

# SQL

Base de Datos  
2do Cuatrimestre 2014  
Lic. Gerardo Rossel

# SQL – Introducción

- ◆ Structured Query Language
- ◆ Es el lenguaje más universalmente usado para bases de datos relacionales
- ◆ Lenguaje declarativo de alto nivel
- ◆ Desarrollado por IBM (1974-1977)
- ◆ Se convirtió en un standard definido por :
  - ANSI (American National Standards Institute) e
  - ISO (International Standards Organization)
- ◆ El standard que veremos es el SQL:1999 (Existen revisiones del 2003 y 2008)

# SQL - Introducción

Las Sentencias del SQL se dividen en:

- ◆ Sentencias DDL (Data Definition Language): Permiten crear/modificar/borrar estructuras de datos.
- ◆ Sentencias DML (Data Manipulation Language): para manipular datos
- ◆ También provee sentencias para:
  - Definir permisos (control de acceso de usuarios)
  - Manejo de transacciones
  - Otros

# DDL - Create table

```
CREATE TABLE empleados (  
    enombre      char(15) NOT NULL,  
    ecod         integer  NOT NULL,  
    efnac        date,  
    dcod         integer  
)
```

*Crea la tabla empleados con 4 columnas. La tabla no tendrá ninguna fila, hasta que no se ejecute un insert.*

# DDL - Create table

```
CREATE TABLE empleados (  
    enombre      char(15) NOT NULL,  
    ecod         integer  NOT NULL,  
    efnac       date,  
    dcod        integer,  
    CONSTRAINT PK_Empleados  
    PRIMARY KEY (ecod)  
)
```

*Es posible definir una  
clave primaria*

# DDL - Create table

```
CREATE TABLE empleados (  
    enombre char(15) NOT NULL,  
    ecod          integer NOT NULL,  
    efnac         date,  
    dcod          INTEGER,  
    CONSTRAINT PK_Empleados PRIMARY KEY(ecod),  
    CONSTRAINT FK_Deptos FOREIGN KEY (dcod) REFERENCES Deptos  
)
```

*Define la columna dcod como clave foránea apuntando a Deptos*

# DDL – Sentencia Drop table

- **DROP TABLE *table*;**

Ejemplo:

**DROP TABLE empleados;**

*Borra la tabla y todas sus filas*

# DDL – Alter table

- ◆ Permite:

- agregar columnas
- cambiar la definición de columnas
- agregar o borrar constraints

- ◆ **ALTER TABLE *table***

- ADD (*column datatype* [DEFAULT *expr*]);**

- Modificar Columna SQL Server

- ALTER TABLE *table* ALTER COLUMN *column datatype***

- Modificar Columna PostgreSQL

- ALTER TABLE *table* ALTER COLUMN *column* TYPE *datatype***

- ALTER TABLE *table* ALTER COLUMN *column* SET NOT NULL;**

- Modificar Columna Oracle

- ALTER TABLE *table* MODIFY *column datatype*;**



# DDL – Alter table

Ejemplo agregar clave primaria

```
ALTER TABLE tabla ADD CONSTRAINT pktbl PRIMARY KEY( idTabla)
```

Ejemplo agregar una clave foraneo

```
ALTER TABLE Hincha ADD CONSTRAINT FK_Hincha_Club FOREIGN  
KEY(ClubId) REFERENCES Club (ClubId)
```

Ejemplo agregar un check

```
ALTER TABLE club ADD CHECK (fechafundacion > ' 25/01/1900');  
ALTER TABLE club ADD CONSTRAINT fechafundacioncorrecta  
CHECK (fechafundacion > ' 25/01/1900');
```

# SQL - Instrucciones DML

Instrucciones DML: Permiten Manipular (leer y modificar) los datos almacenados en las tablas.

- ♦ INSERT: Crear nuevas filas en una tabla
- ♦ SELECT: Leer filas (o columnas) de tablas.
- ♦ UPDATE: Modificar filas existentes en una tabla
- ♦ DELETE: Borrar filas de una tabla.

# DML – SELECT

SELECT [ALL/DISTINCT] *select\_list*  
FROM *table* [*table alias*] [...]  
[WHERE *condition*]  
[GROUP BY *column\_list*]  
[HAVING *condition*]  
[ORDER BY *column\_name* [ASC/DESC] [...]]

# DML – SELECT

```
SELECT a1, ..., an  
FROM t1, ..., tn  
WHERE <cond>  
ORDER BY ai, aj
```

En algebra relacional:

$$\Pi_{a1 \dots an} (\sigma_{\langle \text{cond} \rangle} (t1 \mid X \mid \dots \mid X \mid tn))$$

# SELECT

```
SELECT ecod, enombre  
FROM empleados  
WHERE dcod=5;
```

En algebra relacional:

$$\Pi_{\text{ecod, enombre}} (\sigma_{\langle \text{dcod}=5 \rangle}(\text{empleados}))$$

*Obtener las columnas ecod, enombre de la tabla empleados de aquellas filas cuya columna dcod tiene el valor 5*

# SELECT (\*)

- ♦ Para acceder a todas las columnas → \*

SELECT \*

FROM empleados

WHERE dcod=40

*Obtener TODAS las columnas de la tabla empleados de aquellas filas cuya columna dcod tiene valor 40*

