hace
`GROUP_CONCAT(<atr>, <sep>)`	concatena los valores de un grupo en un solo string, separados por `<sep>` (por defecto `,`)
`TOTAL(<atr>)`	como `SUM`, pero **nunca da** `NULL`: si el grupo está vacío, da `0.0`

```sql
SELECT GROUP_CONCAT(titulo, ', ') FROM juegos WHERE genero = 'Plataformas';
-- devuelve un solo string con todos los títulos de ese género, separados por ', '
```

No son parte del estándar SQL, sino de SQLite (en Postgres el equivalente de `GROUP_CONCAT` es `STRING_AGG`). Revisen la documentación del motor que usen.

Agregación sin `GROUP BY`: toda la tabla es un grupo

```sql
SELECT COUNT(*) AS cantidad, ROUND(AVG(precio)) AS precio_promedio
FROM juegos;
```

Explicación: sin `GROUP BY`, SQL trata **toda la tabla ya filtrada por el `WHERE`** como un único grupo, y cada `AGG` se calcula sobre ese grupo completo. Por eso el resultado siempre tiene **una sola fila**: es el caso particular de `GROUP BY` donde hay exactamente un grupo, en vez de varios.

¡Ojo! Atributos sin agregar, sin `GROUP BY`

```sql
SELECT genero, MIN(precio) AS min_precio
FROM juegos
WHERE genero = 'RPG';
```

Aunque hoy `genero` sea siempre `'RPG'` en las filas filtradas, esta consulta es **ERROR** en Postgres (SQLite a veces lo deja pasar, pero **no deberían escribirlo así**): `genero` no está agregado ni está en un `GROUP BY`, así que el motor no sabe qué valor de `genero` mostrar por fila.

Regla: todo atributo en el `SELECT` de una consulta con agregación debe estar agregado (`AGG(...)`) o estar en el `GROUP BY`.

¿Por qué existe esta regla?

`GROUP BY` colapsa muchas filas en una sola fila por grupo. Para cada columna del `SELECT`, el motor necesita decidir **un único valor** por grupo:

- Si la columna está agregada (`AGG(...)`), no hay problema: la función reduce todos los valores del grupo a uno solo.
- Si la columna está en el `GROUP BY`, tampoco hay problema: por definición, todas las filas de un mismo grupo comparten ese valor.
- Si la columna **no es ninguna de las dos cosas**, dentro de un mismo grupo puede haber varios valores distintos, y no hay ninguna regla para elegir cuál mostrar.

Ejemplo: por qué esto es un error

```sql
SELECT genero, titulo, COUNT(*) AS cantidad
FROM juegos
GROUP BY genero;
```

`genero` está en el `GROUP BY` y `COUNT(*)` está agregado, pero `titulo` no es ninguna de las dos cosas. Dentro del grupo `'RPG'` hay varios títulos distintos (asumiendo que hay más de un juego de ese género), así que no hay un único `titulo` para esa fila resumen.

`GROUP BY`

```
SELECT <atr1>, AGG(<atr2>)                SELECT <atr1>, AGG(<atr2>) FROM ... WHERE ... AND <atr1>='v1'
FROM <lista-relaciones>                   UNION
WHERE <condicion>                    ≡    SELECT <atr1>, AGG(<atr2>) FROM ... WHERE ... AND <atr1>='v2'
GROUP BY <atr1>                           UNION ...  (uno por cada valor v de <atr1>)
```

`GROUP BY` arma un grupo por cada valor distinto de `<atr1>`, y calcula `AGG(<atr2>)` **dentro de cada grupo por separado**.

Orden de ejecución de una consulta SQL

Orden de ejecución de una consulta SQL (Clase 05):

```
1. FROM
2. WHERE
3. GROUP BY
4. HAVING
5. SELECT
```

Primero se arma el `FROM`, luego `WHERE` filtra filas, después se agrupa con `GROUP BY`, `HAVING` filtra esos grupos, y recién al final `SELECT` decide qué columnas mostrar. Este orden es clave para entender por qué `WHERE` y `HAVING` no son intercambiables.

