Bases de Datos

Clase 4: SQL Avanzado

Clase anterior

```
SELECT atributos
FROM relaciones
WHERE condiciones / selecciones
```

Además, existen operadores como `LIKE`, `DISTINCT`, `ORDER BY`, `UNION`, etc.

Agregación

¿Qué hace esta consulta?

```
SELECT AVG(precio)
FROM Productos
WHERE fabricante = ‘Toyota’
```

También podemos usar SUM, MIN, MAX, COUNT, etc.

Agregación

¿Qué diferencia estas consultas?

```
SELECT COUNT(*)
FROM Productos
WHERE año > 2012
```

```
SELECT COUNT(fabricante)
FROM Productos
WHERE año > 2012
```

`COUNT(*)` cuenta los nulos. Más de eso en un rato

Ojo: En ambas se cuentan los duplicados

Ejemplo

Compra(producto, fecha, precio, cantidad)

producto	fecha	precio	cantidad
tomates	07/02	100	6
tomates	06/07	150	
zapallo	08/02	800	1
zapallo	09/07	1000	
zapallo	01/01	600	1

```
SELECT COUNT(*)
FROM Compra
```

5

```
SELECT COUNT(cantidad)
FROM Compra
```

3

Ejemplo

Compra(producto, fecha, precio, cantidad)

producto	fecha	precio	cantidad
tomates	07/02	100	6
tomates	06/07	150	
zapallo	08/02	800	1
zapallo	09/07	1000	
zapallo	01/01	600	1

```
SELECT DISTINCT COUNT(cantidad)
FROM Compra
```

3

```
SELECT COUNT(DISTINCT cantidad)
FROM Compra
```

2

GROUP BY

```
SELECT fabricante, COUNT(fabricante)
FROM Productos
WHERE año > 2012
GROUP BY fabricante
```

Esta consulta:

- Computa los resultados según el `FROM` y `WHERE`
- Agrupa los resultados según los atributos del `GROUP BY`
- Para cada grupo se aplica independientemente la agregación

Ejemplo

Compra(producto, fecha, precio, cantidad)

producto	fecha	precio	cantidad
tomates	07/02	100	6
tomates	06/07	150	4
zapallo	08/02	800	1
zapallo	09/07	1000	2
zapallo	01/01	600	3

Ejemplo

```
SELECT producto,
   SUM(precio*cantidad) as ventaTotal
FROM Compra
WHERE fecha > '10/01'
GROUP BY producto
```

```sql
SELECT producto, SUM(precio*cantidad) as ventaTotal
FROM Compra
WHERE fecha > '10/01'
GROUP BY producto
```

1) Se computa el FROM y el WHERE

producto	fecha	precio	cantidad
tomates	07/02	100	6
tomates	06/07	150	4
zapallo	08/02	800	1
zapallo	09/07	1000	2

```sql
SELECT producto, SUM(precio*cantidad) as ventaTotal
FROM Compra
WHERE fecha > '10/01'
GROUP BY producto
```

2) Agrupar según el GROUP BY

producto	fecha	precio	cantidad
tomates	07/02	100	6
	06/07	150	4
zapallo	08/02	800	1
	09/07	1000	2

```sql
SELECT producto, SUM(precio*cantidad) as ventaTotal
FROM Compra
WHERE fecha > '10/01'
GROUP BY producto
```

3) Agregar por grupo y ejecutar la proyección

producto	ventaTotal
tomates	1200
zapallo	2800

HAVING

Misma consulta, pero sólo queremos los productos que se vendieron más de 100 veces

```
SELECT producto, SUM(precio*cantidad) AS ventaTotal
FROM Compra
WHERE fecha > ’10/01’
GROUP BY producto
HAVING SUM(cantidad) > 100
```

¿Por qué usamos HAVING y no lo incluimos en el WHERE?

