

# Unidad 3.3: BBDD relacionales

Álgebra relacional  
y SQL 2



# INDICE

- Operaciones de álgebra relacional extendida
  - Asignación
  - Funciones agregadas
  - Reunión externa
- SQL:
  - Subconsultas anidadadas
  - Vistas
  - Consultas complejas
  - Otras características SQL

Referencias: Silberschatz Capítulo 3

# Operación de asignación $\leftarrow$

- Operación de asignación ( $\leftarrow$ ) permite escribir una expresión de álgebra relacional mediante la asignación de partes de esa expresión a variables temporales.
- Ejemplo: escribir  $r \div s$  como

temp1  $\leftarrow \Pi_{R-S} (r)$

temp2  $\leftarrow \Pi_{R-S} ((\text{temp1} \times s) - \Pi_{R-S,S} (r))$

result = temp1 - temp2

- El resultado de la derecha de  $\leftarrow$  se asigna a la variable relación temporal de la izquierda que puede usarse en expresiones posteriores.

# Funciones agregadas y operaciones

○ **Función agregada** toma un conjunto de valores y devuelve uno sólo.

- **avg**: valor medio
- **min**: valor mínimo
- **max**: valor máximo
- **sum**: suma de valores
- **count**: número de valores

○ **Operación agregada** en el álgebra relacional

$$G_1, G_2, \dots, G_n \quad \mathcal{G} \quad F_1(A_1), F_2(A_2), \dots, F_n(A_n) \quad (E)$$

- E es una expresión
- $G_1, G_2, \dots, G_n$  atributos por los que se agrupa (puede ser vacío)
- $F_i$  funciones agregadas
- $A_i$  nombres de atributo.

# Funciones agregadas y operaciones

## Ejemplo

- Determinar la suma total de los sueldos de los empleados a tiempo parcial. trabajo-por-horas

$G_{\text{sum}(\text{sueldo})}(\text{trabajo-por-horas})$

nombre-empleado	nombre-sucursal	sueldo
González	Centro	1.500
Díaz	Centro	1.300
Jiménez	Centro	2.500
Catalán	Leganés	1.600
Cana	Leganés	1.500
Cascallar	Navacerrada	5.300
Fernández	Navacerrada	1.500
Ribera	Navacerrada	1.300

-Resultado: una tupla de valor 16.500

- Si quisiéramos hacer lo mismo, pero de cada sucursal:

$\text{nombre-sucursal } G_{\text{sum}(\text{sueldo})}(\text{trabajo-por-horas})$

-Agrupar por nombre-sucursal y aplica la función a cada grupo

nombre-empleado	nombre-sucursal	sueldo
González	Centro	1.500
Díaz	Centro	1.300
Jiménez	Centro	2.500
Catalán	Leganés	1.600
Cana	Leganés	1.500
Cascallar	Navacerrada	5.300
Fernández	Navacerrada	1.500
Ribera	Navacerrada	1.300

nombre_sucursal	sum(sueldo)
Centro	5.300
Leganés	3.100
Navacerrada	8.100

Obsérvese que el resultado no tiene nombre → usar as

# Funciones agregadas

- Resultado de agregación no tiene nombre
  - Se puede usar la operación de renombramiento

*nombre-sucursal*  $\mathcal{G}_{\text{sum}(\text{sueldo}) \text{ as suma-sueldo, max}(\text{sueldo}) \text{ as sueldo-máximo}}$  (*trabajo-por-horas*)

<i>nombre-sucursal</i>	<i>suma-sueldo</i>	<i>sueldo-máximo</i>
Centro	5.300	2.500
Leganés	3.100	1.600
Navacerrada	8.100	5.300

- Para borrar los valores duplicados antes de aplicar la función de agregación:
  - Añadir -distinct después de la operación.
  - Ejemplo: Hallar el n° de sucursales que aparecen en la relación trabajo-por -horas. El resultado sería una única tupla con valor 3.