`WHERE` vs. `HAVING`

```
SELECT <atr1>, AGG(<atr2>)
FROM <lista-relaciones>
WHERE <condicion-tupla>
GROUP BY <atr1>
HAVING <condicion-grupo>
```

Explicación: `WHERE` filtra **filas individuales**, antes de agrupar, por lo que no puede usar `AGG(...)` en su condición: los grupos todavía no existen. `HAVING` filtra **grupos completos**, después del `GROUP BY`, por lo que sí puede usar `AGG(...)`, como en `HAVING COUNT(*) > 2`.

`HAVING` ≡ `UNION` de grupos filtrados

```
SELECT <atr1>, AGG(<atr2>)                SELECT <atr1>, AGG(<atr2>) FROM ... WHERE <condicion-tupla> AND <atr1>='v1'
FROM <lista-relaciones>                   “AND <condicion-grupo>”
WHERE <condicion-tupla>             ≡     UNION
GROUP BY <atr1>                           SELECT <atr1>, AGG(<atr2>) FROM ... WHERE <condicion-tupla> AND <atr1>='v2'
HAVING <condicion-grupo>                  “AND <condicion-grupo>”
                                           UNION ...
```

La condición de `HAVING` es a nivel de grupo, y NO de tupla: en la versión con `UNION` no es un `AND` más dentro del `WHERE` de cada subconsulta, sino un filtro que se aplica sobre el resultado agregado (`AGG(<atr2>)`) de cada grupo `v1, v2, ...` de `<atr1>`.

Ejercicios

Agregación

Ejercicio 1: Enunciado

"Muestra el nombre de cada desarrollador, la cantidad de juegos que tiene y el precio promedio de esos juegos (alias `cantidad` y `precio_promedio`), pero solo para los desarrolladores cuyo precio promedio sea mayor al precio promedio de toda la tienda. Ordena de mayor a menor precio promedio."

```
desarrolladores(id_desarrollador, nombre, pais)                                          ar = 3
juegos(id_juego, titulo, id_desarrollador, genero, plataforma, precio, anio_lanzamiento, rating)    ar = 8
compras(id_compra, id_juego, nombre_comprador, fecha_compra)                             ar = 4
```

Ejercicio 1: Solución

```sql
SELECT d.nombre, COUNT(*) AS cantidad, ROUND(AVG(j.precio)) AS precio_promedio
FROM desarrolladores AS d JOIN juegos AS j
  ON d.id_desarrollador = j.id_desarrollador
GROUP BY d.id_desarrollador, d.nombre
HAVING AVG(j.precio) > (SELECT AVG(precio) FROM juegos)
ORDER BY precio_promedio DESC;
```

Ejercicio 1: Explicación

- El `HAVING` compara el promedio de cada grupo contra una **subconsulta escalar** (`SELECT AVG(precio) FROM juegos`, sin `GROUP BY`, así que da un único número) en vez de contra una constante fija como `HAVING COUNT(*) > 2`: es el mismo `HAVING`, pero el umbral también se calcula con una consulta.
- El `JOIN` con `desarrolladores` es necesario porque `juegos` solo guarda `id_desarrollador`, no el `nombre`: sin el join no tendríamos qué mostrar.
- `GROUP BY d.id_desarrollador, d.nombre`: agrupamos por la clave real (`id_desarrollador`), pero como `nombre` también aparece sin agregar en el `SELECT`, tiene que estar en el `GROUP BY`.
- **`Bethesda Game Studios` nunca puede aparecer en este resultado:** al no tener ningún juego, el `JOIN` lo descarta por completo antes de llegar al `GROUP BY`.