Consultas con Agregación

```
SELECT <S>
FROM R1, ..., Rn
WHERE <Condición 1>
GROUP BY a1, ..., ak
HAVING <Condición 2>
```

- S puede contener atributos de a_1, …, a_k y/o agregados, pero ningún otro atributo (¿Por qué?)
- Condición 1 es una condición que usa atributos de R_1, …, R_n
- Condición 2 es una condición de agregación de los atributos de R_1, …, R_n

Consultas con Agregación

Evaluación

```
SELECT <S>
FROM R1, …, Rn
WHERE <Condición 1>
GROUP BY a1, …, ak
HAVING <Condición 2>
```

- Se computa el FROM - WHERE de R_1, …, R_n
- Agrupar la tabla por los atributos de a_1, …, a_k
- Computar los agregados de la Condición 2 y mantener grupos que satisfacen
- Computar agregados de S y entregar el resultado

Consultas Anidadas

- Como ya habíamos visto con las operaciones de conjuntos, una consulta puede estar constituida por operaciones entre consultas.

- Pero esa no es la única forma, SQL nos ofrece mucho más.

Consultas Anidadas

Como condición

Consideremos este esquema:

Bandas(nombre, vocalista, ...)

Estudiantes_UC(nombre, ...)

Toco_en(nombre_banda, nombre_festival)

Obtengamos todas las bandas cuyo vocalista sea un estudiante UC y que hayan tocado en lollapallooza

Consultas Anidadas

Como condición

```
SELECT Bandas.nombre
FROM Bandas, Estudiantes_UC
WHERE Bandas.vocalista = Estudiantes_UC.nombre
AND Bandas.nombre IN (
    SELECT Toco_en.nombre_banda
    FROM Toco_en
    WHERE Toco_en.nombre_festival = ‘Lollapalooza’
)
```

Comprobamos que Bandas.nombre esté dentro del listado de bandas que han tocado en Lollapalooza.

Consultas Anidadas

Como condición

Si la sub consulta retorna un escalar podemos usar las operadores condicionales típicos. Si no podemos hacerlo agregando un operador adicional.

- s `IN` R
- s > `ALL` R (no disponible en SQLite)
- s > `ANY` R (no disponible en SQLite)
- `EXISTS` R

Consultas Anidadas

ALL, ANY

Cervezas(nombre, precio, ...)

Cervezas más baratas que la Austral Calafate

```
SELECT Cervezas.nombre
FROM Cervezas
WHERE Cervezas.precio < ALL (
    SELECT Cervezas2.precio
    FROM Cervezas AS Cervezas2
    WHERE Cervezas2.nombre = ‘Austral Calafate’
)
```

Consultas Anidadas

ALL, ANY

Cervezas que no son la más cara

```
SELECT Cervezas.nombre
FROM Cervezas
WHERE Cervezas.precio < ANY (
    SELECT Cervezas2.precio
    FROM Cervezas AS Cervezas2
)
```

Consultas Anidadas

Podemos expresar estas consultas con `SELECT` - `FROM` - `WHERE`?

Hint: Las consultas SFW son **monótonas**. Una consulta con `ALL` no es monótona. Una consulta con `ANY` lo es

Sub Consultas Relacionadas

Nombres de películas que se repiten en años diferentes

```
SELECT p.nombre
FROM Películas AS p
WHERE p.año <> ANY (
    SELECT año
    FROM Películas
    WHERE nombre = p.nombre
)
```

La sub consulta depende de la externa!

Consultas Anidadas

Como Joins

El nombre de cada actor junto con el total de películas en las que ha actuado.

```
SELECT Actores.nombre, agg.cuenta
FROM Actores, (
    SELECT id_actor as id, COUNT(*) as cuenta
    FROM Actuo_en
    GROUP BY id_actor
) as agg
WHERE Actores.id = agg.id
```

Consultas Anidadas

Como Joins

El nombre de cada actor junto con el año de la primera película en la que actuó.

```
SELECT Actores.nombre, agg.año
FROM Actores, (
    SELECT Actuo_en.id_actor as id, MIN(Peliculas.año) as año
    FROM Actuo_en, Peliculas
    WHERE Actuo_en.id_pelicula = Peliculas.id
    GROUP BY id_actor
) as agg
WHERE Actores.id = agg.id
```

¿Qué pasa si queremos el nombre de la película además del año?

Información Incompleta

- En una base de datos real, muy seguido no tendremos los datos para llenar todas las columnas al agregar una fila.

- También puede ser que por la lógica del problema, que un campo esté vacío tenga una semántica relevante para la aplicación.

- Con SQL podemos modelar la falta de información mediante nulos (`NULL`).

- Los nulos en las tablas generan ciertos comportamientos extraños que es bueno tener en cuenta al trabajar con ellos. Los discutiremos en esta clase.