$\mathcal{G}_{\text{count-distinct}(\text{nombre-sucursal})}$  (*trabajo-por-horas*)

trabajo-por-horas

<i>nombre-empleado</i>	<i>nombre-sucursal</i>	<i>sueldo</i>
González	Centro	1.500
Díaz	Centro	1.300
Jiménez	Centro	2.500
Catalán	Leganés	1.600
Cana	Leganés	1.500
Cascallar	Navacerrada	5.300
Fernández	Navacerrada	1.500
Ribera	Navacerrada	1.300

# Valores nulos

- Es posible tener valores nulos en ciertas tuplas para ciertos atributos.
- Null significa **valor desconocido (unknown)** o que no existe
- El tratamiento de null depende del tipo de operación:
  - Expresión aritmética que incluya null es ***null***.
  - Operación de comparación que incluya null es ***unknown*** o desconocido
  - Operadores lógicos con *unknown*:
    - OR:        (unknown or true)                        = true,  
              (unknown or false)                      = unknown  
              (unknown or unknown)                  = unknown
    - AND: (true and unknown)                            = unknown  
         (false and unknown)                        = false  
         (unknown and unknown) = unknown
    - NOT: (not unknown)                                 = unknown



# Valores nulos en las operaciones de álgebra relacional

- Selección  $\sigma_p(E)$  :
  - si  $p$  es falso o *unknown*, no añade la tupla
- Join ( $r \bowtie s$ )
  - Similar a selección (al realizar una selección tras el producto cartesiano)
  - si valor nulo en el atributo común para  $r \bowtie s$ , no añade la tupla
  - Outer join o reunión externa igual, excepto con las tuplas que no aparecen en el resultado
    - Esas tuplas se añaden con nulos dependiendo si es: izquierda, derecha o total.
- Proyección (y proyección generalizada), unión, intersección, diferencia:
  - Los trata igual que cualquier otro valor
  - Al eliminar duplicados las tuplas *null* se eliminan (sin conocer su valor se está suponiendo que son iguales!!)
- Funciones de agregación  $\mathcal{G}$  :
  - Valores nulos en atributos agregados se borran antes de aplicar la agregación.



# Vistas

- Hasta ahora se ha operado en el nivel lógico (relaciones)
- En algunos casos no es deseable que todos los usuarios vean todo el modelo lógico de la base de datos
- **Vista:** relación que no forma parte del modelo conceptual pero que se hace visible al usuario como una relación virtual

**create view** *v* **as** <Expresión de consulta>,      donde  
<Expresión de consulta> es cualquier expresión de consulta legal del álgebra relacional.

**create view** *todos-los-clientes* **as**  
 $\Pi_{\text{nombre-sucursal}, \text{nombre-cliente}} (\text{impositor} \bowtie \text{cuenta})$   
 $\cup \Pi_{\text{nombre-sucursal}, \text{nombre-cliente}} (\text{prestatario} \bowtie \text{préstamo})$

# Vistas

- Una vez creada, se puede utilizar

$$\Pi_{\text{nombre-cliente}} (\sigma_{\text{nombre-sucursal} = \text{«Navacerrada»}} (\text{todos-los-clientes}))$$

- No se guarda el resultado, sino la definición de la vista
- Algunos SBGD permiten guardar el resultado  $\Rightarrow$  vistas materializadas/mantenimiento de vistas/instantánea/snapshot
- Actualizaciones sobre vistas  $\Rightarrow$  deben traducirse a las relaciones reales

-Ejemplo: Supongamos la siguiente inserción en la relación préstamo (número-préstamo, nombre-sucursal, importe):

**insert into préstamo-sucursal values ('P-37', 'Navacerrada')**

**create view préstamo-sucursal as**  
 $\Pi_{\text{nombre-sucursal, número-préstamo}} (\text{préstamo})$

- Faltaría el valor para el importe
- Hay 2 enfoques:
  - Se puede permitir, pero la tupla sería: (P-37, Navacerrada, null)
  - Rechazar la inserción y devolver error.

$$\text{préstamo-sucursal} \leftarrow \text{préstamo-sucursal} \cup \{(P-37, \text{«Navacerrada»})\}$$

*préstamo* (número préstamo, nombre\_sucursal, importe)

# Vistas

*préstamo* (número préstamo, nombre\_sucursal,  
importe)  
*prestatario* (nombre cliente, número préstamo)

- o Ejemplo: Otro problema de actualización en vistas:

**create view** *información-crédito* **as**  
 $\Pi_{\text{nombre-cliente, importe}}(\text{prestatario} \bowtie \text{préstamo})$

*información-crédito*  $\leftarrow$  *información-crédito*  
 $\cup \{(\text{«González»}, 1900)\}$

