

3

Lenguajes de acceso a bases de datos:



Javier Arroyo

Yolanda García

Luis Garmendia

Virginia Francisco Gilmartín

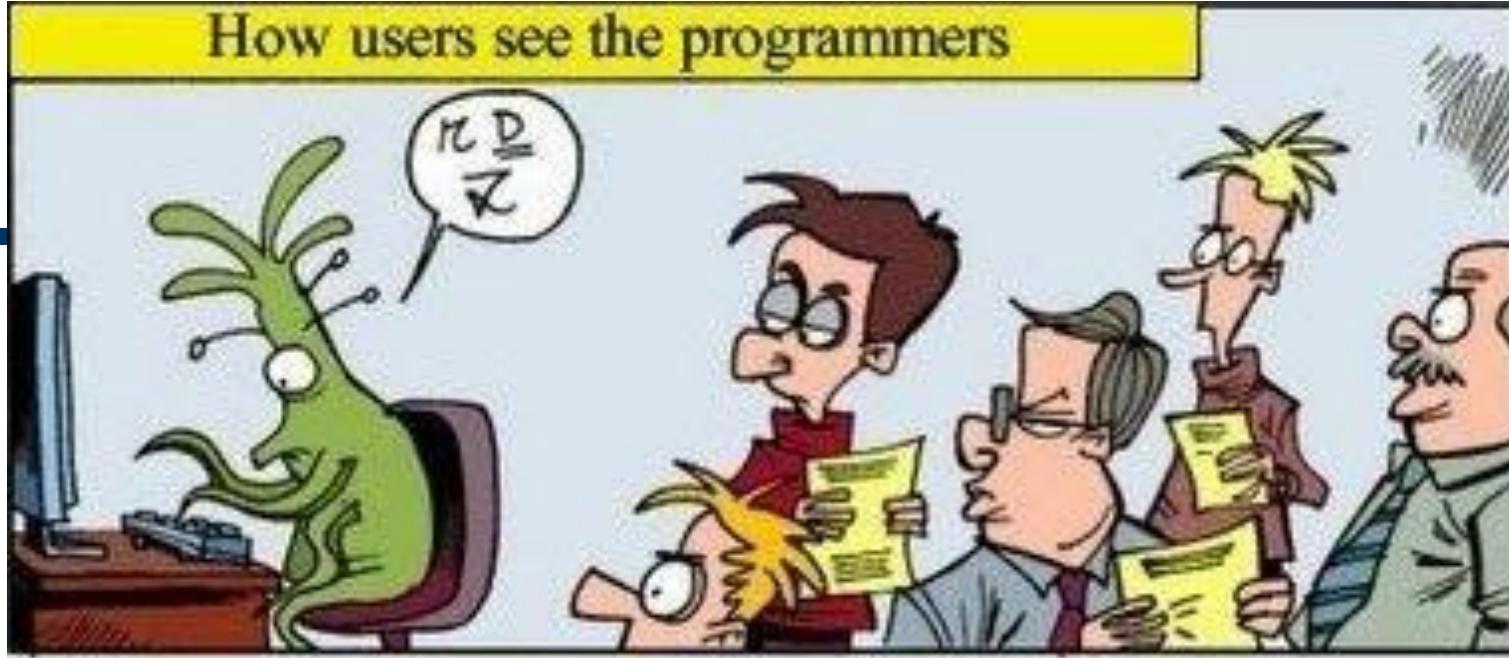
Iván Martínez

Fernando Sáenz

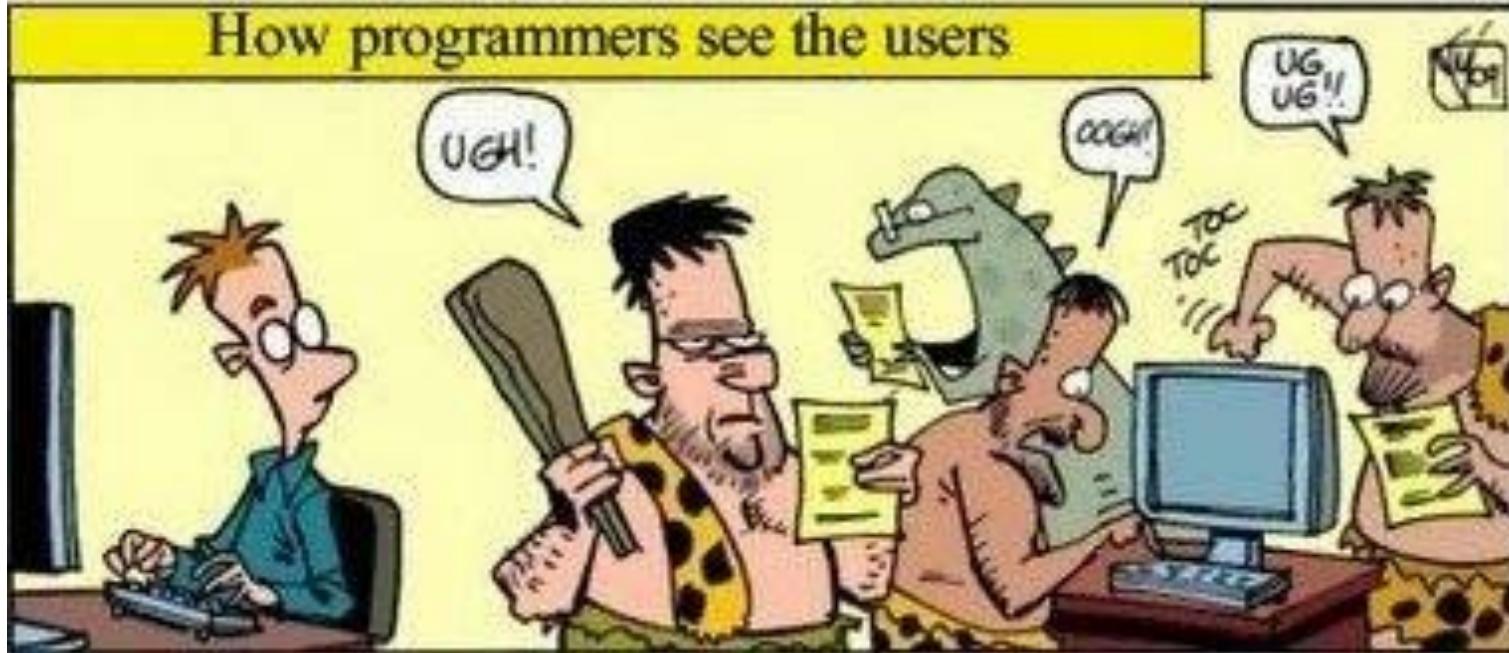
Facultad de Informática
Universidad Complutense



How users see the programmers



How programmers see the users

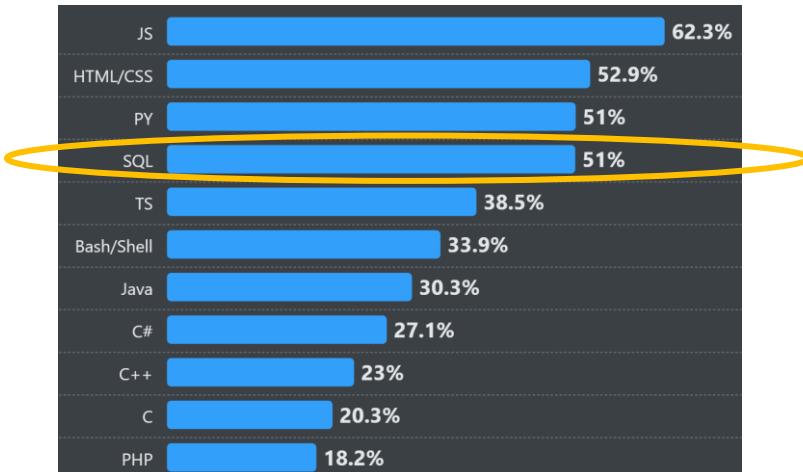


Introducción a SQL

- ◆ Structured Query Language (SQL)
- ◆ Lenguaje **declarativo** de acceso a bases de datos basado en el cálculo relacional y genera planes de consultas basados en álgebra relacional.
- ◆ Originalmente desarrollado en los 70 por IBM en su Research Laboratory de San José a partir del sencillo cálculo relacional de tuplas creado por Ted Codd.

Introducción a SQL

- ◆ Lenguaje estándar *de facto* en los SGBD comerciales
 - ¿Un lenguaje con 50 años a sus espaldas es el más usado en BD?



[Stack Overflow Developer Survey, 2024](#)

In 2023, the IEEE [noted](#) that SQL was the most popular language for developers to know when it came to getting a job

- No obstante: "*The best games are easy to learn but hard to master*" - Nolan Bushnell, fundador de Atari
 - Aforismo también conocido como "Ley de Bushnell" o "Ley de Nolan"



Introducción a SQL

- ◆ Propuesta inicial:
 - SEQUEL (Structured English QUERy Language), IBM 1976
- ◆ Estándares:
 - SQL-86 (ANSI SQL: Structured Query Language)
 - SQL-89 (SQL1)
 - SQL-92 (SQL2), gran revisión del estándar
 - **SQL:1999** (SQL3), Añade disparadores, algo de OO, ... ¡recursión!
 - SQL:2003. Añade XML, secuencias y columnas autonuméricas.
 - SQL:2006, 2008, 2011, 2016, 2019, 2023 (¡Tipo JSON!)...

Partes de SQL

- ◆ Componentes:

- Lenguaje de Definición de Datos (LDD / DDL)
 - Proporciona comandos para la creación, borrado y modificación de esquemas relacionales (tablas, vistas, dominios, ...)
- Lenguaje de Manipulación de Datos (LMD / DML)
 - Basado en el álgebra relacional y el cálculo relacional de tuplas permite realizar consultas y adicionalmente insertar, borrar y modificar tuplas
- Lenguaje de Control de Datos (LCD / DCL)
 - Definición de usuarios y privilegios

SQL - LDD

- ◆CREATE
- ◆ALTER
- ◆DROP

Creación de tablas

- ◆ La creación de tablas se lleva a cabo con la sentencia **CREATE TABLE**.

- ◆ **Ejemplo:** creación del siguiente esquema de BD.

CLIENTES (DNI, NOMBRE*, DIR*) **SUCURSALES (NSUC, CIUDAD*)**
CUENTAS (COD, DNI, NSUCURS, SALDO*)

- ◆ Se empieza por las tablas más independientes (las que no dependen de otras debido a restricciones de integridad referencial). En este ejemplo:

```
1 CREATE TABLE CLIENTES (
2   DNI      VARCHAR(9),
3   NOMBRE   VARCHAR(20),
4   DIR      VARCHAR(30),
5   PRIMARY KEY (DNI)
6 );
```

```
1 CREATE TABLE SUCURSALES (
2   NSUC      VARCHAR(4),
3   CIUDAD   VARCHAR(30),
4   PRIMARY KEY (NSUC)
5 );
```

Creación de tablas

- ◆ Como ves en el ejemplo, cada columna viene acompañada de su tipo.
 - **VARCHAR(*N*)**: Indica una cadena de hasta *N* caracteres.
 - Al final se pueden añadir restricciones de tabla, como la clave primaria. En el ejemplo: **PRIMARY KEY (DNI)** especifica que DNI es la clave primaria de la tabla CLIENTES.
 - La clave primaria nunca puede tener valor **NULL**.

```
1 CREATE TABLE CLIENTES (
2   DNI      VARCHAR(9),
3   NOMBRE   VARCHAR(20),
4   DIR      VARCHAR(30),
5   PRIMARY KEY (DNI)
6 );
```

```
1 CREATE TABLE SUCURSALES (
2   NSUC     VARCHAR(4),
3   CIUDAD   VARCHAR(30),
4   PRIMARY KEY (NSUC)
5 );
```

Creación de tablas (cont.)