Información Incompleta

Peliculas	titulo	director	actor
	Django sin cadenas	Tarantino	Di Caprio
	Django sin cadenas	Tarantino	Waltz
	null	Tarantino	Thurman
	null	Tarantino	null
	El Hobbit	Jackson	McKellen
	Señor de los Anillos	null	McKellen

¿Qué significan los nulos en este caso?

Nulos

Significado

En el caso anterior puede significar que no se dispone de la información, pero existe!

En general los nulos pueden significar:

- Valor existe, pero no tengo la información
- Valor no existe (si premios = null la información no existe)
- Ni siquiera sé si el valor existe o no

Nulos

Consultando con nulos

Sea la relación **R**(a, b), las consultas:

- `SELECT * FROM R`
- `SELECT * FROM R WHERE R.b = 3 OR R.b <> 3`

¿Son lo mismo?

Si R.b es nulo, ni R.b = 3 o R.b <> 3 evalúan a verdadero

Nulos

Consultando con nulos

La consulta:

```
SELECT * FROM R
```

Equivale a la unión de:

- `SELECT * FROM R WHERE R.b = 3`
- `SELECT * FROM R WHERE R.b <> 3`
- `SELECT * FROM R WHERE R.b IS NULL`

Para ver si un elemento es nulo usamos `IS NULL`

Para ver si un elemento no es nulo usamos `IS NOT NULL`

Nulos

Operaciones con nulos

Si algún argumento de una operación aritmética es nulo, el resultado es nulo

Nulos

Operaciones con nulos

Sean las siguientes instancias de **R** y **S**:

R A	S B
1	2
null	3

La consulta `SELECT R.A + S.B FROM R, S` retorna:

Respuesta $R.A + S.B$
3
4
null
null

Nulos

Operaciones con nulos

Sean las siguientes instancias de **R** y **S**:

R	A	S	B
	1		2
	null		3

Tenemos que R.A = S.B vale:

- FALSE cuando R.A = 1 y S.B = 2
- UNKNOWN cuando R.A = null y S.B = 2

Lógica de tres valores

SQL usa lógica de tres valores:

x	NOT x
true	false
false	true
unknown	unknown

Lógica de tres valores

SQL usa lógica de tres valores:

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

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

Nulos

Ejemplo

R A
1
2

S A
1
2
3
4

```
SELECT S.A FROM S
WHERE S.A NOT IN (SELECT R.A FROM R)
```

Resultado: {3, 4}

Nulos

Ejemplo

R A
1
2
null

S A
1
2
3
4

```
SELECT S.A FROM S
WHERE S.A NOT IN (SELECT R.A FROM R)
```

Resultado: tabla vacía!

Lógica de tres valores

El resultado puede ser contraintuitivo pero es correcto

Veamos la evaluación de 3 NOT IN (SELECT R.A FROM R)

$$
\begin{aligned}
3 \ \texttt{NOT IN} \ & \{1, 2, \mathit{null}\} \\
= \ & \texttt{NOT (3 IN } \{1, 2, \mathit{null}\} \\
= \ & \texttt{NOT (3} = 1 \ \texttt{OR} \ 3 = 2 \ \texttt{OR} \ 3 = \mathit{null}\texttt{)} \\
= \ & \texttt{NOT (false OR false OR unknown)} \\
= \ & \texttt{NOT (unknown)} \\
= \ & \texttt{unknown}
\end{aligned}
$$

Lógica de tres valores

Además `1 NOT IN (SELECT R.A FROM R)` es `FALSE` (Lo mismo para 2)

Para el valor 4 aplicamos el razonamiento anterior.

Resultado: tabla vacía

Nulos

Agregación

A
1
null

```
SELECT COUNT(*) FROM R
```
vale 2

```
SELECT COUNT(R.A) FROM R
```
vale 1

Nulos

Agregación

`SELECT SUM(R.A) FROM R` retorna 1 si `R.A = {1,null}`

Para funciones de agregación:

- Ignore todos los nulos
- Compute el valor de la agregación

La única excepción es COUNT(*)

Nulos

Al crear una tabla o agregar una columna podemos incluir una restricción para que no permita NULLs

```
CREATE TABLE <nombre> (...
    <atributo> <tipo> NOT NULL,
...)
```