- Habría que insertar: (González, nulo) en prestatario y (nulo, nulo, 1900) en préstamo.
- No se consigue la tupla deseada (González, 1900)

- o Generalmente, no se permite actualización sobre vistas

# Vistas

*préstamo* (número préstamo, nombre\_sucursal, importe)  
*prestatario* (nombre cliente, número préstamo)  
*cuenta* (número cuenta, nombre\_sucursal, saldo)  
*impositor* (nombre cliente, número cuenta)

- Se puede definir vistas sobre otras vistas, pero sin recursividad:

**create view todos-los-clientes as**

$\Pi_{\text{nombre-sucursal, nombre-cliente}} (\text{impositor} \bowtie \text{cuenta})$   
 $\cup \Pi_{\text{nombre-sucursal, nombre-cliente}} (\text{prestatario} \bowtie \text{préstamo})$

**create view cliente-navacerrada as**

$\Pi_{\text{nombre-cliente}} (\sigma_{\text{nombre-sucursal} = \text{«Navacerrada»}} (\text{todos-los-clientes}))$

- La expansión de vistas es una manera de definir el significado de unas vistas definidas en términos de otras.
- Procedimiento expansión de vistas

**repeat**

Buscar todas las relaciones de vistas  $v_i$  de  $e_1$

Sustituir la relación de vistas  $v_i$  por la expresión que define  $v_i$

**until** no queden más relaciones de vistas en  $e_1$

$\sigma_{\text{nombre-cliente} = \text{«Martín»}} (\Pi_{\text{nombre-cliente}} (\sigma_{\text{nombre-sucursal} = \text{«Navacerrada»}} (\Pi_{\text{nombre-sucursal, nombre-cliente}} (\text{impositor} \bowtie \text{cuenta}) \cup \Pi_{\text{nombre-sucursal, nombre-cliente}} (\text{prestatario} \bowtie \text{préstamo}))))$

$\sigma_{\text{nombre-cliente} = \text{«Martín»}} (\text{cliente-navacerrada})$

$\sigma_{\text{nombre-cliente} = \text{«Martín»}} (\Pi_{\text{nombre-cliente}} (\sigma_{\text{nombre-sucursal} = \text{«Navacerrada»}} (\Pi_{\text{nombre-cliente}} (\sigma_{\text{nombre-sucursal} = \text{«Navacerrada»}} (\text{todos-los-clientes}))))$

# SQL

- Funciones de agregación
- Valores nulos
- Subconsultas anidadas
- Vistas
- Consultas complejas

# Funciones de agregación

- Toman una colección de valores de entrada y devuelven uno de salida:
  - Mínimo: `min`
  - Máximo: `max`
  - Cuenta: `count`
  - Suma: `sum`
  - Media: `avg`
- Notad que se utiliza después del `SELECT` (no en el `WHERE`). Para usarlo en `WHERE` tenemos que anidar `SELECT`s como veremos más adelante:
- Se pueden aplicar a un grupo de conjuntos de tuplas: `group by`

Valores de entrada numéricos

```
select avg (saldo)
from cuenta
where nombre-sucursal = 'Navacerrada'
```

```
- select nombre-sucursal
  from cuenta
  where saldo = (select max (saldo) from cuenta) -- También IN
```

-Ejemplo: Determinar el saldo medio de las cuentas de cada sucursal:

```
select nombre-sucursal, avg (saldo)
from cuenta
group by nombre-sucursal
```

# Funciones de agregación

- Si se desea eliminar los duplicados antes de efectuar la agregación  $\Rightarrow$  `distinct`
  - Ejemplo: Determinar el número de impositores de cada sucursal (cada impositor se debe contar una sola vez con independencia de que tenga varias cuentas)

```
select nombre_sucursal, count (distinct nombre_cliente)
from impositor, cuenta
where impositor.número_cuenta = cuenta.número_cuenta
group by nombre_sucursal
```

- Condiciones aplicadas a grupos en lugar de a tuplas  $\Rightarrow$  `having`
  - Ejemplo: sucursales en las que el saldo medio de la cuenta sea superior a 1200€

```
select nombre-sucursal, avg (saldo)
from cuenta
group by nombre-sucursal
having avg (saldo) > 1200
```

```
select count (*)
from cliente
```

- Contar tuplas de una relación (incluyendo nulos)  $\Rightarrow$  `count(*)`. No permite `distinct`