CLIENTES (DNI, NOMBRE*, DIR*) SUCURSALES (NSUC, CIUDAD*)
CUENTAS (COD, DNI, NSUCURS, SALDO*)

- ◆ La última es la tabla CUENTAS, con las claves externas:

```
1 CREATE TABLE CUENTAS (
2   COD VARCHAR(4) NOT NULL,
3   DNI VARCHAR(9) NOT NULL,
4   NSUCURS VARCHAR(4) NOT NULL,
5   SALDO INT DEFAULT 0 CHECK(SALDO>=0),
6   PRIMARY KEY (COD, DNI, NSUCURS),
7   FOREIGN KEY (DNI) REFERENCES CLIENTES (DNI),
8   FOREIGN KEY (NSUCURS) REFERENCES SUCURSALES (NSUC));
```

Restricción de columna
Esta coma permite separar y distinguir las restricciones de tabla
Restricción de tabla

- ◆ La siguiente transparencia explica un poco este ejemplo.

Creación de tablas (cont.)

- ◆ **NOT NULL**: Restricción que impide valores nulos. Aplicarlo a una clave en ciertos sistemas es redundante y en otros se requiere (como en DB2).
- ◆ **DEFAULT**: Permite indicar un valor por defecto en caso de que no se proporcione uno al insertar una tupla.
- ◆ **CHECK(*C*)**: Condición *C* que debe cumplir toda tupla insertada.
- ◆ **FOREIGN KEY**: Establece las restricciones de integridad referencial como restricción de tabla. Por ejemplo, los valores del campo **DNI** de **CUENTAS** deben encontrar correspondencia en el campo **DNI** de **CLIENTES**:

FOREIGN KEY (DNI) REFERENCES CLIENTES (DNI)

También se admite como restricción de columna:

DNI CHAR(9) REFERENCES CLIENTES(DNI)

En ambos casos, si los atributos se denominan igual, se pueden omitir de la tabla referenciada.

- ◆ Las claves candidatas se podrían indicar con la cláusula **UNIQUE** (o con un índice sin duplicados en MS Access). En este ejemplo no hay.

Creación de tablas: tipos de datos

Tipos de datos de SQL estándar	
CHARACTER	
CHARACTER VARYING (o VARCHAR)	
CHARACTER LARGE OBJECT	
NCHAR	
NCHAR VARYING	
BINARY	
BINARY VARYING	
BINARY LARGE OBJECT	
NUMERIC	
DECIMAL	
SMALLINT	
INTEGER	
BIGINT	
FLOAT	
REAL	
DOUBLE PRECISION	
BOOLEAN	
DATE	
TIME	
TIMESTAMP	
INTERVAL	

45 páginas del estándar (SQL Foundation) dedicadas a los tipos de datos

Tipo de datos en DES	Significado
varchar string	Cadena de longitud ilimitada
char	Carácter (cadena de longitud 1)
char(N) varchar(N)	Cadena de longitud hasta N caracteres
integer int	Número entero
float real	Número en coma flotante
date	Fecha expresada como date(Year,Month,Day)
time	Hora expresada como time(Hour,Minute,Second)
datetime timestamp	Momento (timestamp) expresado como datetime(Year,Month,Day,Hour,Minute,Second)

Tipo de datos en MS Access	Longitud	Descripción
BINARY	1 byte	Tipo de datos binario.
BIT	1 byte	Valores Sí/No o True/False
BYTE	1 byte	Un valor entero entre 0 y 255.
COUNTER	4 bytes	Un número incrementado automáticamente (de tipo Long)
CURRENCY	8 bytes	Un entero escalable entre 922.337.203.685.477,5808 y 922.337.203.685.477,5807.
DATETIME	8 bytes	Un valor de fecha y hora entre los años 100 y 9999.
SINGLE	4 bytes	Un valor en coma flotante de precisión simple con un rango de -3.402823*10 ³⁸ a -1.401298*10 ⁻⁴⁵ para valores negativos, 1.401298*10 ⁻⁴⁵ a 3.402823*10 ³⁸ para valores positivos, y 0.
DOUBLE	8 bytes	Un valor en coma flotante de doble precisión con un rango de -1.79769313486232*10 ³⁰⁸ a -4.94065645841247*10 ⁻³²⁴ para valores negativos, 4.94065645841247*10 ⁻³²⁴ a 1.79769313486232*10 ³⁰⁸ para valores positivos, y 0.
SHORT	2 bytes	Un entero corto entre -32,768 y 32,767.
LONG	4 bytes	Un entero largo entre -2,147,483,648 y 2,147,483,647.
LONGTEXT	1 byte por carácter	De cero a un máximo de 1.2 gigabytes.
LONGBINARY	Según se necesite	De cero 1 gigabyte. Utilizado para objetos OLE.
TEXT	1 byte por carácter	De 0 a 255 caracteres.

Tipo de datos	Sinónimos
BINARY	VARBINARY
BIT	BOOLEAN LOGICAL LOGICAL1 YESNO
BYTE	INTEGER1
COUNTER	AUTOINCREMENT
CURRENCY	MONEY
DATETIME	DATE TIME TIMESTAMP
SINGLE	FLOAT4 IEEE SINGLE REAL
DOUBLE	FLOAT FLOAT8 IEEE DOUBLE NUMBER NUMERIC
SHORT	INTEGER2 SMALLINT
LONG	INT INTEGER INTEGER4
LONGBINARY	GENERAL OLEOBJECT
LONGTEXT	LONGCHAR MEMO NOTE
TEXT	ALPHANUMERIC CHAR CHARACTER STRING VARCHAR
VARIANT	VALUE

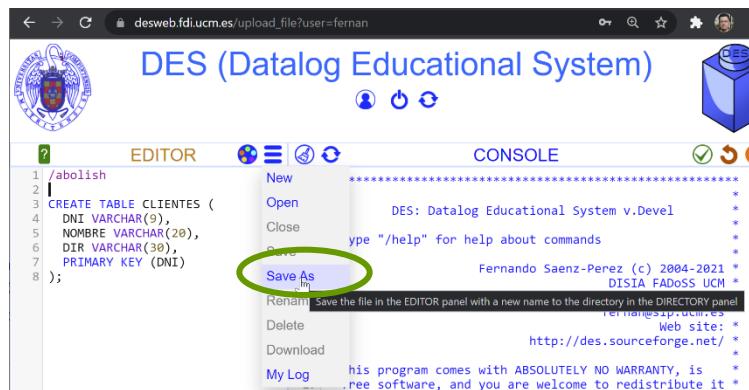
Creación de tablas (Ejercicio 01)

◆ Intenta crear estas tablas en DES:

- Accede a la dirección <https://desweb.fdi.ucm.es/> desde tu navegador web.
- Pulsa el botón Normal. Te pedirá tus credenciales de acceso, que son:
 - Usuario: `dg25_idcorreo`

Donde `idcorreo` es el identificador de tu usuario de correo electrónico. Por ejemplo, el mío sería "**fernan**" (porque mi correo es `fernan@ucm.es`); es decir, mi usuario para DES sería: `dg25_fernan`. Escribe el que corresponda en tu caso y ¡no uses el mío! 😊

- Contraseña: igual que el usuario. Lo puedes cambiar en el panel superior una vez hayas iniciado sesión.
- Si el navegador te pide Reenviar formulario, hay que pulsar Aceptar.
- Escribe en el panel **EDITOR** (arriba a la izquierda) la instrucción de creación de la primera tabla, precedida por el comando `/abolish`. Guarda el archivo con el nombre `ejercicio01.sql`:



The screenshot shows the DES (Datalog Educational System) interface. On the left, the **EDITOR** pane contains the following SQL code:

```
1 /abolish
2
3 CREATE TABLE CLIENTES (
4   DNI VARCHAR(9),
5   NOMBRE VARCHAR(20),
6   DIR VARCHAR(30),
7   PRIMARY KEY (DNI)
8 );
```

On the right, the **CONSOLE** pane shows the system prompt and some user information:

```
*****
DE: Datalog Educational System v.Devel
type "/help" for help about commands
Fernando Saenz-Perez (c) 2004-2021 *
DISIA FADoSS UCM *
```

A context menu is open over the code in the **EDITOR** pane, with the "Save As" option highlighted by a green circle. A tooltip for this option is visible, stating: "Save the file in the EDITOR panel with a new name to the directory in the DIRECTORY panel".

Creación de tablas (Ejercicio 01)

- ◆ Pulsa **Save As** para que se guarde este archivo. Te aparecerá el botón verde **Run** arriba a la izquierda. Púlsalo y comprueba que el panel **DATABASES** contiene la tabla que has creado.

DES (Datalog Educational System)

User: fernan

Run EDITOR Save Save As Close Colors Clear Manual Refresh CONSOLE Commit Rollback Restart

```
1 /abolish
2
3 CREATE TABLE CLIENTES (
4   DNI VARCHAR(9),
5   NOMBRE VARCHAR(20),
6   DIR VARCHAR(30),
7   PRIMARY KEY (DNI)
8 );
9
10
11
```

DES>

Info: Batch file processed.

DES>

Remove Selected DIRECTORY Upload DATABASES Refresh

	New Folder	
..		
DLLDebugger		Remove
fuzzy		Remove
hypothetical		Remove
ontology		Remove
persistence		Remove
SQLDebugger		Remove
ackermann.dl	Download	Remove
aggregates.dl	Download	Remove
aggregates.ra	Download	Remove
aggregates.sql	Download	Remove
alcanzabilidad.sql	Download	Remove

Databases Answer

- ..
- Databases
- \$des
 - Tables
 - CLIENTES(DNI,NOMBRE,DIR)
 - Columns
 - DNI:varchar(9)
 - NOMBRE:varchar(20)
 - DIR:varchar(30)
 - PK: [DNI]
 - Views
 - Constraints

Creación de tablas (Ejercicio 01)

- ◆ *Añade* al final del archivo (en el mismo panel EDITOR) la creación de la **segunda tabla** y pulsa Run de nuevo. Comprueba que la estructura de tablas en el panel DATABASES refleja el cambio.
- ◆ Repite lo mismo para la **tercera tabla** y comprueba que ha creado bien.
- ◆ Si hubiera algún error se mostraría en rojo en el panel CONSOLE.
- ◆ Abre el archivo Guía de uso.pdf que tienes en el panel DIRECTORY. Léelo completamente y familiarízate con DESweb.

```
1 CREATE TABLE CUENTAS (
2   COD VARCHAR(4) NOT NULL,
3   DNI VARCHAR(9) NOT NULL,
4   NSUCURS VARCHAR(4) NOT NULL,
5   SALDO INT DEFAULT 0 CHECK(SALDO>=0),
6   PRIMARY KEY (COD, DNI, NSUCURS),
7   FOREIGN KEY (DNI) REFERENCES CLIENTES (DNI),
8   FOREIGN KEY (NSUCURS) REFERENCES SUCURSALES (NSUC));
```

Modificación y eliminación de tablas

◆ Modificación de tablas: sentencia **ALTER TABLE**.

- Es posible añadir, modificar y eliminar campos. Ejemplos:

- Adición del campo PAIS a la tabla CLIENTES

ALTER TABLE CLIENTES ADD PAIS VARCHAR(10);

- Modificación del tipo del campo PAIS (no es SQL estándar).

ALTER TABLE CLIENTES ALTER PAIS SET TYPE VARCHAR(20);

También se permite redefinir completamente el campo, por ejemplo:

ALTER TABLE CLIENTES ALTER PAIS VARCHAR(20) NOT NULL;

- Eliminación del campo PAIS de la tabla CLIENTES

ALTER TABLE CLIENTES DROP PAIS;

- También es posible añadir nuevas restricciones a la tabla (claves externas, restricciones **CHECK**...):

- Evitar que el número de sucursal sea la cadena vacía:

ALTER TABLE SUCURSALES ADD CONSTRAINT CHECK(NSUC<>'');

◆ Eliminación de tablas: sentencia **DROP TABLE**.