Ejercicio 2: Enunciado

"Muestra el nombre de los desarrolladores que tienen más juegos que el promedio de juegos por desarrollador en toda la tienda."

```
desarrolladores(id_desarrollador, nombre, pais)                                          ar = 3
juegos(id_juego, titulo, id_desarrollador, genero, plataforma, precio, anio_lanzamiento, rating)    ar = 8
compras(id_compra, id_juego, nombre_comprador, fecha_compra)                             ar = 4
```

Ejercicio 2: Solución

```sql
SELECT d.nombre, COUNT(*) AS cantidad
FROM desarrolladores AS d JOIN juegos AS j
  ON d.id_desarrollador = j.id_desarrollador
GROUP BY d.id_desarrollador, d.nombre
HAVING COUNT(*) > (
    SELECT AVG(cantidad_por_dev)
    FROM (
        SELECT COUNT(*) AS cantidad_por_dev
        FROM juegos
        WHERE id_desarrollador IS NOT NULL
        GROUP BY id_desarrollador
    ) AS conteos
);
```

Ejercicio 2: Explicación

- SQL no permite anidar funciones de agregación directamente (`AVG(COUNT(*))` es un error): por eso primero hay que calcular el `COUNT(*)` **por desarrollador** en una subconsulta del `FROM`, y recién en la consulta de afuera de esa promediar esos conteos con `AVG`.
- La subconsulta más interna (`SELECT COUNT(*) AS cantidad_por_dev FROM juegos GROUP BY id_desarrollador`) da una tabla con una fila por desarrollador: cuántos juegos tiene. La subconsulta que la envuelve (`AVG(cantidad_por_dev)`) promedia esa columna, dando un único número: el promedio de juegos por desarrollador en toda la tienda.
- El `HAVING COUNT(*) > (...)` de la consulta principal es el mismo patrón del Ejercicio 1, pero acá el "umbral" no es un `AVG` directo sobre una tabla: primero hay que agrupar y contar, y recién ahí promediar eso.

`∀` en consultas de primer orden

SQL, álgebra relacional y lógica de primer orden son igual de expresivos (**Teorema de Codd**). Hoy usamos ese puente para el caso que se pone interesante: la cuantificación **universal**.

SQL no tiene `FORALL`, así que cada `∀` se traduce a un `NOT EXISTS`:

$$\forall x.\, \varphi(x) \equiv \neg \exists x.\, \neg \varphi(x)$$

"Para todo x se cumple ϕ" es lo mismo que "no existe un x que sea **contraejemplo** de ϕ": buscamos el contraejemplo y verificamos que no exista.

Ejercicio 3: Enunciado (Propuesto)

"Muestra (en SQL, Algebra relacional y en Consultas de Primer Orden) el nombre de los desarrolladores que tienen al menos un juego en `'PC'` Y al menos un juego en `'PS5'` (presencia en ambas plataformas), usando `NOT EXISTS` anidado."

```
desarrolladores(id_desarrollador, nombre, pais)                                          ar = 3
juegos(id_juego, titulo, id_desarrollador, genero, plataforma, precio, anio_lanzamiento, rating)    ar = 8
compras(id_compra, id_juego, nombre_comprador, fecha_compra)                             ar = 4
```

Ejercicio 3: Solución (SQL)

```sql
SELECT d.nombre
FROM desarrolladores AS d
WHERE NOT EXISTS (
    SELECT 1
    FROM (SELECT 'PC' AS plataforma UNION ALL SELECT 'PS5') AS p
    WHERE NOT EXISTS (
        SELECT 1
        FROM juegos AS j
        WHERE j.id_desarrollador = d.id_desarrollador
          AND j.plataforma = p.plataforma
    )
);
```

Ejercicio 3: Solución (Consultas de Primer Orden)

$$PL(p) \leftarrow p = \text{'PC'} \vee p = \text{'PS5'}$$

$$HasP(i, p) \leftarrow \exists j\, \exists t\, \exists g\, \exists pr\, \exists a\, \exists r.\ juegos(j, t, i, g, p, pr, a, r)$$

$$Q(n) \leftarrow \exists i\, \exists pa.\ desarrolladores(i, n, pa) \wedge \neg \exists p.\ \big(PL(p) \wedge \neg HasP(i, p)\big)$$

En palabras: "no existe una plataforma de la lista en la que este desarrollador NO tenga ningún juego".