# Funciones de agregación

- Si la cláusula **having** aparece en **where**, SQL aplica primero el predicado del **where**

-Ejemplo: Determinar el saldo medio de cada cliente que vive en Madrid y tiene al menos 3 cuentas

- 1º. Se aplica where
- 2º se agrupa el resultado por nombre-cliente
- 3º se aplica having

```
select impositor.nombre-cliente, avg (saldo)
from impositor, cuenta, cliente
where impositor.número-cuenta
      = cuenta.número-cuenta and
      impositor.nombre-cliente
      = cliente.nombre-cliente and
      ciudad-cliente = 'Madrid'
group by impositor.nombre-cliente
having count (distinct impositor.número-cuenta) >= 3
```

- En SELECT y en HAVING solamente se pueden poner las columnas que aparecen en GROUP BY y funciones agregadas sobre el resto de columnas de la tabla.

-En el select anterior, no sería correcto: **select** impositor.nombre-cliente, saldo **from...**

- Tendríamos que añadir saldo a group by

- Las funciones agregadas no se pueden componer:

-Por ejemplo, max( avg (...)) no está permitido



# Valores nulos

- Modelo relacional permite usar valores nulos cuando no hay información.
  - Su uso con operadores aritméticos y de comparación causa algunos problemas
- Se permite usar la palabra NULL para buscar esa información (is null/is not null)

```
select número-préstamo  
from préstamo  
where importe is null
```

- Comportamiento SQL con null:
  - Expresión aritmética que contenga null  $\Rightarrow$  resultado null
  - Comparaciones que contengan null  $\Rightarrow$  desconocido unknown (desconocido)
    - Salvo con isnull y isnotnull
  - Con operadores lógicos:
    - AND
      - Cierto and desconocido = desconocido
      - Falso and desconocido = falso
      - Desconocido and desconocido = desconocido
    - OR
      - Cierto or desconocido = cierto
      - Falso or desconocido = desconocido
      - Desconocido or desconocido = desconocido
    - NOT: not desconocido = desconocido

# Valores nulos

- `select ...from R1,...,Rn where P`, si el predicado `P` se evalúa como falso o desconocido para alguna tupla de `R1x...xTn`, esa tupla no se añade al resultado
- SQL permite usar `is unknown/is not unknown` para comprobar si el resultado de una comparación es desconocido o no
- En operaciones de agregación SQL ignora los valores nulos al aplicar la operación salvo con la función `count(*)`
  - `count (*)` de una colección vacía (todos los valores de entrada nulos) = 0
  - Para el resto de funciones de agregación aplicadas sobre una colección vacía = null
- En SQL-99 se introdujo:
  - Tipo de dato boolean con valores: Cierto, falso y desconocido
  - Funciones de agregación sobre valores boolean: `some`, `every`

# Subconsultas anidadas

- Es una expresión `select-from-where` que se anida dentro de otra consulta (detrás del `from`, en vez de una tabla va otro `select`)
- Uso:
  - Comprobación de pertenencia a conjuntos: **(NOT) IN**
  - Comparación de conjuntos: **SOME, ALL,**
  - Cardinalidad de conjuntos: **(NOT) EXISTS, UNIQUE**

# Subconsultas anidadas

## Pertenencia a conjuntos: IN

- o Permite comprobar la pertenencia de las tuplas a una relación
  - Utiliza cálculo relacional para establecer la pertenencia a conjuntos
- o **(not) in** comprueba la pertenencia al conjunto resultado de una cláusula select
- o Ejemplo: Determinar todos los clientes que tienen tanto un préstamo como una cuenta en el banco ↔ *De entre los clientes que tienen cuenta, seleccionar los que tienen algún préstamo*

```
select distinct nombre_cliente  
from prestatario  
where nombre_cliente in (select nombre_cliente  
                        from impositor)
```

1. Determinamos los clientes que tienen una cuenta en el banco

2. Determinamos los clientes que tienen préstamos entre los que tienen cuenta

# Subconsultas anidadas

## Pertenencia a conjuntos: IN

- o Se puede usar con más de un atributo

-Ejemplo: Determinar todos los clientes que tienen tanto un préstamo como una cuenta en la sucursal de Navacerrada

```
select distinct nombre-cliente
from prestatario, préstamo
where prestatario.número-préstamo =
      préstamo.número-préstamo and
      nombre-sucursal = 'Navacerrada' and
      (nombre-sucursal, nombre-cliente) in
      (select nombre-sucursal, nombre-cliente
       from impositor, cuenta
       where impositor.número-cuenta
            = cuenta.número-cuenta)
```