# DML - INSERT

## ♦ INSERT:

- Agrega filas en una tabla.
- **Unica** sentencia que provee SQL para agregar filas.
- Existen 2 Formas de ejecutar el insert

### 1) Usando la cláusula VALUES (agrega una sola fila por cada comando insert)

```
INSERT INTO table [(column [, column...])]  
VALUES (value [, value...]);
```

```
INSERT INTO empleados VALUES (1, 'Juan Perez', '04/04/98', 100)
```

```
INSERT INTO deptos (dcod, ddescr) VALUES (50, 'CONTABILIDAD')
```

```
INSERT INTO deptos VALUES (50, 'CONTABILIDAD')
```

```
INSERT INTO deptos DEFAULT VALUES;
```

### 2) Usando la cláusula SELECT (agrega un conjunto de filas mediante un solo insert)

# DML - INSERT

- También pueden insertarse un conjunto de filas

```
INSERT INTO table [(column [, column...])]  
SELECT...
```

```
INSERT INTO gerentes(gcod, gnombre, gsalarario)  
SELECT ecod, enombre, esalarario  
FROM empleados  
WHERE ecargo = 'GERENTE';
```

*La cantidad de columnas y tipos que devuelve el select debe coincidir con la cantidad de columnas de la Tabla.*



# DML - UPDATE

- ◆ Modifica filas existentes en una tabla

```
UPDATE table  
SET column = value [, column = value, ...]  
[WHERE condition];
```

- Ejemplos

```
UPDATE empleados  
SET dcod = 20  
WHERE ecod = 7782;
```

# DML - DELETE

- ◆ Borra filas existentes en una tabla

```
DELETE [FROM] table  
[WHERE condition];
```

- Ejemplos

```
DELETE FROM departamentos  
WHERE ddescr = 'FINANZAS';
```

*Delete sin where borra todas las filas,  
pero la tabla permanece creada (sin filas)*



# Manos a la obra!

- ♦ Ejercicios 1 al 4

# SELECT (Join)

SELECT nombre  
FROM empleados, deptos  
WHERE dcod = deptoid  
AND dnombre = 'Sistemas'

Tabla Deptos

deptoid	integer,
dnombre	char(30)
gerente	integer
pcod	integer

Condición de Junta

En algebra relacional:

$\Pi_{\text{nombre}} (\sigma_{\langle \text{dcod}=\text{deptoid AND dnombre}=\text{'Sistemas'} \rangle} (\text{empleados X deptos}))$

*Los empleados que trabajan en depto Sistemas*

# SELECT (join)

SELECT  
FROM  
WHERE  
AND  
AND

enombre, pnombre  
empleados, deptos, provincias  
dcod = deptoid  
pcod = provid  
dnombre = 'Sistemas'

Tabla Empleados

enombre	char(30),
ecod	integer,
Efnac	date,
dcod	integer

Tabla Deptos

deptoid	integer,
dnombre	char(30)
gerente	integer
pcod	integer

Tabla Provincias

provid	integer,
pnombre	char(30)
region	integer

# SELECT (join)

**Si los nombres de columnas se repiten, hay que anteponer el nombre de la tabla para evitar ambigüedades.**

```
SELECT      empleados.nombre, provincias.nombre
FROM        empleados, deptos, provincias
WHERE       empleados.deptoid = deptos.deptoid
AND         deptos.provid = provincias.provid
AND         deptos.nombre = 'Sistemas'
```

Tabla Empleados

nombre	char(30),
ecod	integer,
Efnac	date,
deptoid	integer

Tabla Deptos

deptoid	integer,
nombre	char(30)
gerente	integer
provid	integer

Tabla Provincias

provid	integer,
nombre	char(30)
region	integer

# SELECT (Alias)

Puedo usar alias de tablas para simplificar el SQL.

SELECT  
FROM  
WHERE  
AND  
AND

e.nombre, p.nombre  
empleados e, deptos d, provincias p  
e.deptoid = d.deptoid  
d.provid = p.provid  
d.nombre = 'Sistemas'

Los Alias se usan mayormente para simplificar la escritura del SELECT, sin embargo algunos tipos de subqueries requieren el uso de alias, ya que de otra manera no es posible escribirlos

Tabla Empleados

nombre	char(30),
ecod	integer,
Efnac	date,
deptoid	integer

Tabla Deptos

deptoid	integer,
nombre	char(30)
gerente	integer
provid	integer

Tabla Provincias

provid	integer,
nombre	char(30)
region	integer

# Funciones agregadas (Group by)

- ◆ Funciones: COUNT, SUM, MAX, MIN, AVG
- ◆ Operan sobre un grupo de filas y devuelven 1 resultado
- ◆ Los grupos de filas se definen con la clausula GROUP BY
- ◆ Si el select no tiene un GROUP BY el grupo está formado por todas las filas que recupera.