Inner Joins

Usualmente hacemos JOINS, especificando en la sentencia `FROM` de la consulta las tablas que queremos usar y en el `WHERE` las condiciones.

```
SELECT *
FROM Peliculas, Actuo_en
WHERE id = id_pelicula
```

Pero SQL tiene una sintaxis mucho más clara para expresar un join normal: `INNER JOIN` (alias `JOIN`)

Inner Joins

Estas 3 consultas son equivalentes:

```
SELECT *
FROM Peliculas,
Actuo_en
WHERE id = id_pelicula
```

```
SELECT *
FROM Peliculas JOIN Actuo_en
ON id = id_pelicula
```

```
SELECT *
FROM Peliculas INNER JOIN Actuo_en
ON id = id_pelicula
```

Outer Joins

Consideremos estas tablas:

Estudios

Nombre	Titulo
Warner	Argo
Warner	El Origen
MGM	El Hobbit

Peliculas

Titulo	Ingreso
Argo	136
El Origen	292
El Artista	44

Escribamos una consulta que liste los ingresos totales de cada estudio.

Outer Joins

```
SELECT Estudio.nombre, SUM(Pelicula.ingreso)
FROM Estudio JOIN Pelicula
ON Estudio.titulo = Pelicula.titulo
GROUP BY Estudio.nombre
```

- Resultado: (Warner, 428)

¿Cuál es el problema de esta consulta?

El estudio MGM, cuyas películas no tenemos información va a desaparecer por no tener contraparte en el `JOIN`.

Outer Joins

Lo solucionamos con un Outer Join izquierdo, que mantiene las tuplas sin pareja de la primera tabla:

```
SELECT Estudio.nombre, SUM(Pelicula.ingreso)
FROM Estudio LEFT OUTER JOIN Pelicula
ON Estudio.titulo = Pelicula.titulo
GROUP BY Estudio.nombre
```

- Resultado: (Warner, 428), (MGM, null)

Left Outer Join

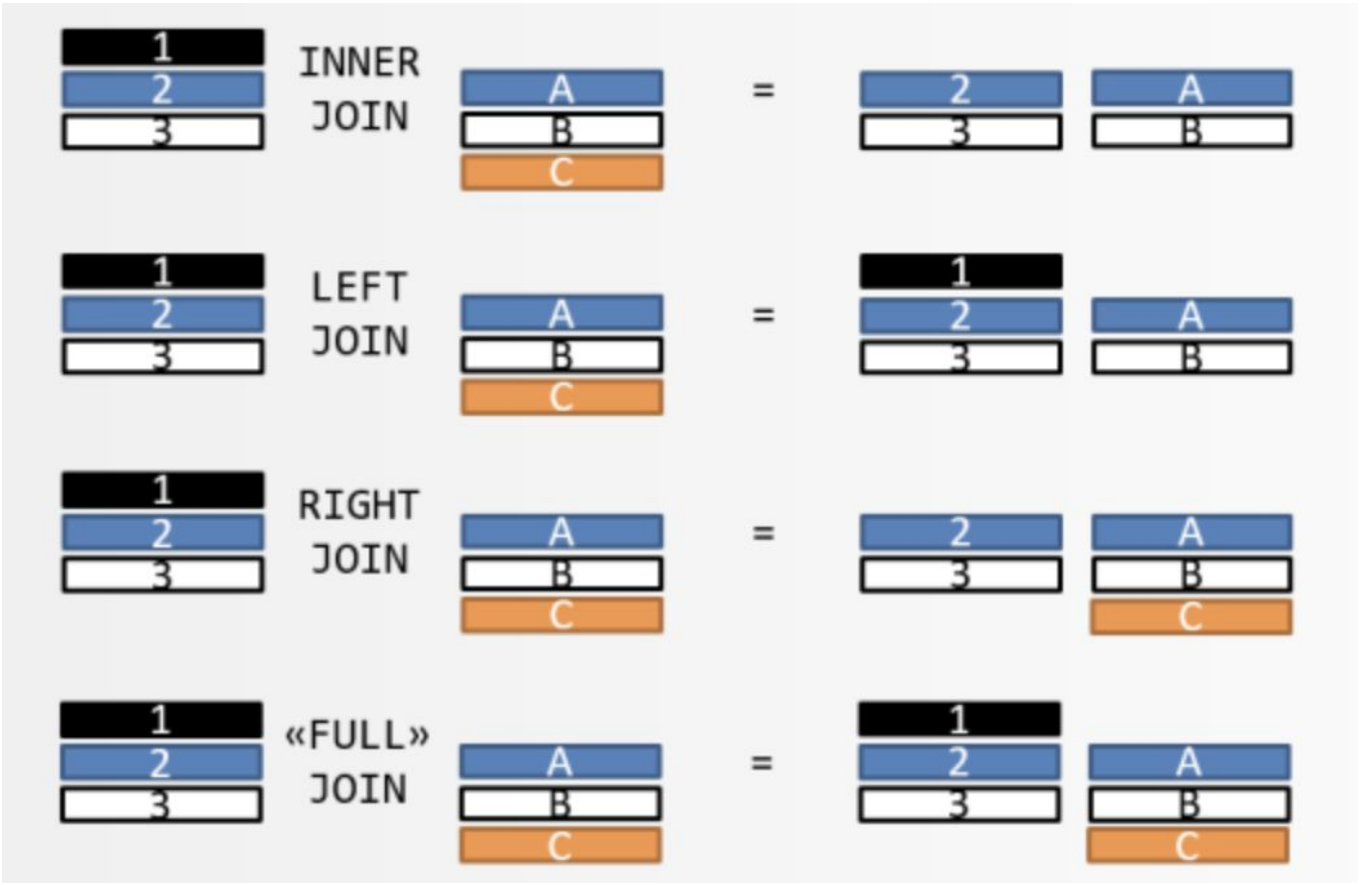
```
SELECT *
FROM Estudio LEFT OUTER JOIN Pelicula
ON Estudio.titulo = Pelicula.titulo
```