De la relación de sucursales y clientes obtenida, seleccionar los que tienen préstamo en la sucursal de Navacerrada

Sucursales y nombres de clientes que tienen cuenta

- o Con **NOT IN** sería análogo

-Ejemplo: Todos los clientes que tienen concedido un préstamo, pero no tienen abierta cuenta

```
select distinct nombre-cliente
from prestatario
where nombre-cliente not in (select nombre-cliente
                             from impositor)
```

# Subconsultas anidadas

## Pertenencia a conjuntos: IN

- o También se puede usar sobre conjuntos enumerados
  - Ejemplo: Todos los clientes que tienen concedido un préstamo y cuyos nombres no son ni Santos ni Gómez cuenta

```
select distinct nombre-cliente  
from prestatario  
where nombre-cliente not in ('Santos', 'Gómez')
```

# Subconsultas anidadas

## Comparación de conjuntos: SOME

o Se utiliza SOME (antiguamente ANY) y se puede usar con: <, <=, >, >=, <>

o Ejemplo: Determinar el nombre de todas las sucursales que poseen activos mayores que, al menos, una sucursal de Barcelona

```
select distinct T.nombre-sucursal  
from sucursal as T, sucursal as S  
where T.activo > S.activo and  
      S.ciudad-sucursal = 'Barcelona'
```

- Usando SOME:

```
select nombre-sucursal  
from sucursal  
where activo > some (select activo  
                    from sucursal  
                    where ciudad-sucursal  
                        = 'Barcelona')
```

# Subconsultas anidadas

## Comparación de conjuntos: ALL

o Para comparar con todas las tuplas: all

o Se puede usar con: <, <=, >, >=, <>

- Ejemplo: Determinar el nombre de todas las sucursales que tienen activos superiores al de todas las sucursales de Barcelona

```
select nombre-sucursal
from sucursal
where activo > all (select activo
                    from sucursal
                    where ciudad-sucursal
                      = 'Barcelona')
```

- Ejemplo: Determinar la sucursal que tiene el saldo medio máximo

```
select nombre-sucursal
from cuenta
group by nombre-sucursal
having avg (saldo) >= all (select avg (saldo)
                          from cuenta
                          group by nombre-sucursal)
```

Obtenemos  
saldos medios  
de cada  
sucursal

Sucursales con saldo  
medio superior o igual  
que todos los saldos  
medios



# Subconsultas anidadas

## Comprobación de relaciones vacías: EXISTS

- Podemos comprobar si las subconsultas tienen alguna tupla en su resultado.
  - Si se devuelven tuplas o no  $\Rightarrow$  exists (true si no es vacía)
- Ejemplo: Determinar todos los clientes que tienen tanto una cuenta abierta en el banco como un préstamo concedido

```
select nombre_cliente
from prestatario
where exists (select *
               from impositor
               where impositor.nombre_cliente = prestatario.nombre_cliente)
```

# Subconsultas anidadas

Comprobación de relaciones vacías: NOT EXIST & EXCEPT

## o **Not exists:**

- Comprobar la inexistencia de tuplas en el resultado
- Se puede usar para simular la operación de continencia de conjuntos:
  - La relación A **contiene** a la relación B  $\leftrightarrow$  **not exists (B except A)**

# Subconsultas anidadas

## Comprobación de relaciones vacías: NOT EXIST & EXCEPT

### oEjemplo:

- Determinar todos los clientes que tienen una cuenta en todas las sucursales de Barcelona ↔
  - las sucursales en las que el cliente tiene cuenta (A) contienen a las sucursales de Barcelona (B)

**select distinct** *S.nombre-cliente*

**from** *impositor as S*

**where not exists** ((**select** *nombre-sucursal*  
**from** *sucursal*  
**where** *ciudad-sucursal*  
= 'Barcelona')

**except**

(**select** *R.nombre-sucursal*  
**from** *impositor as T, cuenta as R*  
**where** *T.número-cuenta*  
= *R.número-cuenta* **and**  
*S.nombre-cliente*  
= *T.nombre-cliente* ))