# Funciones agregadas (Group by)

**select min(salario), max(salario) from empleados**

Output pane		
Data Output Explain Messages His		
	min numeric	max numeric
1	1000.00	2200.00

**select dcod, min(salario), max(salario)  
from empleados group by dcod**

Data Output Explain Messages History			
	dcod integer	min numeric	max numeric
1	20	2200.00	2200.00
2	15	1000.00	2000.00
3	10	1000.00	1200.00

**select dcod, avg(salario) SAL\_PRM  
from empleados group by dcod**

Data Output Explain Messages		
	dcod integer	sal_prm numeric
1	20	2200.00000000
2	15	1500.00000000
3	10	1100.00000000

**select ecod, nombre, dcod, salario  
from Empleados**

Output pane				
Data Output Explain Messages History				
	ecod integer	nombre character varying(50)	dcod integer	salario numeric
1	1	Juan	10	1000.00
2	2	Pedro	15	2000.00
3	4	Juana	20	2200.00
4	5	Cata	15	1000.00
5	3	Maria	10	1200.00

# SELECT (group by)

```
SELECT dcod, enombre, AVG(esalario)
FROM empleados
GROUP BY dcod;
```

♦ Es posible ?, Qué devolverá ?

Si en SELECT hay funciones de agregación, entonces solo puedo proyectar columnas que estén en el group by o constantes.

# SELECT (group by – having)

```
SELECT dcod, count(*) , AVG(esalario)
FROM empleados
GROUP BY dcod;
```

```
SELECT dcod, count(*) , AVG(esalario)
FROM empleados
GROUP BY dcod
HAVING count(*) > 10 -- cond/restric sobre el grupo
```

# SELECT (order by)

- ♦ Para ordenar las filas que retorna la consulta.
- ♦ El valor por default es ASC

```
SELECT ddescr, enombre, esalario  
FROM empleados e, departamentos d  
WHERE e.dcod = d.dcod  
ORDER BY esalario DESC, d.dcod ASC
```

# SELECT (like)

```
SELECT *  
FROM empleados  
WHERE enombre LIKE '%H%';
```

◆ Otras opciones:

```
WHERE enombre LIKE '__H_';  
WHERE enombre LIKE '__H%';
```

# SELECT (distinct)

- ◆ SQL no elimina automáticamente las tuplas duplicadas. Para hacerlo se usa DISTINCT

```
SELECT DISTINCT dcod
```

```
FROM empleados
```

# SELECT RESUMEN

SELECT	<i>/* columnas/expresiones a ser retornadas */</i>
FROM	<i>/* relaciones entre tablas */</i>
[WHERE	<i>/* condic sobre la filas a ser retornadas */ ]</i>
[GROUP BY	<i>/* atributos de agrupamiento */ ]</i>
[HAVING	<i>/*cond sobre los grupos */ ]</i>
[ORDER BY	<i>/*orden en que se retornan las filas*/ ]</i>



# Labo

- ♦ Ejercicios hasta el 6.3



# Select Anidados

La clausula **WHERE** puede contener un **Select anidado** !

```
SELECT S.sname
FROM Sailors S
WHERE S.sid IN (SELECT R.sid
                FROM Reserves R
                WHERE R.bid=103)
```

Reserves

sid	bid	day
1	101	9/12
2	103	9/13
1	105	9/13

Sailors

sname
Bilbo

sid	sname	rating	age
1	Frodo	7	22
2	Bilbo	2	39
3	Sam	8	27

Primero obtiene el conjunto de los marinos que alquilaron el bote #103...(Inner query)

...y luego para cada fila del outer query verifica si cumple la clausula IN

sid
2

Buscar los nombres de los Marinos que alquilaron el bote #103:

# Consultas anidadas

```
SELECT      select_list
FROM        table
WHERE       expr operator
            (SELECT      select_list
             FROM        table);
```

- Usar operadores “single-row” para subqueries que retornan una fila (=, >, <, <>, >=, <=)
- Usar operadores “multiple-rows” para subqueries que retornan 0,1 o varias filas (IN, ANY, ALL)

# Consultas anidadas

```
SELECT enombre, esalario
FROM empleados
WHERE esalario = (SELECT MIN(esalario)
                  FROM empleados);
```

```
SELECT dcod, MIN(esalario)
FROM empleados
GROUP BY dcod
HAVING MIN(esalario) > (SELECT AVG(esalario)
                        FROM empleados);
```

*Es responsabilidad de quien escribe el query asegurar que el subquery devolverá una sola fila. Si el subquery devuelve 0 o + de 1 fila, da error al ejecutarlo*

# Consultas anidadas

OPERADOR	SIGNIFICADO
IN	Retorna TRUE si está incluido en los valores retornados por el subquery
ANY	Retorna TRUE si la comparación es TRUE para al menos un valor retornado por el subquery
ALL	Retorna TRUE si la comparación es TRUE para todos los valores retornados por el subquery
EXISTS	Retorna TRUE si el subquery devuelve al menos una fila. FALSE si devuelve 0 filas

# Consultas anidadas

```
SELECT  enombre, esalario
FROM    empleados
WHERE   dcod IN
        (SELECT dcod
         FROM  departamentos
         WHERE ddescr LIKE '%FINAN%') ;
```

```
SELECT dcod, ddescr
FROM  departamentos d
WHERE NOT EXISTS (SELECT *
                  FROM empleados e
                  WHERE  d.dcod = e.dcod) ;
```