nombre	titulo	titulo	ingreso
Warner	Argo	Argo	136
Warner	El Origen	El Origen	292
MGM	El Hobbit	null	null

Outer Joins

- `R LEFT OUTER JOIN S`: mantenemos las tuplas de **R** que no tienen correspondencia.
- `R RIGHT OUTER JOIN S`: mantenemos las tuplas de **S** que no tienen correspondencia.
- `R FULL OUTER JOIN S`: mantenemos las tuplas de **R** y **S** que no tienen correspondencia

Más detalles acá y acá



[https://blog.jooq.org/say-no-to-venn-diagrams-when-explaining-joins/]

Redundancia en SQL

Recordemos esta consulta:

```
SELECT Bandas.nombre
FROM Bandas, Estudiantes_UC
WHERE Bandas.vocalista = Estudiantes_UC.nombre
AND Bandas.nombre IN (
    SELECT Toco_en.nombre_banda
    FROM Toco_en
    WHERE Toco_en.nombre_festival = ‘Lollapalooza’
)
```

```sql
SELECT Bandas.nombre
FROM Bandas, Estudiantes_UC
WHERE Bandas.vocalista = Estudiantes_UC.nombre
AND Bandas.nombre IN (
    SELECT Toco_en.nombre_banda
    FROM Toco_en
    WHERE Toco_en.nombre_festival = 'Lollapalooza'
)
```

```sql
SELECT DISTINCT Bandas.nombre
FROM Bandas, Estudiantes_UC, Toco_en
WHERE Bandas.vocalista = Estudiantes_UC.nombre
AND Banda.nombre = Toco_en.nombre_banda
AND Toco_en.nombre_festival = 'Lollapalooza'
```

```sql
SELECT Bandas.nombre
FROM Bandas, Estudiantes_UC
WHERE Bandas.vocalista = Estudiantes_UC.nombre
INTERSECT
SELECT Toco_en.nombre_banda
FROM Toco_en
WHERE Toco_en.nombre_festival = 'Lollapalooza'
```

Son todas equivalentes!

Redundancia en SQL

¿Cómo saber cuál usar?

- Dada la naturaleza declarativa de SQL, es muy difícil predecir cómo diferentes formas de hacer una consulta puede tener un mejor rendimiento al ser ejecutadas por el RDBMS.
- Una aplicación real necesita hacer muchas consultas junto con código procedural para realizar su tarea. Cada consulta implica conectarse con la base de datos, y las conexiones tienen un *overhead* de tiempo adicional.
- Por eso en la práctica generalmente optimizamos el número de consultas a hacer, más que cómo están escritas esas consultas.