Conj. de sucursales que  
están en Barcelona

Conj. de sucursales en las  
que el cliente tiene cuenta

Busco los clientes que tienen  
cuenta en el banco para los que  
que not exists es true:  
las sucursales en las que tienen  
la cuenta están contenidas en el  
conjunto de sucursales que están  
en Barcelona

# Subconsultas anidadas

## Comprobación de tuplas duplicadas: UNIQUE

- o Unique  $\Rightarrow$  cierto si no devuelve tuplas duplicadas
- o Ejemplo: Determinar todos los clientes que tienen, a lo sumo, una cuenta en la sucursal de Navacerrada

```
select T.nombre-cliente
from impositor as T
where unique (select R.nombre-cliente
               from cuenta, impositor as R
               where T.nombre-cliente
                  = R.nombre-cliente and
                  R.número-cuenta
                  = cuenta.número-cuenta and
                  cuenta.nombre-sucursal
                  = 'Navacerrada')
```

- o Not unique  $\Rightarrow$  si hay tuplas duplicadas



# Vistas

- Es una consulta que se presenta como una tabla (virtual) a partir de un conjunto de tablas en una base de datos relacional
  - Se trata de relaciones que no forman parte del modelo lógico, pero que se hacen visibles a los usuarios para adaptar lo que ven a sus necesidades
- Se define con **crea create view v as <expresión de consulta>**
  - Ejemplo: Supongamos que en el departamento de Publicidad les interesa tener los clientes que tienen o bien cuenta abierta o bien préstamo concedido y los nombres de las sucursales en las que los tienen.

```
create view todos-los-clientes as
  (select nombre-sucursal, nombre-cliente
   from impositor, cuenta
   where impositor.número-cuenta
        = cuenta.número-cuenta)
union
  (select nombre-sucursal, nombre-cliente
   from prestatario, préstamo
   where prestatario.número-préstamo
        = préstamo.número-préstamo)
```



# Vistas

- Se pueden redefinir los nombres de los atributos
  - Ejemplo: Vista que muestra la suma del importe de todos los créditos de cada sucursal.

```
create view total-préstamos-sucursal  
            (nombre-sucursal, total-préstamos) as  
select nombre-sucursal, sum (importe)  
from préstamo  
group by nombre-sucursal
```

- Las vistas se utilizan después como cualquier otra relación:

```
select nombre-cliente  
from todos-los-clientes  
where nombre-sucursal = 'Navacerrada'
```

- Cuando se crea una vista el SGBD guarda la definición de la vista (no el resultado), por tanto, cuando se utiliza una vista en una query se vuelve a calcular siempre que se evalúa la consulta.

# Consultas complejas

## Relaciones derivadas

- SQL permite componer varios bloques SQL para expresar consultas complejas.
- Las relaciones derivadas permiten usar subconsultas dentro de la cláusula **from**
  - Para ello, proporcionamos un nombre a la relación resultado usando **as**
  - Ejemplo: Determinar el saldo medio de las cuentas de las sucursales en las que el saldo medio sea superior a 1200€

```
select nombre-sucursal, saldo-medio
from (select nombre-sucursal, avg (saldo)
      from cuenta
      group by nombre-sucursal)
as resultado (nombre-sucursal, saldo-medio)
where saldo-medio > 1200
```

- Ejemplo: Determinar el saldo total máximo de las sucursales

```
select max(saldo-total)
from (select nombre-sucursal, sum(saldo)
      from cuenta
      group by nombre-sucursal) as
      total-sucursal(nombre-sucursal,
                      saldo-total)
```

# Consultas complejas

## Claúsula with

- o Introducida en SQL-99. No lo incorporan todos los SGBD
- o Define una vista temporal que existe mientras exista la consulta
- o Ejemplo: Determinar todas las sucursales donde el saldo total de todas las cuentas es mayor que la media de los saldos totales de las cuentas de todas las sucursales.

```
with total_sucursales (nombre_sucursal, valor) as  
    select nombre_sucursal, sum(saldo)  
    from cuenta  
    group by nombre_sucursal  
with media_total_sucursales(valor) as  
    select avg(valor)  
    from total_sucursales  
select nombre_sucursal  
from total_sucursales, media_total_sucursales  
where total_sucursales.valor >= media_total_sucursales.valor
```

Vista temporal de las sucursales con sus totales de saldo

Vista temporal con el valor medio de los saldos totales de todas las sucursales

Seleccionamos sucursal en la que el saldo total de sus cuentas es mayor o igual a la media



# Modificación de la base de datos

## Actualización sobre vistas

- Pueden producir problemas si la vista no contiene la PK de la tabla original:
  - La vista contiene un subconjunto de atributos de otras relaciones por lo que, al realizar una operación sobre ella, pueden faltar datos que deben tratarse como error o null.