Ejercicio 3: Solución (Álgebra relacional)

$$R \leftarrow \pi_{id_desarrollador,\, plataforma}(juegos)$$

$$S \leftarrow \{\text{'PC'},\ \text{'PS5'}\} \quad (\text{relación literal de un atributo})$$

$$Ids \leftarrow \pi_{id_desarrollador}(R) - \pi_{id_desarrollador}\big((\pi_{id_desarrollador}(R) \times S) - R\big)$$

$$out \leftarrow \pi_{nombre}\big(desarrolladores \bowtie \rho_{Ids}(Ids)\big)$$

Ids es exactamente la **división relacional** (investigar) $R \div S$: los `id_desarrollador` que tienen pareja en R para **cada** valor de S.

Ejercicio 3: Explicación (1/2)

- La subconsulta externa (SQL) / el conjunto PL o S (lógica/álgebra) enumera el conjunto sobre el que cuantificamos: acá `{'PC', 'PS5'}`. La interna busca un juego de `d` en cada elemento; si alguna plataforma no tiene ningún juego de `d`, es el "contraejemplo" y el `NOT EXISTS` externo falla.
- Por eso el `NOT EXISTS` de afuera y el `NOT EXISTS` de adentro **no juegan el mismo rol**: el de afuera busca el contraejemplo (la plataforma problema), el de adentro pregunta si esa plataforma en particular no tiene juego.

Ejercicio 3: Explicación (2/2)

- **En lógica, negar es directo:** basta anteponer `¬` a `HasP(i,p)`. **En álgebra no existe un operador de negación sobre fórmulas**, así que la universalidad se reconstruye con diferencia de conjuntos (`-`): por eso Ids necesita el patrón "producto cruz, menos, proyectar, menos" en vez de una negación directa.
- `Bethesda Game Studios` **nunca puede aparecer en este resultado:** al no tener ningún juego, ya falla para la primera plataforma que se pruebe (`HasP` es falso para toda `p`), así que siempre existe un contraejemplo.
- El mismo patrón, cambiando `{'PC', 'PS5'}` por `SELECT DISTINCT plataforma FROM juegos` (o, en álgebra, $S \leftarrow \pi_{plataforma}(juegos)$), respondería "presencia en **todas** las plataformas que existan en la tienda", sin asumir cuántas hay ni cuáles son.

Repaso

Nulos y lógica de tres valores (Clase 06)

`NULL` en SQL

- `NULL` puede significar: el valor existe pero no lo sabemos, o el valor simplemente no aplica.
- Para preguntar si un valor es `NULL`, se usan predicados que sí devuelven `TRUE` / `FALSE`:

```
WHERE rating IS NULL
WHERE rating IS NOT NULL
```

¡Ojo! `WHERE rating = NULL` **está mal**: nunca es verdadero, porque comparar con `NULL` da `UNKNOWN`, no `TRUE`. Usen siempre `IS NULL` / `IS NOT NULL`.

SQL: lógica de tres valores

Tres valores de verdad: `true`, `false`, `unknown`.

Semántica de `WHERE`: una fila queda **solo si la condición da `true`** (no basta con "no ser `false`"; `unknown` también se descarta).

`NOT true = false`, `NOT false = true`, `NOT unknown = unknown`.

AND	false	true	unknown
unknown	false	unknown	unknown
false	false	false	false
true	false	true	unknown

OR	false	true	unknown
unknown	unknown	true	unknown
false	false	true	unknown
true	true	true	true

Operaciones vs. predicados con `NULL`

```
null ⊛ a  =  a ⊛ null  =  op(null)  :=  null        para +, *, /, %, ... (una operación)
null ~ a  =  a ~ null  =  P(null)   :=  unknown     para =, <, ≤, ... (un predicado/comparación)
```

Ejemplos (con `rating(Dota 2) = NULL`, `precio(Dota 2) = 0`):

```
rating(Dota 2) + 10 = 103                          → null = 103            → unknown
(rating(Dota 2) IS NULL) AND precio(Dota 2) = 0     → true AND true         → true
rating(Dota 2) = rating(Pyre)                      → null = null           → unknown
rating(Dota 2) > 50 OR precio(Dota 2) = 0          → unknown OR true       → true
```

`IS NULL` es un predicado especial: **siempre** da `true` o `false`, nunca `unknown` (a diferencia de `= NULL`).