# Consultas anidadas

```
SELECT  dcod
FROM    empleados
GROUP BY dcod
HAVING  AVG(esalario) >= ALL
        (SELECT AVG(esalario)
         FROM    empleados
         GROUP BY dcod );
```

# Consultas anidadas

```
UPDATE  empleados
      SET  (cargo, dcod) = (SELECT cargo, dcod
                           FROM  empleados
                           WHERE ecod = 7499)
      WHERE ecod = 7698;
```

```
DELETE FROM empleados
WHERE  dcod =
      (SELECT  dcod
       FROM    departamentos
       WHERE   ddescr = 'VENTAS');
```

# Consultas anidadas

```
SELECT esalario
FROM (SELECT esalario, egeren, dcod
      FROM empleados
      WHERE egeren IS NOT NULL)
WHERE dcod = 7698;
```

**ATENCION:** Se debe evitar si es posible.  
En general en el parcial se pide no usar select en el from, pero hay ejercicios en la práctica que es la única forma de resolverlos.



# SELECT (UNION)

- ♦ El operador UNION retorna las filas pertenecientes a ambas consultas eliminando las duplicadas

```
SELECT enombre, ecargo
FROM empleados
UNION
SELECT enombre, efuncion
FROM emp_hist;
```

# SELECT (UNION ALL)

- ◆ El operador UNION ALL retorna las filas pertenecientes a ambas consultas incluídas las duplicadas

```
SELECT enombre, ecargo
FROM empleados
UNION ALL
SELECT enombre, efuncion
FROM emp_hist;
```

# SELECT (INTERSECT)

- ♦ El operador INTERSECT retorna las filas comunes a ambas consultas

```
SELECT enombre, ecargo
FROM empleados
INTERSECT
SELECT enombre, efuncion
FROM emp_hist;
```

# SELECT (MINUS / EXCEPT )

- ♦ El operador MINUS/ EXCEPT retorna las filas de la primera consulta que no están presentes en la segunda

```
SELECT nombre, ecargo
FROM empleados
EXCEPT [all]
SELECT nombre, efuncion
FROM emp_hist;
```

# Mas consultas anidadas

- Empleados que ganan más que el promedio de salarios de su departamento

```
SELECT enombre, esalario, dcod
FROM empleados e1
WHERE esalario > (SELECT AVG(esalario)
                  FROM empleados e2
                  WHERE e1.dcod = e2.dcod);
```

*Es un subquery Correlacionado, ya que en el subquery, se hace referencia a la tabla del query externo. Por cada fila candidata del query externo, se ejecuta el subquery para verificar si la fila pertenece al resultado.*

# Mas consultas anidadas

- Empleados que tienen algun empleado a cargo

```
SELECT enombre
FROM empleados e1
WHERE EXISTS (SELECT *
              FROM empleados e2
              WHERE e1.ecod = e2.egeren) ;
```

# Mas consultas anidadas

- El menor salario por departamento de aquellos con más de 7 empleados.

```
SELECT dcod, MIN(esalario)
FROM empleados e1
GROUP BY dcod
HAVING COUNT(*) > 7
```

# Mas consultas anidadas

- Actualizar el salario de los empleados de los departamentos 1020 y 1040, sumandole el ultimo premio asignado

```
UPDATE empleados e
SET e.esalario = (SELECT e.esalario + p1.premio
                  FROM   premios p1
                  WHERE  p1.ecod = e.ecod AND
                        p1.fecha_premio =
                        (SELECT MAX(p2.fecha_premio)
                         FROM   premios p2
                         WHERE  e.ecod=p2.ecod)
)
WHERE dcod IN ('1020', '1040');
```





# Labo



- ♦ Ejercicios hasta 6.7

# División en SQL

**Sailors**

sid	sname	rating	age
1	Frodo	7	22
2	Bilbo	2	39
3	Sam	8	27

**Boats**

bid	bname	color
101	Nina	red
103	Pinta	blue

**Reserves**

sid	bid	day
1	103	9/12
2	103	9/13
3	103	9/14
3	101	9/12
1	103	9/13

# División en SQL (con Not Exists)

Obtener los marinos que alquilaron TODOS los botes

```
SELECT  S.sname  
FROM    Sailors S  
WHERE NOT EXISTS  
    (SELECT  B.bid  
     FROM    Boats B  
     WHERE NOT EXISTS  
         (SELECT  R.bid  
          FROM    Reserves R  
          WHERE   R.bid=B.bid  
                  AND R.sid=S.sid))
```

*} Obtener los marinos S tales que...*

*} No existe ningún bote B ...*

*} Sin una reserva a nombre del marino S*

# Division

```
SELECT  S.sname
FROM    Sailors S
WHERE NOT EXISTS
      (SELECT  B.bid
       FROM    Boats B
       WHERE NOT EXISTS
            (SELECT  R.bid
             FROM    Reserves R
             WHERE   R.bid= 103
                     AND R.sid= 3  )
```

Sailors

sid	sname	rating	age
1	Frodo	7	22
2	Bilbo	2	39
3	Sam	8	27

Boats

bid	bname	color
101	Nina	red
103	Pinta	blue

Reserves

	sid	bid	day
R	1	103	9/12
R	2	103	9/13
R	3	103	9/14
R	3	101	9/12
R	1	103	9/13

# Null Values

- ◆ En algunos casos no se dispone de un valor para asignar a una columna
  - *Por ejemplo: fecha de emisión del registro*
  - SQL provee un valor especial para estos casos: NULL

