BASES DE DATOS Y SISTEMAS DE INFORMACIÓN

Tema 2.1

Miquel Gómez Corral Ing. Informática ETSINF, UPV

TEMA 2.1

SQL(*lito* 🍊)

SINTAXIS SQL

Las **consultas** se hacen siempre con `SELECT`. Para **Insertar** `INSERT`, para **Borrar** `DELETE` y para **Actualiza** `UPDATE`.

El resultado de hacer una consulta `SELECT` es siempre un atabla. Puede tener valores duplicados, para quitarlos hay que poner `UNIQUE` después.

Consulta SELECT

Tiene **3 partes**, se coge la **tabla/s del FROM** y se le pasa a la cláusula WHERE, este, elimina de la/s tabla/s las filas en las que la condición no es cierta (las que evalúa a falso o a indefinido), y le pasa el resultado a la cláusula SELECT. Este, de **todas las filas que le han llegado** del WHERE, retornará solo las expresiones A y B, o sea los atributos A y B.

```
SELECT [atributo1, atri2…]  FROM tabla1, [tabla2, tabla3…]   WHERE condición.
```

SELECT: Son los atributos, si los quieres todos pones ' * '. Detrás de él se le pueden especificar cosas como:

- **COUNT(*)**: Se le pone * o un atributo para que **cuente cuantos elementos hay**. Si se le pone `COUNT(DISTINCT atributo)` nos dice cuántos valores distintos de ese atributo hay. ***Igual con los siguientes:***
- **AVG(atr), MAX(art), MIN(atr),SUM(atr)…**: **Devuelven un número** según la Cláusula.

Se pueden devolver varios campos con a misma consulta al igual que indicar (*atributo1 + atributo2 – atributo3…*)

```
SELECT 'Núm.de ciclistas =', COUNT (*) Cuenta, 'Edad media=', AVG (edad) FROM Ciclista
```

'NÚM.DECICLISTAS='	CUENTA	'EDADMEDIA='	AVG(EDAD)
Núm. de ciclistas =	100	Edad media=	29,89

y Todos estos operadores/contadores… NO CUENTAN LOS VALORES NULOS (ni los suman, resta…) IGNORAN

- **DISTINCT**: Se especifica después del SELECT y hace que la tabla devuelta **NO tenga tuplas repetidas.**

FROM: Indicas las **tablas de las que se va a extraer la información**. A cada tabla se le **puede poner un apodo** para poder abreviar y diferenciar: `…FROM Película p, Actor a, Actua Act`…. Al especificar varias tablas, EL **SGBD hace el producto cartesiano** de todas las tablas (*por eso pueden haber algunas repetidas o así*)

WHERE: **Indica la condición** que han de cumplir las tuplas devueltas. *Se pueden poner cosas como p.duración>=60. a.Nombre LIKE '%Antonio%' para decir que el nombre contenga "Antonio"…* y Más cosas que vienen luego uwu.

- **LIKE**: Comparar una tira de caracteres con un patrón.
- **BETWEEN**: Comprueba si un escalar está en un rango entre dos valores. `WHERE edad BETWEEN 20 AND 30`
- **IN**: Comprobar si un valor está dentro de un conjunto de valores. *Se los puedes dar o hacer una subconsulta que te devuelva una tabla y ver si dicho valor está en esa tabla.* `WHERE p.categoria IN ('1','2','3')`
- **IS NULL**: permite comprobar si el valor de un atributo es nulo. `WHERE p.director IS NULL`
- **NOT**: Se pone delante de otro operador para invertirlo.
- **EXIST**: Se usa en subconsulta. Valida una tupla si la subconsulta devuelve al menos una tupla.

ORDER BY: Se pone al final de toda la consulta y se especifica la columna/s por la/s que se quiere ordenar:

```
SELECT nompuerto, altura, categoria FROM Puerto ORDER BY altura, categoria;
```

SUBCONSULTAS

Una subconsulta es una consulta que se realiza dentro de otra para poder expresar una condición. *Se evalúa primero esta subconsulta y luego las condiciones donde se usa*. Se usan principalmente los predicados **`IN`**, **`EXISTS`** y las comparaciones **`{=,<> o !=, <, >, <=, >=}`**.

IN: Sirve para **comprobar si cierto valor está** entre los valores de una tabla. Para conseguir está tabla se usan subconsultas. `WHERE R.A IN SELECT (SELECT A FROM S)` *Devuelve las filas de R cuyo valor del atributo A esta en S*

EXISTS: Está consulta **devuelve verdadero o falso** si, la subconsulta que hemos hecho tiene alguna tupla o no. Si yo hago `WHERE EXIST (SELECT * FROM S WHERE…)` y la subconsulta devuelve una o más filas, da true, si no, da false.