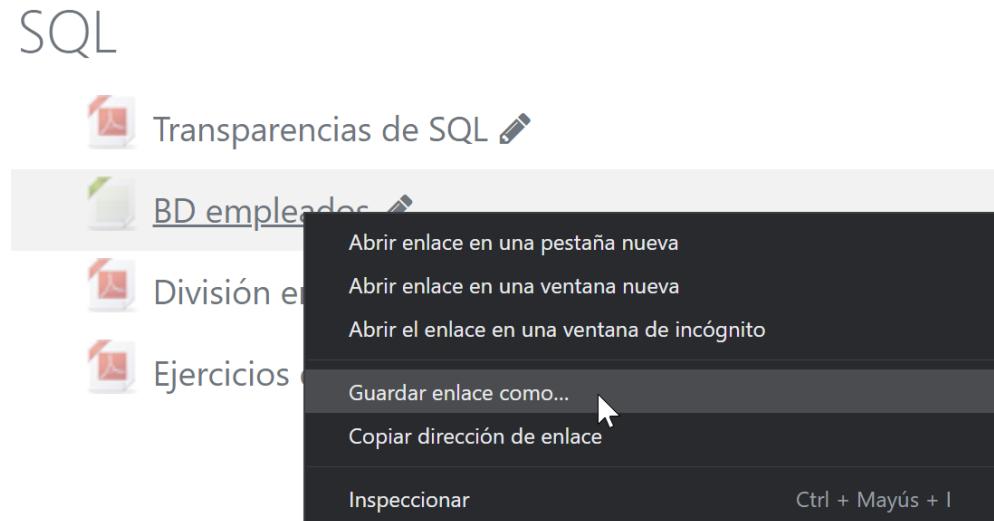
- **DROP TABLE CUENTAS;**

Modificación y eliminación de tablas (Ejercicio)

- ◆ Desde el inductor de comandos (**DES>**) que se encuentra en el panel **CONSOLE** (el de arriba a la derecha), intenta eliminar la tabla **CLIENTES** con la sentencia **DROP TABLE**. No olvides el punto y coma final (**;**) y pulsar Intro. ¿Qué ocurre con las restricciones de integridad referencial de **CUENTAS**? Examina la estructura de tablas en el panel **DATABASES**.
- ◆ Elimina todas las tablas con el mismo método. La base de datos debería quedar vacía.
- ◆ Como este ejercicio se hace desde el inductor de comandos, no es necesario guardar ningún archivo.

Creación de una base de datos (Ejercicio 02)

- ◆ Descarga el archivo Empleados.sql (basado en la BD 'sample' de IBM DB2) que se encuentra en el CV (en la pestaña "3. SQL", al lado de donde están estas transparencias).
- ◆ Si tu navegador solo te lo muestra pero no lo descarga, entonces usa el siguiente elemento del menú contextual:



- ◆ Sube el archivo con el icono **Upload** (o desde el menú de archivo) a DESweb. Ábrelo (pulsando sobre su nombre en el panel **DIRECTORY**) y ejecútalo (pulsando el icono **Run**). Se deberían crear las tablas e insertar los datos que vienen en las siguientes transparencias.

Ejemplo

```
1 CREATE TABLE EMPLOYEE
2   (EMPNO      CHARACTER(6) PRIMARY KEY
3    ,FIRSTNAME VARCHAR(12) NOT NULL
4    ,MIDINIT    CHARACTER(1)
5    ,LASTNAME   VARCHAR(15) NOT NULL
6    ,WORKDEPT   CHARACTER(3)
7    ,PHONEENO   CHARACTER(4)
8    ,HIREDATE   DATE
9    ,JOB        CHARACTER(8)
10   ,EDLEVEL    SMALLINT NOT NULL
11   ,SEX        CHARACTER(1)
12   ,BIRTHDATE  DATE
13   ,SALARY     DECIMAL(9,2)
14   ,BONUS      DECIMAL(9,2)
15   ,COMM       DECIMAL(9,2));
```

Ejemplo

EMPLOYEE

EMPNO	FIRSTNAME	M	LASTNAME	DPT	PH#	HIREDATE	SX	ED	BIRTHDATE	SALARY	COMM
000010	CHRISTINE	I	HAAS	A00	3978	1995-01-01	F	18	1963-08-24	152750	4220
000020	MICHAEL	L	THOMPSON	B01	3476	2003-10-10	M	18	1978-02-02	94250	3300
000030	SALLY	A	KWAN	C01	4738	2005-04-05	F	20	1971-05-11	98250	3060
000050	JOHN	B	GEYER	E01	6789	1979-08-17	M	16	1955-09-15	80175	3214
000060	IRVING	F	STERN	D11	6423	2003-09-14	M	16	1975-07-07	72250	2580
000070	EVA	D	PULASKI	D21	7831	2005-09-30	F	16	2003-05-26	96170	2893
000090	EILEEN	W	HENDERSON	E11	5498	2000-08-15	F	16	1971-05-15	89750	2380
000100	THEODORE	Q	SPENSER	E21	0972	2000-06-19	M	14	1980-12-18	86150	2092
000110	VINCENZO	G	LUCCHESI	A00	3490	1988-05-16	M	19	1959-11-05	66500	3720
000120	SEAN		O'CONNELL	A00	2167	1993-12-05	M	14	1972-10-18	49250	2340
000130	DELORES	M	QUINTANA	C01	4578	2001-07-28	F	16	1955-09-15	73800	1904
000140	HEATHER	A	NICHOLLS	C01	1793	2006-12-15	F	18	1976-01-19	68420	2274
000150	BRUCE		ADAMSON	D11	4510	2002-02-12	M	16	1977-05-17	55280	2022
000160	ELIZABETH	R	PIANKA	D11	3782	2006-10-11	F	17	1980-04-12	62250	1780
000170	MASATOSHI	J	YOSHIMURA	D11	2890	1999-09-15	M	16	1981-01-05	44680	1974
000180	MARILYN	S	SCOUTTEN	D11	1682	2003-07-07	F	17	1979-02-21	51340	1707
000190	JAMES	H	WALKER	D11	2986	2004-07-26	M	16	1982-06-25	50450	1636
000200	DAVID		BROWN	D11	4501	2002-03-03	M	16	1971-05-29	57740	2217
000210	WILLIAM	T	JONES	D11	0942	1998-04-11	M	17	2003-02-23	68270	1462
000220	JENNIFER	K	LUTZ	D11	0672	1998-08-29	F	18	1978-03-19	49840	2387
000230	JAMES	J	JEFFERSON	D21	2094	1996-11-21	M	14	1980-05-30	42180	1774
000240	SALVATORE	M	MARINO	D21	3780	2004-12-05	M	17	2002-03-31	48760	2301
000250	DANIEL	S	SMITH	D21	0961	1999-10-30	M	15	1969-11-12	49180	1534
000260	SYBIL	P	JOHNSON	D21	8953	2005-09-11	F	16	1976-10-05	47250	1380
000270	MARIA	L	PEREZ	D21	9001	2006-09-30	F	15	2003-05-26	37380	2190
000280	ETHEL	R	SCHNEIDER	E11	8997	1997-03-24	F	17	1976-03-28	36250	2100
000290	JOHN	R	PARKER	E11	4502	2006-05-30	M	12	1985-07-09	35340	1227

Ejercicio: Avanzado el tema, y cuando veamos las expresiones, intentad hacer un listado como este en DES

Ejemplo

- ◆ DEPARTMENT(DEPTNO, DEPTNAME, MGRNO, ADMRDEPT, LOCATION)

```
1 CREATE TABLE DEPARTMENT
2   (DEPTNO  CHARACTER(3) PRIMARY KEY
3   ,DEPTNAME VARCHAR(36) NOT NULL
4   ,MGRNO    CHARACTER(6)
5   ,ADMRDEPT CHARACTER(3) NOT NULL
6   ,LOCATION  CHARACTER(16));
```

DEPTNO	DEPTNAME	MGRNO	ADMRDEPT	LOCATION
A00	SPIFFY COMPUTER SERVICE DIV.	000010	A00	-
B01	PLANNING	000020	A00	-
C01	INFORMATION CENTER	000030	A00	-
D01	DEVELOPMENT CENTER	-	A00	-
D11	MANUFACTURING SYSTEMS	000060	D01	-
D21	ADMINISTRATION SYSTEMS	000070	D01	-
E01	SUPPORT SERVICES	000050	A00	-
E11	OPERATIONS	000090	E01	-
E21	SOFTWARE SUPPORT	000100	E01	-
F22	BRANCH OFFICE F2	-	E01	-
G22	BRANCH OFFICE G2	-	E01	-

SQL - LMD

- ◆ Selección:

- SELECT

- ◆ Modificación:

- INSERT
 - UPDATE
 - DELETE

Estructura *general** de la sentencia SELECT

SELECT E₁, ..., E_n

- Describe la salida deseada con:
 - Nombres de columnas
 - Expresiones aritméticas
 - Literales
 - Funciones escalares
 - Funciones de agregación

FROM R₁, ..., R_m

WHERE P

- Nombres de las tablas / vistas
- Condiciones de selección de filas

GROUP BY G₁, ..., G_k
HAVING Q

- Nombre de las columnas a agrupar
- Condiciones de selección de grupo

ORDER BY O₁, ..., O_l

- Nombres de columnas por las que ordenar
(también se pueden escribir expresiones)

* Esta no es la sintaxis completa: SQL es bastante extenso...

Estructura *básica* de la sentencia SELECT

- ◆ Una consulta SQL tiene la forma:

```
SELECT E1, ..., En      /* Lista de expresiones */  
FROM R1, ..., Rm        /* Lista de relaciones. A veces opcional */  
WHERE P;                /* Condición. Cláusula opcional */
```

- Es posible que exista el mismo nombre de atributo en dos relaciones distintas.
- Por ello, se añadiría "*nombre_de_relación.*" antes del nombre de atributo para desambiguar. Por ejemplo: CUENTAS.DNI y CLIENTES.DNI

Practica estos ejemplos

- ◆ Descarga desde el CV y a ejecutar en DESweb el archivo Empleados.sql desde la "BD empleados con tipos STRING" (han cambiado los datos y los tipos para que puedas hacer todos los ejemplos a continuación).
- ◆ Sigue paso a paso cada transparencia y ve probando cada una de las consultas que aparecen. Si se te ocurren otras variaciones también te animo a que las pruebes.
- ◆ Pon todas las consultas en el fichero ejercicio02.sql. Recuerda escribir en el recuadro este nombre y pulsar **Save As**.
- ◆ Cada consulta la ejecutarás desde el *prompt* del sistema. Cuando te funcione bien, la copias al archivo ejercicio02.sql (recuerda pulsar el botón **Save** para que no pierdas los cambios).
- ◆ Activa el modo solo SQL para centrarte en este lenguaje con el comando **/sql**

Proyección de algunas columnas

- ◆ Proyectar significa elegir algunas columnas de la relación del **FROM** para que aparezcan en el resultado.
- ◆ La siguiente consulta extrae solo 3 columnas de la tabla de departamentos. Escríbelas en el *prompt* del sistema y comprueba que la salida coincida. Despues cópiala a ejercicio02.sql

```
SELECT DEPTNO, DEPTNAME, ADMRDEPT  
FROM DEPARTMENT;
```

DEPARTMENT.DEPTNO	DEPARTMENT.DEPTNAME	DEPARTMENT.ADMRDEPT
A00	SPIFFY COMPUTER SERVICE	A00
B01	PLANNING	A00
C01	INFORMATION CENTER	A00
D01	DEVELOPMENT CENTER	A00
D11	MANUFACTURING SYSTEMS	D01
D21	ADMINISTRATION SYSTEMS	D01
E01	SUPPORT SERVICES	A00
E11	OPERATIONS	E01
E21	SOFTWARE SUPPORT	E01
F22	BRANCH OFFICE F2	E01
G22	BRANCH OFFICE G2	E01

Eliminación de filas duplicadas

- ◆ SQL permite duplicados en el resultado
- ◆ Para eliminar las tuplas repetidas se utiliza la cláusula **DISTINCT**.
- ◆ También es posible pedir explícitamente la inclusión de filas repetidas mediante el uso de la cláusula **ALL** (aunque es el comportamiento predeterminado y no es necesario escribirlo).

```
SELECT ADMRDEPT  
FROM DEPARTMENT;
```

ADMRDEPT
A00
A00
A00
A00
D01
D01
A00
E01
E01
...

Equivalentes



```
SELECT ALL ADMRDEPT  
FROM DEPARTMENT;
```

```
SELECT DISTINCT ADMRDEPT  
FROM DEPARTMENT;
```

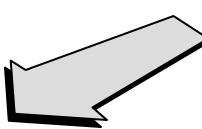
ADMRDEPT
A00
D01
E01

Estos puntos suspensivos indican que hay más filas que no se muestran.

Eliminación de filas duplicadas (ejemplo)

- ◆ ¿Qué trabajos realiza cada departamento?