# Null Values

**Las columnas que no tienen ningún valor asignado contienen valor NULL**

## Ejemplo

create table T1 ( col1 integer, col2 integer, col3 integer)

insert into T1(col1, col3) values (9,9) —→ **El valor de col2 es NULL**

insert into T1(col1, col2, col3) values (8,8,8)

db2 => select \* from t1

C1	C2	C3
9	-	9
8	8	8

2 registro(s) seleccionados.

db2=>select \* from T1  
where C2 IS NULL

C1	C2	C3
9	-	9

1 registro(s) seleccionados.

db2 => select \* from T1  
where C2 IS NOT NULL

C1	C2	C3
8	8	8

1 registro(s) seleccionados.

# Null Values

- ◆ La presencia de *null* genera algunas complicaciones
  - Operador especial para controlar si un valor es nulo (IS NULL o IS NOT NULL).
  - Función ISNULL(...)
  - “edad > 21” - true o false cuando edad es *null*? Qué pasa con el AND, OR y NOT ?
  - Surge la necesidad de una “3-valued logic” (true, false and *unknown*).
  - Hay que ser cuidadoso con la clausula WHERE.
  - En SQL el WHERE elimina toda fila que NO evalua a TRUE en el WHERE (O sea condiciones que evaluan a False o Unknown no califican.)

# Null Values – 3 Valued Logic

(null > 0)

unknown

(null + 1)

null

(null = 0)

unknown

Not	
T	F
F	T
unknown	unknown

AND	T	F	unknown
T	T	F	unknown
F	F	F	F
unknown	unknown	F	unknown

OR	T	F	unknown
T	T	T	T
F	T	F	unknown
unknown	T	unknown	unknown



# Ejemplo con NULL Values

## Obtener los empleados que no tienen gente a cargo

```
SELECT E1.nombre FROM empleados E1
WHERE E1.legajo NOT IN
      (SELECT E2.legGer FROM empleados E2);
```

No rows selected.

Hay un empleado que tiene NULL en MGR, el Subquery contiene un valor nulo, y por lo tanto para todo valor de legajo, el NOT IN dará FALSO

```
SELECT E1.nombre
FROM empleados E1
WHERE NOT EXISTS (SELECT E2.* FROM empleados E2
                  WHERE E2.legmgr = E1.legajo);
```

### NOMBRE

María Lopez  
Pepín Gonzalez

Aquí no entran en juego los valores NULOS, ya que por cada fila se buscan los empleados a cargo, y si el conjunto es vacío, esa fila forma parte del resultado.

```
create table empleados (legajo int, nombre char(20), legGer int);
insert into empleados values (1,'Juan (el dueño)', null);
insert into empleados values (2,'Pedro Perez', 1);
insert into empleados values (3,'Maria Lopez', 2);
insert into empleados values (4,'Pepin Gonzalez', 2);
```

# SELECT (Inner Join)

```
SELECT nombre, ddescr  
FROM empleados e INNER JOIN departamento d  
ON d.dcod=e.dcod
```

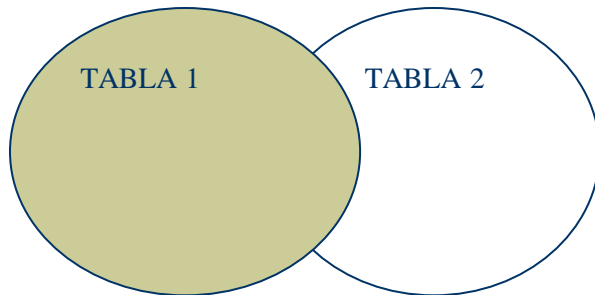
```
SELECT nombre, ddescr  
FROM empleados , departamento d  
WHERE d.dcod=e.dcod
```

# Junta Externa (outer join)

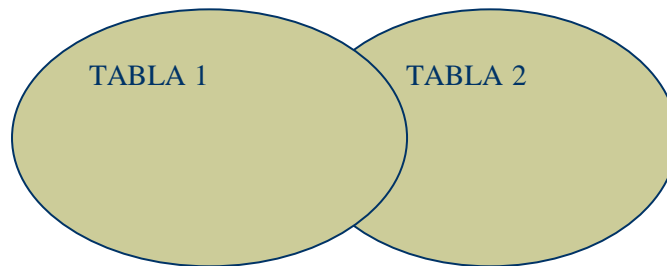
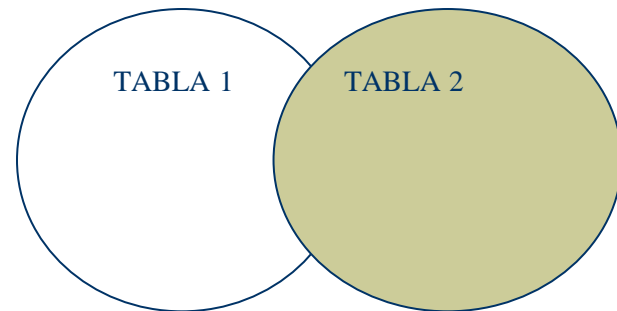
- ◆ Puede ser derecha o izquierda o total. Se incluyen las tuplas que cumplan la condición de join más aquellas en las cuales el valor de la columna (el de la izquierda o derecha según el caso) que participa en el join, tiene valor nulo.
  - *FULL OUTER JOIN*
  - *LEFT OUTER JOIN*
  - *RIGHT OUTER JOIN*