Ejercicios

Nulos y lógica de tres valores

Ejercicio 4: Enunciado

"Evalúa paso a paso el valor de verdad de cada condición, sabiendo que `rating(Hollow Knight) = NULL`, `precio(Hollow Knight) = 14990`, `rating(Celeste) = 92` y `precio(Celeste) = 12990`:"

a) `rating(Hollow Knight) > 80`

b) `rating(Hollow Knight) IS NULL`

c) `rating(Hollow Knight) = rating(Celeste)`

d) `NOT (rating(Hollow Knight) > 80)`

e) `(rating(Hollow Knight) IS NULL OR rating(Hollow Knight) > 80) AND precio(Hollow Knight) < precio(Celeste)`

```
desarrolladores(id_desarrollador, nombre, pais)                                          ar = 3
juegos(id_juego, titulo, id_desarrollador, genero, plataforma, precio, anio_lanzamiento, rating)    ar = 8
compras(id_compra, id_juego, nombre_comprador, fecha_compra)                             ar = 4
```

Ejercicio 4: Solución

```
a) rating(Hollow Knight) > 80
   →  null > 80  →  unknown

b) rating(Hollow Knight) IS NULL
   →  true

c) rating(Hollow Knight) = rating(Celeste)
   →  null = 92  →  unknown

d) NOT (rating(Hollow Knight) > 80)
   →  NOT (unknown)  →  unknown

e) (rating(Hollow Knight) IS NULL OR rating(Hollow Knight) > 80)
   AND precio(Hollow Knight) < precio(Celeste)
   →  (true OR unknown) AND (14990 < 12990)
   →  true AND false
   →  false
```

Ejercicio 5: Enunciado (Propuesto)

"Quieres mostrar el título de los juegos cuyo `rating` NO está entre los ratings de los juegos de género `'RPG'`, así que escribes esta consulta con `NOT IN`:"

```
SELECT titulo FROM juegos
WHERE rating NOT IN (
    SELECT rating FROM juegos WHERE genero = 'RPG'
);
```

¿Por qué esta consulta es riesgosa en nuestra base de datos de hoy? Reescríbela de forma segura usando `NOT EXISTS` correlacionado.

```
desarrolladores(id_desarrollador, nombre, pais)                                          ar = 3
juegos(id_juego, titulo, id_desarrollador, genero, plataforma, precio, anio_lanzamiento, rating)    ar = 8
compras(id_compra, id_juego, nombre_comprador, fecha_compra)                             ar = 4
```

Ejercicio 5: Solución

Por qué es riesgosa: en nuestra base de datos de hoy, algunos juegos no tienen rating (`rating = NULL`). Si **al menos un** RPG tiene `rating IS NULL`, la subconsulta del `IN` devuelve una lista con un `NULL`, por ejemplo `{75, 88, null}`.

`x NOT IN (v1, ..., null)` equivale a `x <> v1 AND ... AND x <> null`. Como `x <> null` siempre da `unknown`, y `true AND unknown = unknown`, el `AND` completo nunca puede dar `true`. Como `WHERE` solo conserva filas donde la condición es `true`, **la consulta devuelve el conjunto vacío**, sin importar el `rating` del juego evaluado: un error silencioso.

```
SELECT j.titulo
FROM juegos AS j
WHERE NOT EXISTS (
    SELECT 1
    FROM juegos AS r
    WHERE r.genero = 'RPG'
      AND r.rating = j.rating
);
```