SELECT DISTINCT WORKDEPT, JOB
FROM EMPLOYEE;



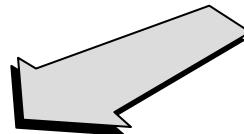
EMPLOYEE.WORKDEPT	EMPLOYEE.JOB
A00	CLERK
B01	MANAGER
C01	ANALYST
E01	MANAGER
D11	DESIGNER
D21	CLERK
E11	FIELDRP
E21	FIELDRP
A00	PRES
A00	SALESREP
C01	MANAGER
D11	MANAGER
D21	MANAGER
E11	OPERATOR
E11	MANAGER

¿Qué ocurre si no ponemos
DISTINCT?

Proyección de todos los atributos

- ◆ Se puede pedir la proyección de todos los atributos de la consulta utilizando el símbolo '*'
 - La tabla resultante contendrá todos los atributos de las tablas que aparecen en la cláusula **FROM**.

SELECT * FROM DEPARTMENT;



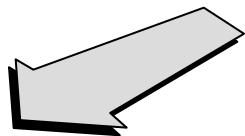
DEPTNO	DEPTNAME	MGRNO	ADMRDEPT	LOCATION
A00	SPIFFY COMPUTER SERVICE DIV.	000010	A00	
B01	PLANNING	000020	A00	
C01	INFORMATION CENTER	000030	A00	
D01	DEVELOPMENTCENTER	-----	A00	
D11	MANUFACTURING SYSTEMS	000060	D01	
D21	ADMINISTRATION SYSTEMS	000070	D01	
E01	SUPPORT SERVICES	000050	A00	
E11	OPERATIONS	000090	E01	
E21	SOFTWARE SUPPORT	000100	E01	
...				

Salida ordenada

- ◆ SQL permite controlar el orden en el que se presentan las tuplas de una relación mediante la cláusula **ORDER BY**.
- ◆ De hecho, sin incluir **ORDER BY**, no podemos saber cómo ordenará el resultado.
- ◆ La cláusula **ORDER BY** tiene la forma
 - ORDER BY E1 <SENTIDO>, ..., En <SENTIDO>**
 - **E1, ..., En** son expresiones (en particular, atributos) de la relación resultante de la consulta. Cada **Ai** también puede ser un número que indique el número de la columna del resultado o un alias (ya se verá).
 - **Ei <SENTIDO>** controla si la ordenación es Ascendente **ASC** o descendente **DESC** por el campo (o expresión) **Ei**. Si no se especifica el sentido, la ordenación es ascendente por defecto.
- ◆ La ordenación se realiza tras haber ejecutado la consulta sobre las tuplas resultantes.
- ◆ La ordenación puede convertirse en una operación costosa dependiendo del tamaño de la relación resultante.

Salida ordenada (ejemplo)

```
SELECT DEPTNO, DEPTNAME, ADMRDEPT  
FROM DEPARTMENT  
ORDER BY ADMRDEPT ASC;
```



Equivalentes

```
SELECT DEPTNO,  
DEPTNAME, ADMRDEPT  
FROM DEPARTMENT  
ORDER BY 3 ASC;
```

DEPTNO	DEPTNAME	ADMRDEPT
A00	SPIFFY COMPUTER SERVICE DIV.	A00
C01	INFORMATION CENTER	A00
B01	PLANNING	A00
E01	SUPPORTSERVICES	A00
D01	DEVELOPMENTCENTER	A00
D11	MANUFACTURING SYSTEMS	D01
D21	ADMINISTRATION SYSTEMS	D01
E21	SOFTWARE SUPPORT	E01
E11	OPERATIONS	E01
...		

Salida ordenada con dos criterios

```
SELECT ADMRDEPT, DEPTNAME, DEPTNO  
FROM DEPARTMENT  
ORDER BY ADMRDEPT ASC, DEPTNO DESC;
```

ADMRDEPT	DEPTNAME	DEPTNO
A00	SUPPORT SERVICES	E01
A00	DEVELOPMENT CENTER	D01
A00	INFORMATION CENTER	C01
A00	PLANNING	B01
A00	SPIFFY COMPUTER SERVICE DIV.	A00
D01	ADMINISTRATION SYSTEMS	D21
D01	MANUFACTURING SYSTEMS	D11
E01	SOFTWARE SUPPORT	E21
E01	OPERATIONS	E11
...		

Aquí se ordena ascendentemente por ADMRDEPT. Como hay varios valores iguales en este campo, por cada uno de ellos, las filas se ordenarán descendente por DEPTNO. Por ejemplo, en el recuadro, el valor que se repite para ADMRDEPT es A00. Las filas se ordenan en ese grupo por los valores de DEPTNO (el primero es E01 y el último es A00).

Selección de filas

- ◆ La cláusula **WHERE** permite filtrar (quitar) filas de la relación resultante.
 - La condición de filtrado se especifica como un predicado (una condición lógica).
- ◆ El predicado de la cláusula **WHERE** puede ser simple o complejo
 - Se utilizan los conectores lógicos **AND** (conjunción), **OR** (disyunción), **XOR** (disyunción exclusiva) y **NOT** (negación)
- ◆ Las expresiones pueden contener
 - Predicados de comparación (**=, <, > ...**)
 - **BETWEEN / NOT BETWEEN**
 - **IN / NOT IN** (con y sin subconsultas)
 - **LIKE / NOT LIKE**
 - **IS NULL / IS NOT NULL**
 - **ALL, SOME / ANY** (subconsultas)
 - **EXISTS** (subconsultas)

Selección de filas

◆ Predicados de comparación

- Operadores: $=$, \neq (es el \neq , que también se puede escribir como \neq),
 $<$, \leq , \geq , $>$

◆ **BETWEEN** *Op1* **AND** *Op2*

- Es el operador de comparación para intervalos de valores o fechas.

◆ **IN** es el operador que permite comprobar si un valor se encuentra en un conjunto.

- Puede especificarse un conjunto de valores (*Val1*, *Val2*, ...)
- Puede utilizarse el resultado de otra consulta **SELECT**.

Selección de filas (cont.)

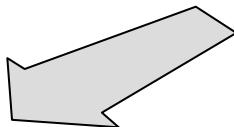
- ◆ **LIKE** es el operador de comparación de cadenas de caracteres con comodines.
 - SQL distingue entre mayúsculas y minúsculas
 - Las cadenas de caracteres se incluyen entre comillas simples
 - SQL permite definir **patrones** a través de los siguientes caracteres:
 - '%', que es equivalente a "cualquier subcadena de caracteres" (incluso vacía)
 - '_', que es equivalente a "cualquier carácter" (y debe haber uno)
 - Se puede indicar un carácter de escape si se busca uno de los comodines en el patrón, como en:

```
select email from employees where email like '%_%' escape '_';
% (empleados con e-mail que contenga un '_')
```
 - ¡No lo uses en lugar del operador de igualdad!
WHERE ADMRDEPT LIKE 'A00';
- ◆ **IS NULL** e **IS NOT NULL** son los operadores de comparación para valores nulos.

Ejemplo de selección de filas

- ◆ ¿Qué departamentos informan al A00?

```
SELECT DEPTNO, ADMRDEPT  
FROM DEPARTMENT  
WHERE ADMRDEPT='A00';
```



DEPTNO	ADMRDEPT
A00	A00
B01	A00
C01	A00
D01	A00
E01	A00

Las cadenas de caracteres (como 'A00') se escriben siempre encerradas entre comillas simples.

- ◆ Cuando lo ejecutes, lee la siguiente transparencia.

Ejemplo de selección de filas

- ◆ Fíjate en el **Warning** (aviso) que emite DES en esta consulta:

```
DES> SELECT DEPTNO, ADMRDEPT  
FROM DEPARTMENT WHERE ADMRDEPT='A00';
```

Warning: [Sem] Constant output column "ADMRDEPT" with value "A00".

Info: 5 tuples computed.

- ◆ Esto es una de las ayudas de este sistema (y que no encontrarás en ningún otro) que pretende ayudarte a cazar posibles errores.
- ◆ Aquí te avisa de que la columna **ADMRDEPT** va a tomar siempre el mismo valor para todas las filas (precisamente porque la condición lo especifica así). Si se sabe cuál es el valor, el sistema se pregunta si realmente quieres que aparezca o si se trata de un error.

Cadenas de caracteres que incluyen un apóstrofe

- ◆ Si una cadena de caracteres incluye un apóstrofe, como O'CONNELL en EMPLOYEE, no se puede simplemente encerrar entre otros apóstrofes:

```
DES> SELECT FIRSTNAME  
      FROM EMPLOYEE  
     WHERE LASTNAME='O'CONNELL';
```

Error: (SQL) Expected infix SQL operator or (SQL) Expected end of SELECT statement near "SELECT FIRSTNAME
 FROM EMPLOYEE
 WHERE LASTNAME='O'"

- ◆ Pulsa el botón **Clear** (se pierde el coloreado de sintaxis).
- ◆ Escribe el apellido de una de estas dos formas: 'O' 'CONNELL' o bien 'O\ 'CONNELL'. Ambas son alternativas que puedes usar según en qué sistema (en DES puedes usar cualquiera de ellas).

```
DES> SELECT FIRSTNAME  
      FROM EMPLOYEE  
     WHERE LASTNAME='O' 'CONNELL';
```

Info: 1 tuple computed.

```
DES> SELECT FIRSTNAME  
      FROM EMPLOYEE  
     WHERE LASTNAME='O\ 'CONNELL';
```

Info: 1 tuple computed.

Ejemplo de selección de filas

- ◆ Necesito el apellido y el nivel de formación de los empleados cuyo nivel de formación es mayor o igual a 19

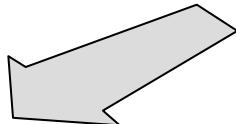
```
SELECT LASTNAME, EDLEVEL  
FROM EMPLOYEE  
WHERE EDLEVEL >= 19;
```

EMPLOYEE.LASTNAME	EMPLOYEE.EDLEVEL
KWAN	20
LUCCHESI	19

Ejemplo de selección de filas

- ◆ Necesito saber el número de empleado, apellido y fecha de nacimiento de aquellos que hayan nacido a partir del 1 de enero de 1985.

```
SELECT EMPNO, LASTNAME, BIRTHDATE  
FROM EMPLOYEE  
WHERE BIRTHDATE >= DATE '1985-01-01'  
ORDER BY BIRTHDATE;
```



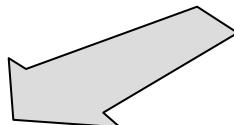
EMPNO	LASTNAME	BIRTHDATE
000290	PARKER	1985-07-09
000240	MARINO	2002-03-31
000210	JONES	2003-02-23
000070	PULASKI	2003-05-26
000270	PEREZ	2003-05-26

El estándar de SQL establece que para escribir una constante de tipo fecha hay que poner **DATE** delante de la fecha (encerrada entre comillas simples). El formato de la fecha (año-mes-día) es también un estándar de ISO. Este formato lo puedes cambiar en DES con el comando **/date_format**.

Múltiples condiciones - AND

- ◆ Necesito el número de empleado, el trabajo y el nivel de formación de los analistas con un nivel de educación 20

```
SELECT EMPNO, JOB, EDLEVEL  
FROM EMPLOYEE  
WHERE JOB='ANALYST' AND EDLEVEL=20;
```



EMPNO	JOB	EDLEVEL
000030	ANALYST	20

- ◆ Aquí DES también te advertirá de las columnas constantes.

Múltiples condiciones: AND/OR

- ◆ Obtener el número de empleado, el trabajo y el nivel de formación de todos los analistas con un nivel 16 y de todos los empleados de nivel 18. La salida se ordena por trabajo y nivel:

```
SELECT EMPNO, JOB, EDLEVEL  
FROM EMPLOYEE  
WHERE (JOB='ANALYST' AND EDLEVEL=16)  
      OR EDLEVEL=18  
ORDER BY JOB, EDLEVEL;
```



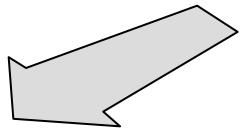
EMPLOYEE.EMPNO	EMPLOYEE.JOB	EMPLOYEE.EDLEVEL
000010	CLERK	18
000220	DESIGNER	18
000140	MANAGER	18
000020	MANAGER	18

DES todavía no sabe avisar de las columnas constantes cuando se incluye OR.