- **`SELECT *`**: Las subconsultas de `EXIST` SIEMPRE se usan con SELECT ***.**

NOT: Se pueden negar las clausulas IN y EXIST Poniéndoles un NOT delante. Esto quiere cedir

- **`NOT IN`**: En un principio esto quiere decir que cierto valor NO aparece entre los devueltos por la subconsulta. Pero ante la posible aparición de valores NULOS que puede dar por culo, debemos poner una clausula: "`SELECT…FROM…`**`WHERE A IS NOT NULL`**". ¡¡O MEJOR USAR NOT EXIST QUE ES MÁS FÁCIL!!
- **`NOT EXIST`**: Es equivalente a hacer una subconsulta con "COUNT (*)" y después hacer "= 0". Si por ejemplo me piden un algo que no está en no se donde: `WHERE` **`NOT`** `EXISTS(`**`SELECT *`** `FROM… WHERE …)`

Uso de `EXISTS` para cuantificación universa

Son aquellas que buscan **información** que satisfaga una condición para todas las filas del conjunto. *Para hacerlo sería como resolver un "*$\forall x\ (G \rightarrow F)$*" pero en SQL no existen ningún operador de esos, por lo que hay que traducirlo.*

$$\forall x\ (G \rightarrow F)\ ==\ \neg\exists\ (F\ \wedge\ \neg G)$$

Esto, en **lenguaje natural** para un ejemplo sería traducir la expresión "*Obtener el nombre del ciclista que ha ganado todas las etapas de más de 200 km*" a "*Obtener el nombre del ciclista tal que no existe una etapa de más de 200 km. que él no haya ganado*". Se puede deducir descomponiendo la frase de esta manera*:*

$$X\ =\ etapas\ de\ más\ de\ 200\ km.\ F(X)\ =\ que\ la\ haya\ ganado\ el\ ciclista\ actual.$$

```
SELECT nombre
FROM Ciclista C
WHERE NOT EXISTS                        /*¬∃*/
      (SELECT * FROM Etapa E
       WHERE km > 200 AND               /*X=etapas de más de 200 km*/
          NOT (C.dorsal = E.dorsal));   /*¬F(X)≡no la ha ganado el ciclista actual*/
```

Algunas de estas consultas también se pueden **resolver contando y comparando cuantos hay de cada cosa** y que haya la misma cantidad: *Preguntas cuantas etapas de +200km hay, luego que cuantas etapas ha ganado cada ciclista y que sean de +200km, comparas ambos números y aquel que cumpla la condición es.*

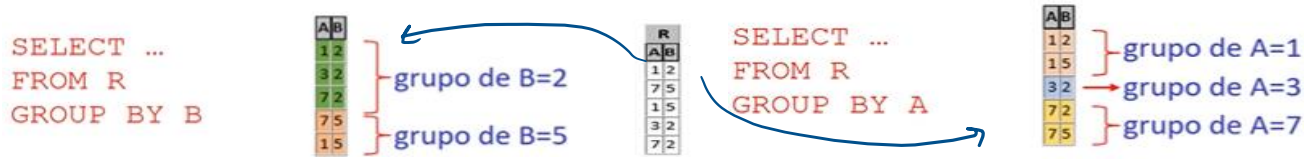
```
SELECT nombre
FROM Ciclista c
WHERE (SELECT COUNT(*) FROM Etapa e
            WHERE Distancia >= 200km)
      =
       (SELECT COUNT(*) FROM Etapa e1
            WHERE e1.dorsal = c.dorsal
                  AND e1.distancia >= 200km)
```

CONSULTAS AGRUPADAS

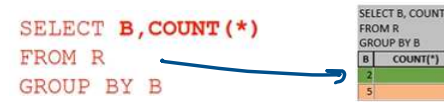
Una consulta **es agrupada** si a parte de las cláusulas `SELECT`, `FROM` y `WHERE` aparecen **`GROUP BY`** y **`HAVING`**. Al añadir GROUP BY y opcionalmente HAVING, las **cláusulas FROM Y WHERE no cambian comportamiento**, solo que ahora **WHERE** en vez de **pasar la tabla resultante** a SELECT ,*después de eliminar las tuplas que no cumplen la condición*, la **pasa a GROUP BY**.