# JUNTA EXTERNA

*LEFT OUTER JOIN*



*RIGHT OUTER JOIN*



*FULL OUTER JOIN*

# SELECT (outer join)

- ♦ Todos los nombres de empleados y si corresponde la descripción del departamento (aunque no tengan departamento )

```
SELECT nombre, ddescr
```

```
FROM empleados e LEFT OUTER JOIN departamento d ON d.dcod=e.dcod
```

- Todos los departamentos, aunque no tengan empleados

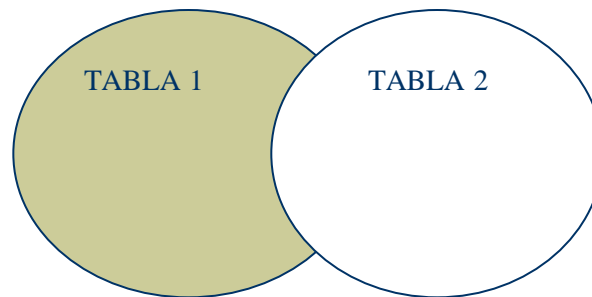
```
SELECT nombre, ddescr
```

```
FROM empleados e RIGHT OUTER JOIN departamento d ON  
d.dcod=e.dcod
```

# EJEMPLO

```
SELECT t1.*  
FROM Tabla1 t1  
WHERE t1.ID NOT IN (SELECT t2.ID FROM Tabla2 t2)
```

```
SELECT t1.*, t2.*  
FROM Tabla1 t1 LEFT JOIN Tabla2 t2 ON t1.ID = t2.ID  
WHERE t2.ID IS NULL
```



# Mas Ejemplos

- ♦ Todos los elementos de la tabla 1 que **no** estén relacionados (mediante una columna, digamos columna1) con la tabla 2 y todos los de la tabla 2 que **no** estén relacionados con la tabla 1

```
SELECT * FROM tabla1 t1 FULL OUTER JOIN tabla2 t2  
ON t1.Columna1 = t2.Columna1  
WHERE t1.Columna1 IS NULL or t2.Columna1 IS NULL
```

# Ejemplo

Padres(padre,hijo)

Jose, jose

Jose, pedro

Horacio, juan

Fernando, paco

Fernando, gustavo

```
SELECT p1.hijo, p2.hijo
FROM Padres p1, Padres p2
WHERE p1.padre = p2.padre
AND p1.hijo <> p2.hijo
```

¿Cómo hago para evitar repetición?





# Labo

- ♦ Ejercicios hasta el 6.9

# RECURSIVIDAD

- ◆ Common Table Expressions: Incorporado en el estándar de 1999.
- ◆ Implementado por:
  - IBM DB2 UDB 8 (Dec. 2002)
  - Microsoft SQL Server 2005 (Oct. 2005)
  - Sybase SQL Anywhere 11 (Aug. 2008)
  - Firebird 2.1 (Sep. 2008)
  - PostgreSQL 8.4 (Jul. 2009)
  - Oracle 11g release 2 (Sep. 2009)
  - HSQLDB 2.3 (Jul. 2013)
  - Teradata y H2 (no se conoce cuando comenzaron Teradata al menos en el 2009)

# WITH Queries (Common Table Expressions)

create table empleados4 (legajo int, nombre char(20), legGer int);

- ♦ La consulta se divide en caso base y caso recursivo.
- ♦ Ejecutar caso base para crear la primera invocación o conjunto de resultados base ( $T_0$ ).
- ♦ Ejecutar caso recursivo con  $T_i$  como entrada y  $T_{i+1}$  como salida.
- ♦ Repetir el paso anterior hasta que se devuelva un conjunto vacío.
- ♦ Se devuelve el conjunto de resultados. Es una instrucción UNION ALL de  $T_0$  a  $T_n$ .

```
insert into empleados values (1,'Juan (el dueño)', null);
insert into empleados values (2,'Pedro Perez', 1);
insert into empleados values (3,'Maria Lopez', 2);
insert into empleados values (4,'Pepin Gonzalez', 2);
```

	legajo integer	nombre character(20)	legger integer	nivel integer
1	1	Juan (el dueño)		0
2	2	Pedro Perez	1	1
3	3	Maria Lopez	2	2
4	4	Pepin Gonzalez	2	2