SELECT con BETWEEN

- ◆ Obtener el número de empleado y el nivel de todos los empleados con un nivel entre 12 y 15, ordenado por nivel educativo:

```
SELECT EMPNO, EDLEVEL  
FROM EMPLOYEE  
WHERE EDLEVEL BETWEEN 12 AND 15  
ORDER BY EDLEVEL;
```

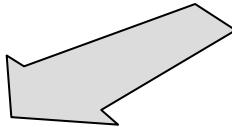


EMPLOYEE.EMPNO	EMPLOYEE.EDLEVEL
000290	12
000100	14
000230	14
000120	14
000270	15
000250	15

SELECT con IN

- ◆ Listar los apellidos y nivel de formación de todos los empleados de nivel 14, 19 o 20, ordenado por nivel educativo y apellido:

```
SELECT LASTNAME, EDLEVEL  
FROM EMPLOYEE  
WHERE EDLEVEL IN (14, 19, 20)  
ORDER BY EDLEVEL, LASTNAME;
```



Estas dos sentencias son equivalentes, pero al usar IN se escribe menos y queda más legible.

```
SELECT LASTNAME, EDLEVEL  
FROM EMPLOYEE  
WHERE EDLEVEL=14 OR  
EDLEVEL=19 OR  
EDLEVEL=20  
ORDER BY EDLEVEL, LASTNAME;
```

EMPLOYEE.LASTNAME	EMPLOYEE.EDLEVEL
JEFFERSON	14
O'CONNELL	14
SPENSER	14
LUCCHESI	19
KWAN	20

LIKE (ejemplo)

- ◆ Obtener el apellido de todos los empleados cuyo apellido empiece por **G**:

```
SELECT LASTNAME  
FROM EMPLOYEE  
WHERE LASTNAME LIKE 'G%' ;
```

Databases	Answer	EMPLOYEE
		EMPLOYEE.LASTNAME
		GEYER
		Pages: 1 1 of 1 Rows: 1

LIKE (ejemplos con %)

```
SELECT LASTNAME  
FROM EMPLOYEE  
WHERE LASTNAME LIKE '%SON' ;
```

THOMPSON
HENDERSON
ADAMSON
JEFFERSON
JOHNSON

```
SELECT LASTNAME  
FROM EMPLOYEE  
WHERE LASTNAME LIKE '%M%N%' ;
```

THOMPSON
ADAMSON
MARINO

Fíjate que, al no indicar ORDER BY, los resultados podrían estar en cualquier orden (depende del sistema).

LIKE (ejemplo con _)

- ◆ ¿Qué empleados tienen una C como segunda letra de su apellido?

```
SELECT LASTNAME  
FROM EMPLOYEE  
WHERE LASTNAME LIKE '_C%';
```

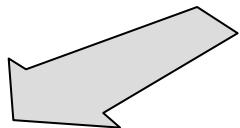
EMPLOYEE.LASTNAME	▲	▼
SCOUTTEN	▲	▼
SCHNEIDER	▲	▼

Pages: 1 1 of 1 Rows: 2

Búsqueda por patrones: NOT LIKE

- ◆ Necesito todos los departamentos excepto aquellos cuyo número *no* empiece por 'D':

```
SELECT DEPTNO, DEPTNAME  
FROM DEPARTMENT  
WHERE DEPTNO NOT LIKE 'D%' ;
```



DEPTNO	DEPTNAME
A00	SPIFFY COMPUTER SERVICE DIV.
B01	PLANNING
C01	INFORMATION CENTER
E01	SUPPORT SERVICES
E11	OPERATIONS
E21	SOFTWARE SUPPORT
...	

Tipos SQL y valores literales

- ◆ La norma SQL define un conjunto de tipos para las columnas de las tablas.
 - Habitualmente cada SGBD tiene tipos propios o particularidades para los tipos de la norma SQL.
- ◆ Es necesario consultar el manual del SGBD para obtener información acerca de los tamaños máximos de almacenamiento. Por ejemplo:
 - En el caso de cadenas, cuál es la longitud máxima de almacenamiento.
 - En el caso de tipos numéricos, cuál es el rango de valores posibles.

Tipos SQL y valores literales

Tipo	Ejemplo
BIGINT	8589934592
INTEGER	186282
SMALLINT	186
NUMERIC(8,2)	999999.99 (precisión, escala)
DECIMAL(8,2)	999999.99 (precisión, escala)
REAL	6.02257E23
DOUBLE PRECISION	3.141592653589
FLOAT	6.02257E23
CHARACTER(<i>n</i>)	'GREECE' ' n = 15 (caracteres)
VARCHAR(<i>n</i>)	'hola' n = 4 (caracteres)
STRING	'hola' (cualquier número de caracteres; no está definido en el estándar de SQL)
DATE	date 'YYYY-MM-DD'
TIME	time 'hh:mm:ss.ccc'
TIMESTAMP	timestamp 'YYYY-MM-DD hh:mm:ss.ccc'

Conversión de tipos

- ◆ SQL incluye un sistema de tipos débiles en el sentido en que puede aplicar un *ajuste (o coerción) automático de tipos*.
- ◆ Esto permite comparar valores de distinto tipo en la mayoría de los sistemas (Access, por ejemplo, no) como en:

```
SELECT 1 FROM dual  
WHERE '5'=5;
```

- ◆ En DES recuerda activar el ajuste automático de tipos (si no lo estuviera ya) con */type_casting on*
- ◆ Otro ejemplo útil es operar con fechas, como en:

```
SELECT 1 FROM dual  
WHERE current_date='2022-10-05';
```

Conversión de tipos

- ◆ No obstante, a veces no es posible realizar la conversión automática.
- ◆ El estándar proporciona la función para la *conversión manual de tipos*:

CAST(*Expresión* AS *Tipo*)

- ◆ Donde *Expresión* es la expresión a convertir al tipo *Tipo*.
- ◆ Por ejemplo:

SELECT CAST(1 AS FLOAT); → {(1.0)}

SELECT CAST('2021-03-11' AS DATE); → {(date '2021-03-11')}

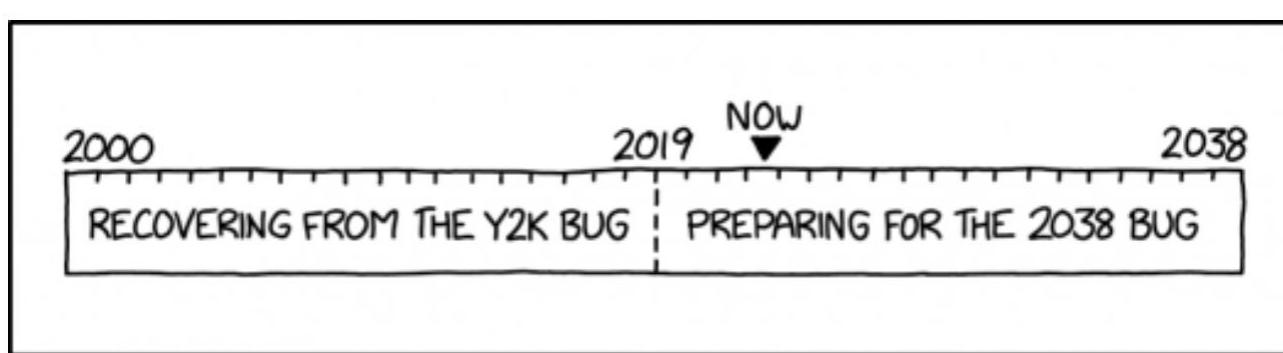
- ◆ Sistemas concretos también proporcionan funciones propias de conversión de tipos, como **Oracle**:

TO_DATE(*Cadena*, *Formato*)

Esto convierte una cadena a un valor de fecha siguiendo el formato especificado. E.g., **TO_DATE('31-12-2012', 'DD-MM-YYYY')**

Oracle también proporciona la función dual **TO_STRING(*Expresión*)** que en este caso puede devolver un string a partir de una fecha (por ejemplo).

2038 DATE Problem



REMINDER: BY NOW YOU SHOULD HAVE FINISHED YOUR Y2K RECOVERY AND BE SEVERAL YEARS INTO 2038 PREPARATION.

- ◆ Unix time: 1970-01-01 00:00:00 – 2038-01-19 03:14:07.
 - 32 bits para representar segundos, con rango: $-2^{31} - (2^{31}-1)$

MySQL> SELECT UNIX_TIMESTAMP('2038-01-20') Result; --> {(0)} Arreglado a partir de MySQL 8.0.28

SQL-Server> SELECT DATEDIFF(SECOND,'1970-01-01', '2038-01-19 03:14:08') AS 'Result'; --> The datediff function resulted in an overflow Hay que usar DATEDIFF_BIG

¡Cuidado con las operaciones sobre las fechas! Pueden retornar un tipo INT de 32 bits y vuelta a empezar...

Expresiones

- ◆ Aunque SQL no es un lenguaje de programación de propósito general, permite definir expresiones calculadas.
- ◆ Estas expresiones pueden contener:
 - Referencias a columnas
 - Valores literales
 - Operadores aritméticos
 - Llamadas a funciones (escalares y de agregación)
- ◆ Los operadores aritméticos son los habituales: +, -, * y /
 - Estos operadores se aplican a valores numéricos (aunque algún sistema sobrecarga + para cadenas).
 - Los operadores '+' y '-' habitualmente funcionan para fechas (internamente se almacenan como números).
- ◆ Aunque las normas SQL definen un conjunto mínimo de funciones, los SGBD proporcionan una gran variedad.
 - Es necesario consultar el manual del SGBD particular.

Funciones matemáticas comunes

Descripción	IBM DB2	SQL Server	Oracle	MySQL	DES
Valor absoluto	ABS	ABS	ABS	ABS	ABS
Menor entero \geq valor	CEIL	CEILING	CEIL	CEILING	CEILING
Mayor entero \leq valor	FLOOR	FLOOR	FLOOR	FLOOR	FLOOR
Potencia	POWER	POWER	POWER	POWER	POWER
Redondeo a un número de cifras decimales	ROUND	ROUND	ROUND	ROUND	ROUND
Truncamiento a un número de cifras decimales	TRUNC	TRUNC	TRUNC	TRUNC	TRUNC
Resto de la división entera	MOD	%	MOD	%	MOD

Funciones de cadena

Descripción	IBM DB2	SQL Server	Oracle	MySQL	DES
Convierte todos los caracteres a minúsculas	LOWER	LOWER	LOWER	LOWER	LOWER
Convierte todos los caracteres a mayúsculas	UPPER	UPPER	UPPER	UPPER	UPPER
Elimina los blancos del final de la cadena	RTRIM	RTRIM	RTRIM	RTRIM	RTRIM
Elimina los blancos del comienzo de la cadena	LTRIM	LTRIM	LTRIM	LTRIM	LTRIM
Devuelve una subcadena	SUBSTR	SUBSTRING	SUBSTR	SUBSTRING	SUBSTR
Concatena dos cadenas	CONCAT	+	CONCAT	CONCAT	(SQL Estándar) + CONCAT

Otras funciones: LEFT, RIGHT, SPACE, REPEAT, LENGTH...

Funciones de selección

- ◆ SQL dispone de varias funciones de selección que permiten devolver valores bajo determinadas condiciones:
 - **CASE**: es una selección condicional con ramificaciones. Tiene dos formas distintas aunque muy parecidas.
- ◆ Las siguientes se pueden ver como casos particulares de **CASE**:
 - **IIF** (Immediate IF): versión de MS Access para una selección entre dos valores.
 - **NULLIF**: si sus dos valores son iguales, se devuelve **NULL**; si no, se devuelve el primero.
 - **NVL** (Not null value): selecciona el primer valor no nulo de entre dos.
 - **COALESCE**: selecciona el primer valor no nulo de una lista.
- ◆ No todos los sistemas tienen todas estas funciones.
- ◆ Como veremos a continuación, estos casos particulares de **CASE** ahoran escribir bastante.

Función CASE

CASE

```
WHEN C_1 THEN resultado_1  
WHEN C_2 THEN resultado_2  
...  
WHEN C_n THEN resultado_n  
ELSE resultado_x  
END
```

CASE A

```
WHEN V_1 THEN resultado_1  
WHEN V_2 THEN resultado_2  
...  
WHEN V_n THEN resultado_n  
ELSE resultado_x  
END
```

La primera condición C_i que se cumpla hace que se devuelva el valor $resultado_i$. Si no se cumple ninguna, se devuelve $resultado_x$.