Ejercicio 5: Explicación

- A diferencia del `NOT IN` (que arma una lista completa y compara contra **todos** sus elementos con un único `AND` global), el `NOT EXISTS` correlacionado evalúa `r.rating = j.rating` **fila por fila**, para cada RPG `r` por separado.
- Si algún RPG tiene `rating IS NULL`, esa comparación da `unknown` **solo para esa fila** de `r`: no contamina el resultado de las demás filas, ni bloquea el `EXISTS` / `NOT EXISTS` completo (`EXISTS` solo necesita encontrar *una* fila en `true`).
- **Caso borde:** si el juego `j` evaluado tampoco tiene rating, `r.rating = j.rating` da `unknown` para cualquier `r`, así que nunca se encuentra una fila y `NOT EXISTS` da `true`: ese juego sí aparece en el resultado (razonable: un juego sin rating no puede estar "entre" los ratings de los RPG).

Repaso

Outer-joins (Clase 06)

Datos opcionales: el problema del `JOIN`

```sql
SELECT d.nombre, j.titulo
FROM desarrolladores AS d JOIN juegos AS j
  ON d.id_desarrollador = j.id_desarrollador;
```

Un `JOIN` (inner join) **descarta** las filas de un lado que no tienen pareja en el otro. Con nuestros datos de hoy: `Bethesda Game Studios` (sin juegos) **no aparece**, y `Undertale` (sin desarrollador) **tampoco**.

¿Cómo mostramos **todas** las filas de un lado, aunque no tengan pareja?

Outer-joins: la idea

Un outer-join hace lo mismo que un join, pero además agrega las filas sin pareja, **rellenando con** `NULL` los atributos del otro lado:

$$\bar{\tau}[X](A) := \begin{cases} \bar{\tau}(A) & \text{si } A \in \text{att}(\bar{\tau}) \\ \text{null} & \text{si } A \notin \text{att}(\bar{\tau}) \end{cases}$$

Tipo	Conserva filas sin pareja de...
`LEFT OUTER JOIN`	la tabla de la izquierda
`RIGHT OUTER JOIN`	la tabla de la derecha
`FULL OUTER JOIN`	ambas tablas

SQLite soporta los tres desde la versión 3.39.

Outer-joins en SQL

```sql
SELECT d.nombre, j.titulo
FROM desarrolladores AS d LEFT OUTER JOIN juegos AS j
  ON d.id_desarrollador = j.id_desarrollador;
```

Con `LEFT OUTER JOIN`, `Bethesda Game Studios` aparece con `titulo = NULL`. Con `RIGHT OUTER JOIN` en el mismo `ON`, en cambio, sería `Undertale` quien aparecería con `nombre = NULL`. Con `FULL OUTER JOIN` aparecen **ambos** casos.

Idea clave: un `NULL` que aparece en una columna que en las filas "normales" nunca es `NULL` (como `titulo` o `id_juego`) es la señal de que esa fila **no tuvo pareja**.

Ejercicios

Outer-joins

Ejercicio 6: Enunciado

"Muestra el nombre de los desarrolladores que aún no tienen ningún juego cargado en la tienda."

```
desarrolladores(id_desarrollador, nombre, pais)                                          ar = 3
juegos(id_juego, titulo, id_desarrollador, genero, plataforma, precio, anio_lanzamiento, rating)    ar = 8
compras(id_compra, id_juego, nombre_comprador, fecha_compra)                             ar = 4
```

Ejercicio 6: Solución

```sql
SELECT d.nombre
FROM desarrolladores AS d LEFT JOIN juegos AS j
  ON d.id_desarrollador = j.id_desarrollador
WHERE j.id_juego IS NULL;
```

```
nombre
Bethesda Game Studios
```

Explicación: el `LEFT JOIN` conserva a todos los desarrolladores. Los que no tienen ningún juego quedan con `j.id_juego = NULL` (porque `id_juego` nunca es `NULL` en una fila "real" de `juegos`). Filtrar por `j.id_juego IS NULL` aísla justo esas filas: es el patrón estándar para "encontrar lo que no tiene pareja".

¿Y si usamos un producto cruz en vez de `LEFT JOIN`? No funciona: el producto cruz `desarrolladores × juegos` solo genera combinaciones de filas que ya existen en ambas tablas, nunca "inventa" una fila con `NULL` para un desarrollador sin pareja.