# WITH Queries (Common Table Expressions)

```
create table empleados4 (legajo int, nombre char(20), legGer int);
```

```
WITH RECURSIVE Jerarquia (legajo , nombre, legGer ,  
nivel)
```

```
AS
```

```
(
```

```
--Caso base
```

```
SELECT legajo , nombre, legGer , 0 as Nivel
```

```
FROM Empleados
```

```
WHERE legGer IS NULL
```

```
UNION ALL
```

```
--Caso recursivo
```

```
SELECT e.legajo , e.Nombre, e.legGer, j.Nivel+1 as Nivel
```

```
FROM Empleados e INNER JOIN Jerarquia j
```

```
ON e.legGer = j.legajo
```

```
)
```

```
select * from Jerarquia
```

```
insert into empleados values (1,'Juan (el dueño)', null);  
insert into empleados values (2,'Pedro Perez', 1);  
insert into empleados values (3,'Maria Lopez', 2);  
insert into empleados values (4,'Pepin Gonzalez', 2);
```

	legajo integer	nombre character(20)	legger integer	nivel integer
1	1	Juan (el dueño)		0
2	2	Pedro Perez	1	1
3	3	Maria Lopez	2	2
4	4	Pepin Gonzalez	2	2

# VISTAS

- ♦ Son relaciones pero de las cuales solo almacenamos su definición, no su conjunto de filas.

- ♦ **CREATE VIEW *view* [(*column* [, *column*...])]  
AS SELECT .....;**

**DROP VIEW *view*;**

```
CREATE VIEW depto_totales (ecod, totalsal, maxsal)
AS SELECT ecod, sum(salario), max(salario)
FROM empleados
GROUP BY ecod;
```

# Stored Procedures

- ♦ Es una porción de código, que se almacena en el catálogo de la base de datos y se puede invocar mediante una sentencia SQL.
- ♦ Se ejecuta en el servidor de base de datos. Reduce el número de idas y vueltas entre aplicaciones y el servidor de base de datos.
- ♦ Encapsulan reglas de negocio fuertemente relacionadas con los datos de la BD y sin interacción con el usuario
- ♦ No es obligatorio que estén escritos en SQL (Java, PL/SQL, PL/pgSQL, Transact SQL)
- ♦ Cada RDBMS tiene su propio lenguaje de Stored Procedure, los cuales incluyen sentencias de control, manejo de variables, etc.
- ♦ Los stored Procedures, tienen un nombre, reciben parametros y pueden devolver resultados
- ♦ Permite reutilizar código entre diversas aplicaciones

# Stored Procedures Contras

- ♦ Velocidad de desarrollo mas lenta.
- ♦ Requiere habilidades especiales
- ♦ Es dificultoso manejar versiones y es mas complejo depurar
- ♦ Puede no ser portable entre diverentes sistemas de bases de datos.

# PostgreSQL

## Stored Procedures

- ◆ PostgreSQL

- Defecto: SQL, PL/SQL y C
- Otros como extensiones: Perl, Python,



# PostgreSQL (Stored Procedures)

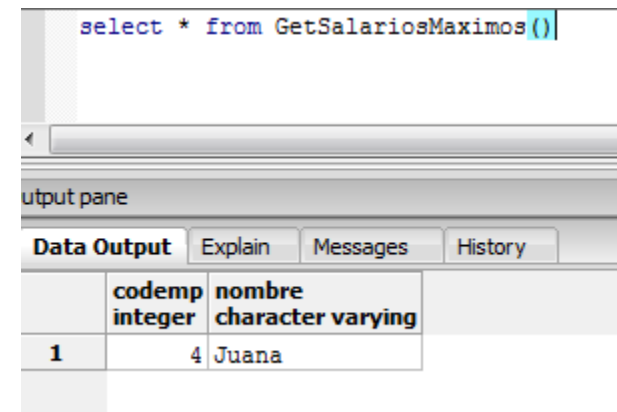
Ejemplo:

```
CREATE OR REPLACE FUNCTION function_name(p1 integer, p2 integer)
RETURNS integer AS
$BODY$BEGIN
    RETURN p1 + p2;
END$BODY$
LANGUAGE plpgsql ;
```

Uso:

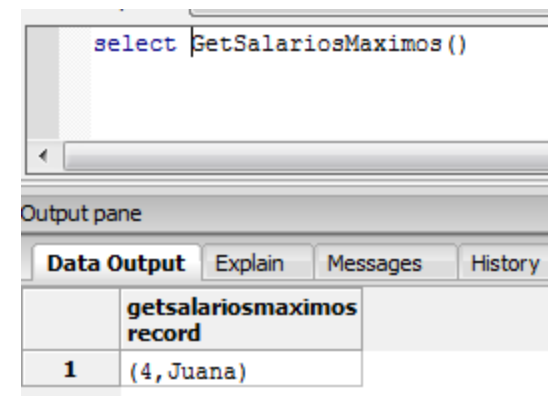
```
select function_name(2,5);
```

```
CREATE OR REPLACE FUNCTION GetSalariosMaximos()
RETURNS table (codemp integer, nombre character varying(50)) AS
$BODY$
    SELECT ecod, nombre FROM empleados
    WHERE salario = ( SELECT max(salario) FROM empleados);
$BODY$
LANGUAGE sql;
```



The screenshot shows a PostgreSQL query editor with the query `select * from GetSalariosMaximos()` entered. Below the query, the 'Output pane' is visible, showing the results of the query. The output is a table with two columns: 'codemp' (integer) and 'nombre' (character varying). The first row of data shows '1' for 'codemp' and 'Juana' for 'nombre'.