El primer valor V_i que sea igual a A hace que se devuelva el valor $resultado_i$. Si no hay ninguno, se devuelve $resultado_x$.

- ◆ En ambas formas de CASE, la línea ELSE es opcional. Si no se incluye, se devolvería NULL en caso de no cumplirse ningún WHERE.

Ejemplos de la función CASE

```
SELECT EMPNO, FIRSTNAME, LASTNAME,  
CASE  
    WHEN EDLEVEL<15 THEN 'SECONDARY'  
    WHEN EDLEVEL<19 THEN 'COLLEGE'  
    ELSE 'POST GRADUATE'  
END AS EDUCATION  
FROM EMPLOYEE;
```



empno	firstname	lastname	education
000010	CHRISTINE	HAAS	COLLEGE
000020	MICHAEL	THOMPSON	COLLEGE
000030	SALLY	KWAN	POST GRADUATE
000050	JOHN	GEYER	COLLEGE
000060	IRVING	STERN	COLLEGE
000070	EVA	PULASKI	COLLEGE
000090	EILEEN	HENDERSON	COLLEGE
000100	THEODORE	SPENSER	SECONDARY
000110	VINCENZO	LUCCHESI	POST GRADUATE

• • •

```
SELECT EMPNO, FIRSTNAME, LASTNAME,  
CASE SEX  
    WHEN 'F' THEN 'FEMALE'  
    WHEN 'M' THEN 'MALE'  
    ELSE 'INVALID VALUE'  
END AS SEX  
FROM EMPLOYEE  
WHERE SALARY < 38000;
```



empno	firstname	lastname	sex
000270	MARIA	PEREZ	FEMALE
000280	ETHEL	SCHNEIDER	FEMALE
000290	JOHN	PARKER	MALE

Pages: 1 1 of 1 Rows: 3

Función IIF

- ◆ **IIF(condición, result_1, result_2)** Si se cumple la condición, se devuelve **result_1**, si no, se devuelve **result_2**. Por tanto, es equivalente a:

CASE

WHEN condición THEN result_1

ELSE result_2

END

Función **NULLIF**

- ◆ **NULLIF(valor_1,valor_2)** Si **valor_1** es igual a **valor_2**, se devuelve **NULL**; si no, se devuelve **valor_1**. Por tanto, es equivalente a:

CASE

```
WHEN valor_1=valor_2 THEN NULL  
ELSE valor_1  
END
```

Función NVL

- ◆ **NVL(valor,resultado)** Si **valor** es **NULL**, se devuelve **resultado**; si no, se devuelve **valor**. Por tanto, es equivalente a:

CASE

 WHEN valor IS NULL THEN resultado

 ELSE valor

END

Función COALESCE

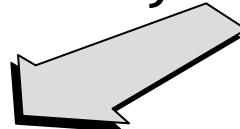
- ◆ COALESCE(valor_1, . . . , valor_n) Devuelve el primer *valor_i* que no sea NULL. Si no hay ninguno que lo sea, devuelve NULL. Por tanto, es equivalente a:

```
CASE
    WHEN valor_1 IS NOT NULL THEN valor_1
    WHEN valor_2 IS NOT NULL THEN valor_2
    ...
    WHEN valor_n IS NOT NULL THEN valor_n
END
```

Operaciones aritméticas (cont.)

- ◆ Necesito saber el resultado de aumentar el salario de mis empleados en un 3,75%:

```
SELECT EMPNO, SALARY, SALARY*1.0375
      FROM EMPLOYEE
     WHERE SALARY < 200000
   ORDER BY EMPNO;
```



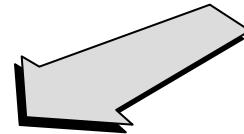
EMPLOYEE.EMPNO	EMPLOYEE.SALARY	\$a17
000010	152750	158478.125
000020	94250	97784.37500000001
000030	98250	101934.37500000001
000050	80175	83181.5625
000060	72250	74959.375
000070	96170	99776.37500000001

Observa que, al no dar nombre a la columna de la expresión calculada, el sistema le asigna un nombre arbitrario.

Expresiones en predicados

- ◆ Necesito saber los empleados cuyo porcentaje de bono (sobre el sueldo) supere el 4%:

```
SELECT EMPNO, SALARY, (BONUS/SALARY)*100 AS PERCENTAGE  
FROM EMPLOYEE  
WHERE (BONUS/SALARY)*100 > 4  
ORDER BY EMPNO;
```



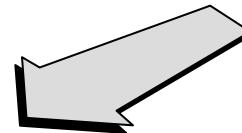
Aquí usamos un alias para darle nombre a la columna resultado de evaluar la expresión.

EMPLOYEE.EMPNO	EMPLOYEE.SALARY	PERCENTAGE
000050	80175	4.008730901153726
000110	66500	5.593984962406015
000120	49250	4.751269035532995
000170	44680	4.4180841539838855
000220	49840	4.789325842696629
000230	42180	4.205784732100522
000240	48760	4.719031993437244
000270	37380	5.858747993579454
000280	36250	5.793103448275862

Uso de funciones aritméticas

- ◆ Necesito saber el resultado de aumentar el salario de mis empleados en un 3,75%, pero solo con dos decimales:

```
SELECT EMPNO, SALARY, TRUNC(SALARY*1.0375, 2)  
FROM EMPLOYEE  
ORDER BY EMPNO;
```

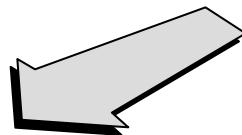


EMPLOYEE.EMPNO	EMPLOYEE.SALARY	\$a16
000010	152750	158478.12
000020	94250	97784.37
000030	98250	101934.37
000050	80175	83181.56
000060	72250	74959.37
000070	96170	99776.37
000090	89750	93115.62
000100	86150	89380.62
000110	66500	68993.75
000120	49250	51096.87
...		

Uso de funciones de cadena

- ◆ Necesito un listado de nombres de empleados del departamento A00 en el que aparezca primero el apellido del empleado, después una coma y finalmente el nombre:

```
SELECT LASTNAME + ', ' + FIRSTNAME AS FULLNAME  
FROM EMPLOYEE  
WHERE WORKDEPT = 'A00'  
ORDER BY FULLNAME;
```



FULLNAME

HAAS, CHRISTA
LUCCHESI, VINCENZO
O'CONNELL, SEAN

Ejercicio

- ◆ ¿Recuerdas el ejercicio propuesto en la página [17](#)? Intenta resolverlo en consola con las funciones de cadena y aritméticas.
- ◆ Por ejemplo, si hacemos:

```
DES> SELECT EMPNO, FIRSTNAME FROM EMPLOYEE;

answer(EMPLOYEE.EMPNO:char(6),EMPLOYEE.FIRSTNAME:varchar(12)) ->
{
    answer('000010','CHRISTINE'),
    answer('000020','MICHAEL'),
    answer('000030','SALLY'),
```

- ◆ No se parece mucho a lo pedido. Pero lo siguiente ya se va pareciendo:

```
DES> SELECT 'EMPNO  FIRSTNME' UNION
SELECT '-----' UNION
SELECT EMPNO+' '+FIRSTNAME FROM EMPLOYEE;

answer($a0:string) ->
{
    answer('EMPNO  FIRSTNME'),
    answer('-----'),
    answer('000010 CHRISTINE'),
    answer('000020 MICHAEL'),
    answer('000030 SALLY'),
```

- ◆ ¿Sabrías terminarlo? (Puede que necesites **LENGTH** y **REPEAT**). Si te decides a hacerlo, guárdalo en tu sesión como **ejercicio09.sql**.

Operadores de conjunto

- ◆ SQL incluye las siguientes operaciones:
 - *Sentencia SQL UNION Sentencia SQL*
 - *Sentencia SQL INTERSECT Sentencia SQL*
 - *Sentencia SQL EXCEPT Sentencia SQL*
 - (**MINUS** en Oracle y ambos soportados en DES)
- ◆ Cada una de estas formas representan una sentencia SQL completa (no es posible añadirles otras cláusulas como, por ejemplo, **ORDER BY**).
- ◆ Por definición, los operadores de conjunto (**UNION** , **INTERSECT** y **EXCEPT**) eliminan las tuplas duplicadas (llevan implícitamente asociado el modificador **DISTINCT**, que también se puede escribir explícitamente).
 - Para retener duplicados se debe utilizar **<Operador> ALL**
- ◆ Los operandos no pueden ser directamente tablas:
 - **programadores UNION analistas** -- No está permitido
 - **SELECT * FROM programadores UNION SELECT * FROM analistas;** -- Así sí lo está
 - (esto es solo un ejemplo, aún no hemos definido estas tablas)

UNION

- ◆ Cada **SELECT** debe tener el mismo número de columnas.
- ◆ Las columnas correspondientes deben tener tipos de datos compatibles.
- ◆ Recuerda que **UNION** elimina duplicados: si no los necesitas, añade **ALL** para mejorar el rendimiento.
- ◆ Si se indica, **ORDER BY** debe ser la última cláusula de la sentencia.

UNION (ejemplo)

- ◆ ¿Cuáles son los códigos de departamento que aparecen en las dos tablas?

```
SELECT DEPTNO FROM DEPARTMENT  
UNION  
SELECT WORKDEPT FROM EMPLOYEE;
```

- ◆ ¿Todos los departamentos tienen asignado algún empleado?

```
DES> SELECT DEPTNO FROM DEPARTMENT;
```

Info: 11 tuples computed.

```
DES> SELECT DISTINCT WORKDEPT FROM EMPLOYEE;
```

Info: 8 tuples computed.

- ◆ No, porque solo aparecen 8 distintos en la tabla de empleados.

INTERSECT

- ◆ La variante especificada con **ALL** devuelve el mínimo número de filas de cada operando por cada fila distinta.
- ◆ La variante especificada con **DISTINCT** descarta los duplicados.

t	x
x	x
1	1
1	1
1	2
2	4
3	
3	

s	x
x	x
1	1
1	1
2	
4	

```
SELECT * FROM t  
INTERSECT ALL  
SELECT * FROM s;
```



1
1
2

```
SELECT * FROM t  
INTERSECT DISTINCT  
SELECT * FROM s;
```



1
2

Recuerda que el modificador **DISTINCT** es el predeterminado y aquí se podría omitir.

EXCEPT

- ◆ La variante especificada con **ALL** devuelve el número de filas del primer operando menos el del segundo por cada fila distinta.
- ◆ La variante especificada con **DISTINCT** descarta los duplicados de los operandos y después actúa como en el caso anterior.

t	x
x	x
1	1
1	1
1	2
2	4
3	
3	

s	x
x	x
1	1
1	1
2	
4	

```
SELECT * FROM t  
EXCEPT ALL  
SELECT * FROM s;
```



1
3
3

```
SELECT * FROM t  
EXCEPT DISTINCT  
SELECT * FROM s;
```



3

Recuerda que el modificador **DISTINCT** es el predeterminado y aquí se podría omitir.

La división en SQL

- ◆ Ya vimos una operación importante en el álgebra relacional: la división.
- ◆ Sin embargo, tanto el estándar SQL como la mayoría de los SGBD **no implementan** esta operación.
- ◆ No obstante, en DES sí lo hace mediante el operador infijo **DIVISION**.
- ◆ Por ejemplo, dado el siguiente esquema de relación:

emp (dni, nom) trab (dni, npro)

la siguiente consulta obtiene el DNI de los empleados que trabajan en todos los proyectos en los que trabaja el empleado con DNI = 123.

```
SELECT dni FROM
  (SELECT * FROM trab)
  DIVISION
  (SELECT npro FROM trab WHERE dni=123);
```

Funciones de agregación

- ◆ Las *funciones de agregación* (también conocidas como *funciones de agregados* o *funciones de columna*) son funciones que toman una colección (conjunto o multiconjunto) de valores de entrada y devuelve un solo valor.
- ◆ Las funciones de agregación habituales son: **COUNT** (recuento), **SUM** (sumatorio), **AVG** (promedio), **MIN** (mínimo) y **MAX** (máximo).
- ◆ Los datos de entrada para **SUM** y **AVG** deben ser una colección de números, pero el resto de operadores pueden operar también sobre colecciones de datos de tipo no numérico.