- Muchas bases de datos imponen:

Una modificación de una vista es válida sólo si la vista en cuestión se define en términos de la base de datos relacional real, esto es, del nivel lógico de la base de datos (y sin usar agregación).

- En general **insert**, **update** y **delete** están prohibidos en vistas

# Modificación de la base de datos - Transacciones

- Una **transacción** es una **unidad de trabajo** compuesta por diversas tareas, cuyo resultado final debe ser que **se ejecuten todas o ninguna** de ellas.
  - Ej: trasfiere 500€ de la cuenta 300 a la 301
  - O está hecha del todo, o no hecha en absoluto
- **Propiedades ACID** (*Atomicity, Consistency, Isolation y Durability*)
  - Una transacción, para cumplir con su propósito y protegernos de todos los problemas que hemos visto. Para ello, debe presentar las siguientes características:
    - **Atomicidad:** las operaciones que componen una transacción deben considerarse como una sola.
    - **Consistencia:** una operación nunca deberá dejar datos inconsistentes.
    - **Aislamiento:** los datos "sucios" deben estar aislados, y evitar que los usuarios utilicen información que aún no está confirmada o validada. (por ejemplo: ¿sigue siendo válido el saldo mientras realizo la operación?)
    - **Durabilidad:** una vez completada la transacción los datos actualizados ya serán permanentes y confirmados.

# Modificación de la base de datos - Transacciones

- En SQL una transacción comienza implícitamente cuando se ejecuta una instrucción implícitamente y finaliza cuando ejecutamos:
  - **COMMIT WORK**, finaliza la transacción anotando los cambios en la BD
  - **ROLLBACK WORK**, deshace los cambios en la BD
- En SQL-99.
  - Cada instrucción SQL es una transacción implícitamente.
  - Se especifica una transacción con:  
**BEGIN**  
    { sql\_statement | statement\_block }  
**END**



# Otras características SQL incorporado

- Utilización de SQL dentro de lenguaje de programación: C, java, fortran, pascal, etc. (lenguaje anfitrión)
- Se necesita un preprocesador especial antes de la compilación:
  - Java (SQLJ)
- Para identificar las peticiones al preprocesador de SQL incorporado se utiliza la instrucción **EXEC SQL**  

```
EXEC SQL <instrucción de SQL incorporado>  
END-EXEC
```
- La sintaxis exacta de las peticiones de SQL incorporado depende del lenguaje en el que se haya incorporado SQL.
  - Ejemplo: JAVA

```
# SQL { <instrucción de SQL incorporado> };
```

## Otras características: SQL incorporado

- Antes de ejecutar ninguna instrucción de SQL el programa debe conectarse con la base de datos:

**EXEC SQL connect to servidor user nombre-usuario END-EXEC**

- Hay algunas diferencias en cuanto a las instrucciones de SQL:

- Para formular una consulta relacional se emplea la instrucción

**declare** <nombre de cursor> **cursor**

- Para obtener el resultado de la query:

- **open** :ejecuta la consulta y guarda el resultado en una relación temporal

**EXEC SQL open** <nombre de cursor> **END-EXEC**

- **fetch**: guarda el resultado en variables (una variable por cada atributo de la relación resultado. Cada instrucción fetch devuelve una sola tupla (usar bucle)

**EXEC SQL fetch** <nombre de cursor> **into** :v1,..., :vn **END-EXEC**

- **close**: para indicar al sistema que borre la relación temporal

**EXEC SQL close** <nombre de cursor> **END-EXEC**



# Otras características SQL incorporado

- Ejemplo: Supóngase la variable del lenguaje anfitrión *importe* y que se desea determinar el nombre y ciudad de residencia de los clientes que tienen más de importe euros en alguna de sus cuentas.