	codemp integer	nombre character varying
1	4	Juana



The screenshot shows a PostgreSQL query editor with the query `select GetSalariosMaximos()` entered. Below the query, the 'Output pane' is visible, showing the results of the query. The output is a table with two columns: 'getsalariosmaximos' (record) and 'nombre' (character varying). The first row of data shows '1' for 'getsalariosmaximos' and '(4, Juana)' for 'nombre'.

	getsalariosmaximos record
1	(4, Juana)

# PostgreSQL (Stored Procedures)

Ejemplo:

```
CREATE FUNCTION get_available_flightid(date)
  RETURNS SETOF integer AS
  $BODY$ BEGIN
    RETURN QUERY SELECT flightid FROM flight
      WHERE flightdate >= $1 AND flightdate < ($1 + 1);
    -- Since execution is not finished, we can check whether rows were returned
    -- and raise exception if not.
    IF NOT FOUND THEN
      RAISE EXCEPTION 'No flight at %.', $1;
    END IF;
    RETURN;
  END $BODY$
LANGUAGE plpgsql;
```

# Trigger

- ◆ Código almacenado en la DB que se ejecuta ante ciertos eventos.
  - Evento: activa el trigger
  - Acción: código que se ejecuta si se dispara el trigger

# Trigger (PostgreSQL)

-- Auditoría de cambios de salarios

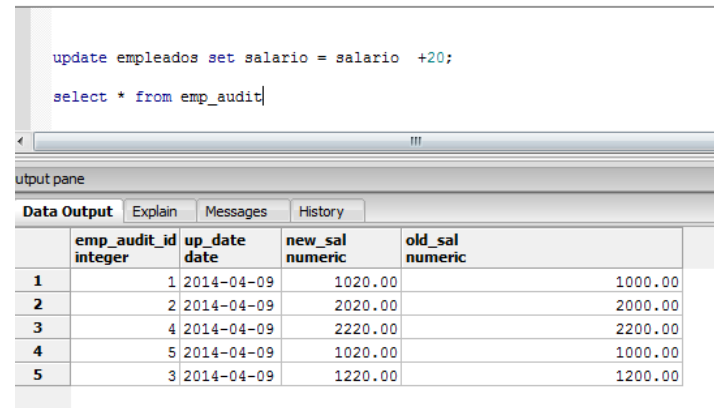
```
CREATE TABLE emp_audit ( emp_audit_id integer, up_date DATE,  
                           new_sal decimal, old_sal decimal);
```

-- Primero debo crear un stored procedure

```
CREATE OR REPLACE FUNCTION actualizar_auditoria() RETURNS trigger AS  
  $BODY$BEGIN  
    insert into emp_audit values(old.ecod, CURRENT_DATE, new.salario, old.salario);  
    return null  
  END$BODY$  
LANGUAGE plpgsql
```

-- Ahora creo el trigger

```
CREATE TRIGGER audit_sal AFTER UPDATE OF salario  
  ON empleados  
FOR EACH ROW  
  WHEN (OLD.salario IS DISTINCT FROM NEW.salario)  
  EXECUTE PROCEDURE actualizar_auditoria();
```



The screenshot shows a PostgreSQL database interface. At the top, there is a query editor with the following SQL code:

```
update empleados set salario = salario +20;  
  
select * from emp_audit;
```

Below the query editor, there is a tabbed interface with the following tabs: Data Output, Explain, Messages, and History. The "Data Output" tab is selected, and it displays the results of the query in a table format.

	emp_audit_id integer	up_date date	new_sal numeric	old_sal numeric
1	1	2014-04-09	1020.00	1000.00
2	2	2014-04-09	2020.00	2000.00
3	4	2014-04-09	2220.00	2200.00
4	5	2014-04-09	1020.00	1000.00
5	3	2014-04-09	1220.00	1200.00

# Trigger (PostgreSQL)

```
CREATE TRIGGER nombre { BEFORE | AFTER }  
  { INSERT | UPDATE | DELETE [ OR ... ] } ON tabla  
  [ FOR [ EACH ] { ROW | STATEMENT } ]  
  EXECUTE PROCEDURE function_name( argumentos )
```

# Trigger (MS SQL)

-- Auditoría de cambios de salarios

```
CREATE TABLE emp_audit ( emp_audit_id integer, up_date DATE,  
                           new_sal decimal, old_sal decimal);
```

--

```
CREATE TRIGGER audit_sal  ON empleados  AFTER UPDATE
```

```
AS
```

```
BEGIN
```

```
    SET NOCOUNT ON;
```

```
    IF UPDATE(salario) -- Solo si se actualiza salario
```

```
    BEGIN
```

```
        INSERT INTO emp_audit(emp_audit_id,up_date, new_sal, old_sal)
```

```
        SELECT i.ecod, getdate(),i.salario, d.salario
```

```
        FROM Inserted i INNER JOIN Deleted d ON i.ecod = d.ecod
```

```
    END
```

```
END
```

# INDICES

- ◆ Es una estructura de acceso físico a datos
- ◆ Son usados para acceder más rápidamente a filas de tablas.
- ◆ Son independientes lógicamente y físicamente de la tabla que indexan

```
CREATE INDEX indice  
ON table (column[, column]...);
```

```
DROP INDEX indice;
```

```
CREATE INDEX emp_enombre_i ON empleados (enombre);
```



# Labo

- ◆ Ejercicios hasta el final