Funciones de agregación

- ◆ Cálculo del sumatorio → **SUM (expresión)**
- ◆ Cálculo de la media → **AVG (expresión)**
- ◆ Obtener el valor mínimo → **MIN (expresión)**
- ◆ Obtener el valor máximo → **MAX (expresión)**
- ◆ Contar el número de filas → **COUNT (*)**
 - Los valores **NULL** *sí* se cuentan.
- ◆ Contar el número de valores ***no nulos*** en una columna → **COUNT (*nombre-columna*)**
- ◆ Contar el número de valores ***distintos y no nulos*** en una columna → **COUNT (DISTINCT *nombre-columna*)**
- ◆ En general, una función de agregación ***no puede incluir a otra*** en su argumento (Oracle sí, e.g., **MAX(AVG(Salary))**)

Funciones de agregación (ejemplo)

- ◆ La siguiente consulta usa todas estas funciones:

```
SELECT SUM(SALARY)      AS SUM,
       AVG(SALARY)       AS AVG,
       MIN(SALARY)        AS MIN,
       MAX(SALARY)        AS MAX,
       COUNT(*)           AS TODOS,
       COUNT(MIDINIT)    AS MIDINIT_NO_NULOS,
       COUNT(DISTINCT WORKDEPT) AS DEPT
  FROM EMPLOYEE;
```

SUM	AVG	MIN	MAX	TODOS	MIDINIT_NO_	DEPT
1763905	65329.814814	35340	152750	27	24	8

- ◆ Fíjate que hay 27 filas en EMPLOYEE (columna TODOS) pero solo hay 24 de ellas que tengan un valor distinto de NULL en MIDINIT. Añade los ejemplos a partir de aquí a ejercicio03.sql.

La agrupación: GROUP BY

- ◆ De manera predeterminada y como se ha visto en el ejemplo, las funciones de agregación se aplican a todas las tuplas resultantes de la consulta.
- ◆ No obstante, podemos agrupar las tuplas resultantes para poder aplicar las funciones de agregación a grupos específicos utilizando la cláusula **GROUP BY**.
- ◆ En la cláusula **SELECT** de consultas que utilizan funciones de agregación solo pueden aparecer funciones de columna, pero:
 - En caso de utilizar **GROUP BY**, también pueden aparecer columnas utilizadas en la agrupación.

GROUP BY (ejemplo)

- ◆ Por ejemplo, podríamos determinar cuántos empleados hay por cada departamento:

```
SELECT WORKDEPT, COUNT(*) AS EMPLEADOS  
FROM EMPLOYEE  
GROUP BY WORKDEPT  
ORDER BY WORKDEPT;
```

EMPLOYEE.WORKDEPT	EMPLEADOS
A00	3
B01	1
C01	3
D11	9
D21	6
E01	1
E11	3
E21	1

GROUP BY: cómo funciona

◆ ¿Cómo funciona la agrupación?

- A partir de las filas que se calculan del resultado de la consulta sin **GROUP BY**, se crean tantos grupos de filas como valores diferentes haya para el criterio de agrupación.
 - En el caso del ejemplo anterior, hay 8 departamentos distintos en la columna **WORKDEPT** de **EMPLOYEE** (se crean 8 grupos). El primer grupo sería el '**A00**' (hay 3 filas con este valor en **WORKDEPT**) y el último el '**E21**' (solo una fila para este otro).
- Para cada grupo, se aplica la función (o funciones) de agregación que se hayan especificado en la **SELECT**, como si se tratase de una consulta sin **GROUP BY**.
 - En el ejemplo, se hace el recuento de filas de cada grupo. Así, a '**A00**' le corresponde 3 (**COUNT(*)=3**) y a '**E21**', 1.
- Finalmente, se genera una fila por cada grupo esas 2 columnas (departamento y resultado de la función de agregación).
- El criterio de agrupación (columna **WORKDEPT**) se puede incluir en la salida de resultados porque solo hay un valor de **WORKDEPT** por cada grupo.

GROUP BY (ejemplo)

Necesito conocer los salarios de todos los empleados de los departamentos A00, B01, y C01. Además, para estos departamentos quiero conocer su masa salarial.



Para entender mejor cómo funciona, primero voy a ver cuáles son todas las filas *sin agrupar*.

Después voy a *agrupar* para conseguir lo que me piden.

```
SELECT      WORKDEPT, SALARY  
FROM        EMPLOYEE  
WHERE       WORKDEPT IN ('A00', 'B01', 'C01')  
ORDER BY    WORKDEPT;
```

```
SELECT      WORKDEPT, SUM(SALARY) AS SUM  
FROM        EMPLOYEE  
WHERE       GROUP BY WORKDEPT  
           WORKDEPT IN ('A00', 'B01', 'C01')  
           WORKDEPT;  
ORDER BY    WORKDEPT;
```

EMPLOYEE.WORKDEPT	EMPLOYEE.SALARY
A00	152750
A00	49250
A00	66500
B01	94250
C01	98250
C01	68420
C01	73800

Lenguaje SQL

EMPLOYEE.WORKDEPT	SUM
A00	268500
B01	94250
C01	240470

Filtrado de grupos: **HAVING**

- ◆ Adicionalmente se pueden aplicar condiciones sobre los grupos utilizando la cláusula **HAVING**.
- ◆ Esta cláusula es parecida a **WHERE**, pero no hay que confundirlas: **WHERE** filtra filas *antes* de agrupar, mientras que **HAVING** filtra grupos (que realmente se reducen a filas) *después* de agrupar.
- ◆ En la condición de **HAVING** van a aparecer en general funciones de agregación (si no, podría resultar sospechoso).

GROUP BY - HAVING (ejemplo)

- ◆ Necesito mostrar los departamentos con más de un empleado:

Para entender mejor cómo funciona, primero voy a ver todos los grupos *sin filtrar*.

```
SELECT WORKDEPT, COUNT(*) AS NUMB  
FROM EMPLOYEE  
GROUP BY WORKDEPT  
ORDER BY WORKDEPT;
```



EMPLOYEE.WORKDEPT	NUMB
A00	3
B01	1
C01	3
D11	9
D21	6
E01	1
E11	3
E21	1

Después *filtro* los grupos para conseguir el resultado final:

```
SELECT WORKDEPT, COUNT(*) AS NUMB  
FROM EMPLOYEE  
GROUP BY WORKDEPT  
HAVING COUNT(*) > 1  
ORDER BY WORKDEPT;
```



EMPLOYEE.WORKDEPT	NUMB
A00	3
C01	3
D11	9
D21	6
E11	3

GROUP BY - HAVING (ejemplo)



Para entender mejor cómo funciona, primero voy a ver todos los grupos *sin filtrar*.

Después voy a *filtrar* para conseguir lo que me piden.

```
SELECT      WORKDEPT, SUM(SALARY) AS SUM  
FROM        EMPLOYEE  
WHERE       WORKDEPT IN ('A00', 'B01', 'C01')  
GROUP BY    WORKDEPT  
ORDER BY    WORKDEPT;
```

```
SELECT      WORKDEPT, SUM(SALARY) AS SUM  
FROM        EMPLOYEE  
WHERE       WORKDEPT IN ('A00', 'B01', 'C01')  
GROUP BY    WORKDEPT  
HAVING     SUM(SALARY) > 100000  
ORDER BY    WORKDEPT;
```

EMPLOYEE.WORKDEPT	SUM
A00	268500
B01	94250
C01	240470

Lenguaje SQL

EMPLOYEE.WORKDEPT	SUM
A00	268500
C01	240470

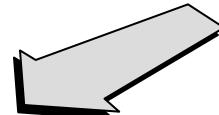
GROUP BY - HAVING (ejemplo)

Necesito, agrupado por departamento, los trabajadores que no sean managers, designer, y fieldrep, con una media de salario mayor que 25000€.



Ya lo entiendo bien, lo hago directamente:

```
SELECT WORKDEPT, JOB, AVG(SALARY) AS AVG  
FROM EMPLOYEE  
WHERE JOB NOT IN ('MANAGER', 'DESIGNER', 'FIELDREP')  
GROUP BY WORKDEPT, JOB  
HAVING AVG(SALARY) > 50000  
ORDER BY WORKDEPT, JOB;
```



EMPLOYEE.WORKDEPT	EMPLOYEE.JOB	AVG
A00	CLERK	152750
A00	PRES	66500
C01	ANALYST	98250
D21	CLERK	54734

GROUP BY - HAVING (ejemplo)

- ◆ Necesito mostrar cada uno de los departamentos con el promedio del nivel educativo de sus empleados y el promedio de años que llevan trabajando para la empresa, siempre que sea superior a 20 años.
- ◆ Nota: **CURRENT_DATE** devuelve la fecha actual y si restas dos fechas se devuelve el número de días entre ellas. Revisa que el formato de fechas sea adecuado con el comando **/date_format**.

Lo hago en dos pasos para entenderlo mejor:

```
SELECT WORKDEPT, AVG(EDLEVEL) AS ED,  
TRUNC(AVG(CURRENT_DATE-HIREDATE)/365.25) AS YEARS  
FROM EMPLOYEE  
GROUP BY WORKDEPT  
ORDER BY AVG(EDLEVEL);
```



EMPLOYEE	ED	YEARS
E21	14	19
E11	15	18
D21	15.5	16
E01	16	40
D11	16.5	18
A00	17	27
C01	18	15
B01	18	16

```
SELECT WORKDEPT, AVG(EDLEVEL) AS ED,  
TRUNC(AVG(CURRENT_DATE-HIREDATE)/365.25) AS YEARS  
FROM EMPLOYEE  
GROUP BY WORKDEPT  
HAVING TRUNC(AVG(CURRENT_DATE-HIREDATE)/365.25) > 20  
ORDER BY AVG(EDLEVEL);
```



EMPLOYEE	ED	YEARS
E01	16	40
A00	17	27

GROUP BY - HAVING (ejemplo)

- ◆ Esta consulta se puede reescribir reusando los alias para que quede más concisa:

```
SELECT WORKDEPT, AVG(EDLEVEL) AS ED, TRUNC(AVG(CURRENT_DATE-HIREDATE)/365.25) AS YEARS  
FROM EMPLOYEE  
GROUP BY WORKDEPT  
HAVING YEARS > 20  
ORDER BY ED;
```

- ◆ Tambien podríamos haber usado la notación posicional en la ordenación (el 1 se refiere a la 1^a columna del resultado, el 2 a la segunda...):

```
SELECT WORKDEPT, AVG(EDLEVEL) AS ED, TRUNC(AVG(CURRENT_DATE-HIREDATE)/365.25) AS YEARS  
FROM EMPLOYEE  
GROUP BY WORKDEPT  
HAVING YEARS > 20  
ORDER BY 2;
```

- ◆ No obstante, se desaconseja usar esta notación porque puedes equivocarte si cambias el orden de la lista de proyección y se te olvida cambiar el ORDER BY. Si usas alias o expresiones no tienes este problema.

GROUP BY - HAVING (ejemplo)

- ◆ ¿Podrías enunciar lo que calcula SELECT 2?

SELECT 1:

```
SELECT  
    WORKDEPT,  
    TRUNC(AVG(EDLEVEL),1) AS ED,  
    MIN(BONUS) AS MIN  
FROM EMPLOYEE  
GROUP BY WORKDEPT;
```



EMPLOYEE	ED	MIN
A00	17	2340
B01	18	3300
C01	18	1904
D11	16.5	1462
D21	15.5	1380
E01	16	3214
E11	15	1227
E21	14	2092

SELECT 2:

```
SELECT  
    WORKDEPT,  
    TRUNC(AVG(EDLEVEL),1) AS ED,  
    MIN(BONUS) AS MIN  
FROM EMPLOYEE  
GROUP BY WORKDEPT  
HAVING MIN(BONUS) >= 3000;
```



EMPLOYEE	ED	MIN
B01	18	3300
E01	16	3214

Consultas con varias tablas

- ◆ Puede ser necesario incluir más de una tabla en la cláusula **FROM** de una consulta porque se necesiten datos que no se encuentran en una única tabla.
- ◆ Siguiendo nuestro ejemplo, podría ser necesario conocer datos de empleados y de departamentos.
- ◆ En el ejemplo, ambas tablas están relacionadas por una restricción de integridad referencial en el código del departamento.
- ◆ Esto significa que a cada empleado le corresponde un departamento (indicado por su código) cuyos datos se encuentran en la tabla de departamentos (como por ejemplo el nombre del departamento).
- ◆ Gráficamente:

Consultas con varias tablas

EMPLOYEE

EMPNO	LASTNAME	WORKDEPT...
000010	HAAS	A00
000020	THOMPSON	C01
000030	KWAN	C01
000040	PULASKI	D21
.	.	.