GROUP BY Col1, Col2…

De **todas las filas que le llegan del WHERE**, **coge la/s columna/s** que le dices, y **agrupa las tuplas por valores iguales de esas columnas**. *Si en el select tenemos que devolver varias columnas (Ej DNI y nombre) TENGO QUE AGRUPAR POR ESAS MISMAS COLUMNAS, no puedo agrupar solo por DNI, sino también por Nombre*.



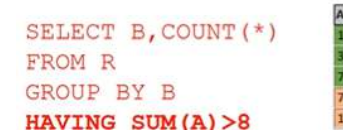
Count: Usada con un COUNT(*) en el SELECT devolvería la **cantidad de elementos de cada grupo** (*por B: 3 y 2, por A: 2, 1 y 2*). *La tabla resultante ahora no sería solo un valor, sino una tabla con varios valores*.



HAVING Condición

Esta **condición se evalúa a cada grupo resultante de GROUP BY** y **si no se cumple lo elimina**. *Es diferente del WHERE xq este evaluaba todas las filas una por una. El Having lo hace grupo por grupo.*

El ejemplo de la derecha lo que haría sería sumar todos los As de cada grupo. Del grupo con B = 2, SUM(A) = 11, por lo que el grupo entero se queda. Pero en B = 5, SUM(A) = 8, por lo que eliminaría ese grupo de la tabla resultado.



Condiciones: Para pedir cosas como "que al menos una fila tenga más de x como valor", o menos, o "que todas cumplan algo" se puede **usar lógica con MAX() y MIN()**. Ejemplos:

- Si te digo, **al menos uno tenga más de 30 años**, si compruebo **MAX(edad) > 30**, sé que al menos uno es +30.
- Si te digo que **todos tengan más de 30**, pues que el **MIN(edad) > 30** sé que todos tienen +30.

Subconsultas: Cuando **te pidan el grupo/s** que por ejemplo tenga/n el **mayor número de filas**, tienes que hacer: HAVING Count(*) = (num de filas del grupo con más filas), pero para conseguir ese num de filas, tienes que hacer una subconsulta: *Se anidan funciones, de todos los grupos, dame cuantos elementos hay, pues de todos, el que más tenga*.

```
HAVING COUNT(*) = (
       SELECT MAX(COUNT(*))
       FROM X
       GROUP BY X.clave)
```

ESTAS CONSULTAS SIEMPRE SE RESUELVEN IGUAL

CUIDADO SI PUEDE SER NULLO EL ELEMENTOS POR EL QUE ESTÁS BUCANDO

SELECT

Ahora se le pueden pedir expresiones que tomen un solo valor por grupo. *Antes no podias juntar el que te devuelva una columna con una COUNT() por ejemplo, pero ahora sí, ya que el count te devuelve otra columna de la tabla.*

Además, tener en cuenta que **si yo quiero devolver una columna/s / atributo/s**. SÍ o SÍ TENGO QUE AGRUPAR POR ESAS COLUMNA/S. Sino da error. *Hay que hacerlo aun que suene redundante.*

DISTINCT: Se tiene que usar solo cuando se concatenan tablas muchos a muchos. Cuando se usan tablas normales, que tienen claves primarias propias todo okey, *PERO SI SE USAN VARIAS TABLAS CON CLAVES AJENAS COMO CP Sí*.

CONSULTAS CONJUNTISTAS

Son las que trabajan con operadores de Conjuntos del Álgebra lineal: **`UNION`**, **`INTERSECT`** y **`MINUS`**. Estos operadores funciones entre consultas (NO subconsultas), ej: `consulta SELECT UNION consulta SELECT`.

Ambas consultas deben devolver **tablas con esquemas compatibles**: *Mismo num atributos, nombre de estos, tipo…*

Valores repetidos: **Todos los operadores eliminan las filas repetidas** menos si se usa "ALL": `UNION ALL`.

Nulos: El UNION puede dar por buenos valores nulos, por lo que habría que usar `IS NOT NULL`.

Ejemplo: Obtener dni de los profesores que son directores de algún departamento y tienen docencia.

Aun que quizás la SGBD pueda ejecutarlo, se pone "AS dni" detrás de director para que tengan el mismo nombre.

```
SELECT director AS dni
FROM Departamento
INTERSECT
SELECT dni
FROM Docencia
```

CONSULTAS CON CONCATENACIÓN EN EL FROM

Consultas Internas (O INNER JOIN)

Antes para **concatenar 2 tablas por un elementos** lo que hacíamos era `SELECT * FROM A, B WHERE A.x = B.x`. Ahora se puede usar una clausula dentro del FROM (FROM A NATURAL B) que hace lo mismo. *Lo que pasa es que no la vamos a usar xq hace falta saber como está declarada la base de datos y justa todos los atributos posibles*. Usamos:

```
FROM A JOIN B ON condition -> FROM A JOIN B ON A.x = B.x
```

Sería como hacer **T ⊗x B**. Concatena las filas A y B por el atributo X. Los valores de X que sean iguales en A y en B.