Ejercicio 7: Enunciado (Propuesto)

"Muestra el título de los juegos que aún no tienen un desarrollador asignado, usando `RIGHT OUTER JOIN`."

```
desarrolladores(id_desarrollador, nombre, pais)                                          ar = 3
juegos(id_juego, titulo, id_desarrollador, genero, plataforma, precio, anio_lanzamiento, rating)    ar = 8
compras(id_compra, id_juego, nombre_comprador, fecha_compra)                             ar = 4
```

Ejercicio 7: Solución

```sql
SELECT j.titulo
FROM desarrolladores AS d RIGHT JOIN juegos AS j
  ON d.id_desarrollador = j.id_desarrollador
WHERE d.id_desarrollador IS NULL;
```

```
titulo
Undertale
```

Explicación: es el mismo patrón del Ejercicio 6, pero "al revés": ahora es `juegos` quien se conserva completo, y `desarrolladores` el que se rellena con `NULL` cuando no hay pareja. `RIGHT JOIN R2 ON ...` es exactamente lo mismo que `LEFT JOIN` invirtiendo el orden de las tablas, por eso `RIGHT OUTER JOIN` es el operador menos usado en la práctica: casi todo se puede escribir con `LEFT JOIN` reordenando el `FROM`.

Ejercicio 8: Enunciado

"Para cada juego, muestra su título y la cantidad de compras que ha tenido (debe salir `0` para los que nunca se han comprado)."

```
desarrolladores(id_desarrollador, nombre, pais)                                          ar = 3
juegos(id_juego, titulo, id_desarrollador, genero, plataforma, precio, anio_lanzamiento, rating)    ar = 8
compras(id_compra, id_juego, nombre_comprador, fecha_compra)                             ar = 4
```

Ejercicio 8: Solución

```sql
SELECT j.titulo, COUNT(c.id_compra) AS cantidad_compras
FROM juegos AS j LEFT JOIN compras AS c ON j.id_juego = c.id_juego
GROUP BY j.id_juego, j.titulo;
```

Ejercicio 8: Explicación

Si hubiéramos usado un `JOIN` normal, los juegos nunca comprados **no aparecerían en absoluto** (ni con un 0).

Con `LEFT JOIN` sí aparecen, con `c.id_compra = NULL`; y como `COUNT` **no cuenta los** `NULL` (Clase 05), `COUNT(c.id_compra)` da exactamente `0` para ellos.

Si hubiéramos usado `COUNT(*)` en vez de `COUNT(c.id_compra)`, ¡el resultado habría sido `1` para esos juegos, no `0`!

Ejercicio 9: Enunciado (Propuesto)

"Usando `FULL OUTER JOIN` entre `desarrolladores` y `juegos`, muestra las filas donde falte información de un lado (desarrollador sin juego, o juego sin desarrollador)."

```
desarrolladores(id_desarrollador, nombre, pais)                                          ar = 3
juegos(id_juego, titulo, id_desarrollador, genero, plataforma, precio, anio_lanzamiento, rating)    ar = 8
compras(id_compra, id_juego, nombre_comprador, fecha_compra)                             ar = 4
```

Ejercicio 9: Solución

```sql
SELECT d.nombre, j.titulo
FROM desarrolladores AS d FULL OUTER JOIN juegos AS j
  ON d.id_desarrollador = j.id_desarrollador
WHERE d.id_desarrollador IS NULL OR j.id_juego IS NULL;
```

```
nombre                  titulo
Bethesda Game Studios   null
null                    Undertale
```

Explicación: un `FULL OUTER JOIN` es la unión de un `LEFT` y un `RIGHT OUTER JOIN`: conserva **todas** las filas de ambos lados. Filtrando por `d.id_desarrollador IS NULL OR j.id_juego IS NULL` nos quedamos solo con las filas "incompletas": las que un `INNER JOIN` habría descartado por completo.

Dudas?