EXEC SQL **Cursor de la consulta: sirve para identificarla en las instrucciones open y**  
**declare** **c** **cursor for**  
**select** *nombre-cliente, ciudad-cliente*  
**from** *impositor, cliente*  
**where** *impositor.nombre-cliente*  
*= cliente.nombre-cliente* **and**  
*cuenta.número-cuenta*  
*= impositor.número-cuenta* **and**  
*impositor.saldo > :importe*  
END-EXEC

EXEC SQL **open** *c* END-EXEC

EXEC SQL **fetch** *c* **into** *:cn, :cc* END-EXEC



# Otras características

## SQL dinámico

- Permite construir y ejecutar consultas en tiempo de ejecución

```
char * prog_sql = «update cuenta set saldo  
                = saldo * 1.05  
                where número-cuenta = ?»  
EXEC SQL prepare prog_din from :prog_sql;  
char cuenta[10] = «C-101»;  
EXEC SQL execute prog_din using :cuenta;
```

- Necesita extensiones del lenguaje y un preprocesador
- Mejor  $\Rightarrow$  normas de conexión. Interfaces para programas de aplicación.
  - Norma ODBC (Open DataBase Connectivity) en C (C++, C# y Visual Basic)
  - Norma JDBC en Java

# Otras características

## ODBC

Ejemplo de código C  
que usa la API de ODBC  
(4.5 SILBERSCHATZ)

```
int ODBCexample()
{
    RETCODE error;
    HENV ent; /* entorno */
    HDBC con; /* conexión a la base de datos */

    SQLAllocEnv(&ent);
    SQLAllocConnect(ent, &con);
    SQLConnect(con, «aura.bell-labs.com», SQL NTS, «avi», SQL NTS, «avipasswd», SQL NTS);
    {
        char nombresucursal[80];
        float saldo;
        int lenOut1, lenOut2;
        HSTMT stmt;

        char * consulta = «select nombre_sucursal, sum (saldo)
                           from cuenta
                           group by nombre_sucursal»;
        SQLAllocStmt(con, &stmt);
        error = SQLExecDirect(stmt, consulta, SQL NTS);
        if (error == SQL SUCCESS) {
            SQLBindCol(stmt, 1, SQL C CHAR, nombresucursal, 80, &lenOut1);
            SQLBindCol(stmt, 2, SQL C FLOAT, &saldo, 0 , &lenOut2);
            while (SQLFetch(stmt) >= SQL SUCCESS) {
                printf (« %s %g\n», nombresucursal, saldo);
            }
        }
        SQLFreeStmt(stmt, SQL DROP);
    }
    SQLDisconnect(con);
    SQLFreeConnect(con);
    SQLFreeEnv(ent);
}
```



# Otras características JDBC

```
public static void ejemploJDBC (String idbd, String idusuario, String contraseña)
{
    try
    {
        Class.forName («oracle.jdbc.driver.OracleDriver»);
        Connection con = DriverManager.getConnection
            («jdbc:oracle:thin:@aura.bell-labs.com:2000:bdbanco»,
            idusuario, contraseña);
        Statement stmt = con.createStatement();
        try {
            stmt.executeUpdate(
                «insert into cuenta values('C-9732', 'Navacerrada', 1200)»);
        } catch (SQLException sqle)
        {
            System.out.println («No se pudo insertar la tupla. » + sqle);
        }
        ResultSet rset = stmt.executeQuery
            («select nombre_sucursal, avg (saldo)
            from cuenta
            group by nombre_sucursal»);
        while (rset.next()) {
            System.out.println(rset.getString («nombre_sucursal») + « » +
                rset.getFloat(2));
        }
        stmt.close();
        con.close();
    }
    catch (SQLException sqle)
    {
        System.out.println («SQLException : » + sqle);
    }
}
```

# Otras características

- Bases de datos formadas por:
  - Catálogos
  - Esquemas
  - Objetos: relaciones, vistas
- Cada usuario tiene un catálogo asignado  
`catálogo5.esquema-banco.cuenta`
- Puede crear esquemas y borrarlos: `create (drop) scheme`
- Extensiones procedimentales (SQL-92)
  - Crear procedimientos (`begin, end`)
    - Nombre
    - Parámetros de entrada
    - Conjunto de instrucciones SQL
  - Procedimientos almacenados

# ¡Practiquemos!

<https://pgexercises.com/>



<https://www.db-book.com/university-lab-dir/sqljs.html>