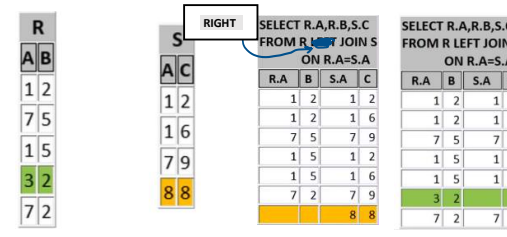
Consultas Externas (O OUTER JOIN)

Las consultas de antes nos concatenan cosas del palo, *todos los valores de X en A que también estén en B*, y **los que estén en uno PERO NO en otro los quita**. Pues si **queremos esos atributos**, tenemos que usar `LEFT, RIGHT o FULL`

Si por ejemplo usamos LEFT "`FROM R LEFT JOIN S ON R.a = S.A`" Lo que haría sería **concatenar como antes**, los valores de "a" iguales en S y R, pero encima, **los que de R NO tengan valor en S**, los va a poner llenando los valores de S como nulos. AL revés si es con RIGHT. "FULL" Haría lo mismo pero con los dos a la vez. (RIGHT Y LEFT)

Es como un "Cuidao no pierdas ninguna fila de este tabla crack

*Esto es útil si queremos que nos **devuelva TODAS las tuplas** de la tablas con las que trabajamos (Aun que no se concatene con nada)*



INSTRUCCIÓN INSERT

Se usa para añadir filas a las tablas. Por ejemplo si quiero añadir a la tabla ciclistas una nueva tupla sería

```
INSERT INTO Ciclista VALUES (101, 'Joan Peris', 27, 'Kelme');
```

Si se **desconoce el valor de algún atributo**, la instrucción debe incluir entre paréntesis el nombre de los atributos a los que se les va a asignar valor, y en el **mismo orden** en que van a introducir dichos valores.

```
INSERT INTO Ciclista (dorsal, nomeq, nombre) VALUES (101, 'Kelme', 'Joan Peris');
```

También se pueden insertar muchas filas de una vez si se obtienen a través de una consulta SELECT

```
INSERT INTO Cicli_win SELECT * FROM Cicli WHERE dorsal IN (SELECT dorsal FROM etapa)
```

INSTRUCCIÓN DELETE

Permite eliminar una o más filas completas de una tabla. En su forma más simple eliminaría todas las filas de la tabla.

```
DELETE FROM Tabla -> DELETE FROM Ciclista
```

Esto dejaría la tabla vacía. Para hacerlo hay que incluir en la instrucción una cláusula WHERE exactamente igual que en el SELECT.

```
DELETE FROM tabla [WHERE condición] -> DELETE FROM Ciclista WHERE edad < 20
```

INSTRUCCIÓN UPDATE

Esta instrucción permite modificar el valor de uno o más atributos en una o más filas de una tabla.

```
UPDATE Tabla SET asignación1, asignación2,…, asignaciónN [WHERE condición]
```

Incrementar en 1 la edad de los ciclistas `UPDATE Ciclista SET edad = edad + 1;`. Y asignación en el valor que sele le da a cada columna a modificar. Se puede hacer de varias filas a la vez. Tiene la forma:

```
columna = {DEFAULT | NULL | expresión_escalar}
```

La parte del WHERE es exactamente igual que la del SELECT y el DELETE:

```
UPDATE Equipo SET director = NULL     WHERE nomeq = 'Banesto';
```

INSTRUCCIONES DE CONTROL DE TRANSACCIONES

Se puede usar lo de "TRANSACCIÓN" para que un conjunto de instrucciones se ejecute sin que se evalue el estado de la BDD. De estas instrucciones o se ejecutan TODAS bien o ninguna.

Para decirle al sistema que ha terminado una transacción y que se pueden guardar todo O si no lo queremos volver atrás podemos usar **COMMIT** (*confirmar*) o **ROLLBACK** (*Deshacer los cambios*).

Con el commit lo que dices es que aceptas como está la base de datos tras tus cambios, **pero puede ser que el SGBD no los acepte xq se incumpla alguna restricción de integridad**. *Si es íntegra la transacción se acepta y todo guarda*.