DEPARTMENT

DEPTNO	DEPTNAME	...
A00	SPIFFY COMPUTER SERVICE DIV.	
C01	INFORMATION CENTER	
D01	DEVELOPMENT CENTER	
D21	ADMINISTRATION SYSTEMS	
.	.	.

Consulta con dos tablas

- ◆ Podemos entonces preguntar por el código de empleados (cuyo apellido comience por HAAS), apellidos, y código y nombre de su departamento con:

```
SELECT EMPNO, LASTNAME, WORKDEPT, DEPTNAME  
FROM EMPLOYEE, DEPARTMENT  
WHERE WORKDEPT = DEPTNO AND  
      LASTNAME LIKE 'HAAS%';
```

EMPLOYEE.EMPNO	EMPLOYEE.LASTNAME	EMPLOYEE.WORKDEPT	DEPARTMENT.DEPTNAME
000010	HAAS	A00	SPIFFY COMPUTER SERVICE

- ◆ Fíjate en la importancia de la condición **WORKDEPT = DEPTNO** : se está exigiendo que las filas de ambas tablas coincidan precisamente en los campos de la clave externa:

EMPLOYEE.WORKDEPT -> DEPARTMENT.DEPTNO

Consulta con dos tablas

- ◆ Añadamos los dos campos involucrados en la clave externa y *quitemos* la correspondencia para ver cómo funciona:

```
SELECT EMPNO, LASTNAME, WORKDEPT, DEPTNO, DEPTNAME  
FROM EMPLOYEE, DEPARTMENT  
WHERE LASTNAME LIKE 'HAAS%';
```

EMPLOYEE.EMPNO	EMPLOYEE.LASTNAME	EMPLOYEE.WORKDEPT	DEPARTMENT.DEPTNO	DEPARTMENT.DEPTNAME
000010	HAAS	A00	A00	SPIFFY COMPUTER SERV
000010	HAAS	A00	B01	PLANNING
000010	HAAS	A00	C01	INFORMATION CENTER
000010	HAAS	A00	D01	DEVELOPMENT CENTER
000010	HAAS	A00	D11	MANUFACTURING SYSTE
000010	HAAS	A00	D21	ADMINISTRATION SYSTE
000010	HAAS	A00	E01	SUPPORT SERVICES
000010	HAAS	A00	E11	OPERATIONS
000010	HAAS	A00	E21	SOFTWARE SUPPORT
000010	HAAS	A00	F22	BRANCH OFFICE F2
000010	HAAS	A00	G22	BRANCH OFFICE G2

Consulta con dos tablas

- ◆ Al aplicar la condición LASTNAME **LIKE** 'HAAS%' solo hay un empleado que la cumpla (código de empleado **000010**).
- ◆ Esta fila se combina con cada una de las filas de la tabla de departamentos (que son 11) porque todas ellas pasan la condición.
- ◆ En el resultado aparecen todas las columnas indicadas en la cláusula **SELECT**
- ◆ La condición de correspondencia permite quedarme exactamente con la fila de interés (los datos del empleado y los de *su* departamento).
- ◆ Si no incluimos la condición de correspondencia, DES protesta:

Warning: [Sem] Missing join condition for [EMPLOYEE,DEPARTMENT].

- ◆ "Join" se puede traducir aquí como "correspondencia".

Consulta con dos tablas

- ◆ En definitiva, poner una coma separando dos tablas equivale a un producto cartesiano \times (todas las combinaciones de filas entre dos tablas).
- ◆ Por ejemplo, para las tablas $t_1(c_1)$ y $t_2(c_2)$, el resultado $t_1 \times t_2$ resulta ser:

$$\begin{array}{c|c} c_1 & c_2 \\ \hline a & c \\ b & d \end{array} \quad \times \quad \begin{array}{c|c} c_1 & c_2 \\ \hline a & c \\ a & d \\ b & c \\ b & d \end{array}$$

$t_1 \qquad t_2$

- ◆ Crea las dos tablas, inserta los datos y escribe la consulta para comprobarlo.

Correspondencias en SQL estándar

- ◆ Aunque las correspondencias se puedan expresar como hemos visto, el estándar de SQL sugiere implícitamente que usemos una cláusula específica para ellas: la combinación **JOIN-ON**. En nuestro ejemplo:

```
SELECT EMPNO, LASTNAME, WORKDEPT, DEPTNAME  
FROM EMPLOYEE JOIN DEPARTMENT ON WORKDEPT = DEPTNO  
WHERE LASTNAME LIKE 'HAAS%';
```

- ◆ La idea es separar las condiciones de correspondencia del resto de condiciones de filtro.
- ◆ De este modo, si se te olvidase incluir la condición con **ON** (en IBM DB2):

```
db2 => SELECT EMPNO, LASTNAME, WORKDEPT, DEPTNAME FROM EMPLOYEE JOIN DEPARTMENT  
WHERE LASTNAME LIKE 'HAAS%'  
SQL0104N Se ha encontrado un símbolo "LASTNAME" inesperado a continuación de  
"JOIN DEPARTMENT WHERE". Los símbolos esperados pueden incluir: "ON".  
SQLSTATE=42601
```

- ◆ (Como hemos visto, DES no rechaza la consulta, simplemente avisa de que puede haber un problema).

Consultas con más tablas

- ◆ Es posible incluir todas las tablas que necesites en la cláusula **FROM**.
- ◆ Pero recuerda:

Tendrás que incluir tantas condiciones de correspondencia como clave externas estén involucradas en las tablas.

- ◆ Si tienes dos tablas, necesitas solo una condición, como en el ejemplo de los departamentos y empleados.
- ◆ Si tienes tres tablas, necesitarás dos condiciones de correspondencia (como en el ejemplo que veremos en la siguiente página).
- ◆ En general, para n tablas necesitas $n-1$ condiciones.

Consulta con tres tablas

PROJECT

PROJNO	PROJNAME	DEPTNO	...
AD3100	ADMIN SERVICES	D01	
AD3110	GENERAL AD SYSTEMS	D21	
AD3111	PAYROLL PROGRAMMING	D21	
AD3112	PERSONELL PROGRAMMING	D21	
AD3113	ACCOUNT. PROGRAMMING	D21	
IF1000	QUERY SERVICES	C01	

Crea el archivo
ejercicio04.sql para las
siguientes consultas.



DEPARTMENT

DEPTNO	DEPTNAME	MGRNO	...
A00	SPIFFY COMPUTER SERVICE DIV.	000010	
B01	PLANNING	000020	
C01	INFORMATION CENTER	000030	
D01	DEVELOPMENT CENTER	-----	
D11	MANUFACTURING SYSTEMS	000060	
D21	ADMINISTRATION SYSTEMS	000070	
E01	SUPPORT SERVICES	000050	



EMPLOYEE

EMPNO	FIRSTNME	MIDINIT	LASTNAME	...
000010	CHRISTA	I	HAAS	
000020	MICHAEL	L	THOMPSON	
000030	SALLY	A	KWAN	
000050	JOHN	B	GEYER	
000060	IRVING	F	STERN	
000070	EVA	D	PULASKI	
000090	EILEEN	W	HENDERSON	
000100	THEODORE	Q	SPENSER	

Consulta con tres tablas. Ejemplo

- ◆ Quiero saber los proyectos que lleva el departamento cuyo código es 'D21', y cuál es su director, listando los campos PROJNO, DEPTNAME, MGRNO y LASTNAME, ordenado por PROJNO (por tanto, necesito incluir las tres tablas).
- ◆ Observa cómo vamos a ir construyendo la relación resultante de la correspondencia de tres tablas:

- ◆ Primero se hace la correspondencia entre PROJECT y DEPARTMENT:
PROJECT JOIN DEPARTMENT
ON PROJECT.DEPTNO = DEPARTMENT.DEPTNO

PROJECT			
PROJNO	PROJNAME	DEPTNO	...
AD3100	ADMIN SERVICES	D01	
AD3110	GENERAL AD SYSTEMS	D21	
AD3111	PAYROLL PROGRAMMING	D21	
AD3112	PERSONNEL PROGRAMMING	D21	
AD3113	ACCOUNT PROGRAMMING	D21	
IF1000	QUERY SERVICES	C01	

DEPARTMENT			
DEPTNO	DEPTNAME	MGRNO	...
A01	SPIFFY COMPUTER SERVICE DIV.	000010	
B01	PLANNING	000020	
C01	INFORMATION CENTER	000030	
D01	DEVELOPMENT CENTER	000040	
D11	MANUFACTURING SYSTEMS	000060	
D21	ADMINISTRATION SYSTEMS	000070	
E01	SUPPORT SERVICES	000050	

- ◆ Después se añade la correspondencia entre DEPARTMENT y EMPLOYEE:
PROJECT JOIN DEPARTMENT
ON PROJECT.DEPTNO = DEPARTMENT.DEPTNO
JOIN EMPLOYEE ON DEPARTMENT.MGRNO = EMPLOYEE.EMPNO

EMPLOYEE				
EMPNO	FIRSTNAME	MIDINIT	LASTNAME	...
000010	CHRISTA	L	HAAS	
000020	MICHAEL	A	THOMPSON	
000030	SALLY	B	KWAN	
000050	JOHN	F	GEYER	
000060	IRVING	D	STERN	
000070	EVA		PULASKI	
000090	EILEEN	W	HENDERSON	
00100	THEODORE	Q	SPENSER	

- ◆ Es decir, simplemente se va añadiendo una combinación JOIN-ON por cada nueva tabla que necesites incluir en el resultado.
- ◆ Nuestra consulta queda como se ve en la siguiente página.

Consulta con tres tablas. Ejemplo

- ◆ Quiero saber los proyectos que lleva el departamento cuyo código es 'D21', y cuál es su director, listando los campos PROJNO, DEPTNAME, MGRNO y LASTNAME, ordenado por PROJNO:

```
SELECT PROJNO, DEPTNAME, MGRNO, LASTNAME  
FROM PROJECT JOIN DEPARTMENT ON PROJECT.DEPTNO = DEPARTMENT.DEPTNO  
                JOIN EMPLOYEE   ON DEPARTMENT.MGRNO = EMPLOYEE.EMPNO  
WHERE DEPARTMENT.DEPTNO = 'D21'  
ORDER BY PROJNO;
```

PROJNO	DEPTNAME	MGRNO	LASTNAME
AD3110	ADMINISTRATION SYSTEMS	000070	PULASKI
AD3111	ADMINISTRATION SYSTEMS	000070	PULASKI
AD3112	ADMINISTRATION SYSTEMS	000070	PULASKI
AD3113	ADMINISTRATION SYSTEMS	000070	PULASKI

Equivalentes

```
SELECT PROJNO, DEPTNAME, MGRNO, LASTNAME  
FROM PROJECT, DEPARTMENT, EMPLOYEE  
WHERE PROJECT.DEPTNO = DEPARTMENT.DEPTNO AND  
      DEPARTMENT.MGRNO = EMPLOYEE.EMPNO AND  
      DEPARTMENT.DEPTNO = 'D21'  
ORDER BY PROJNO;
```

Consulta con tres tablas. Ejemplo

- ◆ A veces resulta prolijo usar la calificación de los atributos con las tablas de donde proceden, por lo que resulta de utilidad usar alias para abbreviar.
- ◆ En este último ejemplo (omitimos **AS**, que es opcional en algunos sistemas):

```
SELECT PROJNO, DEPTNAME, MGRNO, LASTNAME  
FROM PROJECT P JOIN DEPARTMENT D ON P.DEPTNO = D.DEPTNO  
                  JOIN EMPLOYEE E   ON D.MGRNO = E.EMPNO  
WHERE D.DEPTNO = 'D21'  
ORDER BY PROJNO;
```

- ◆ Aunque no sea necesario, puedes calificar otros atributos, como:

```
SELECT P.PROJNO, D.DEPTNAME, D.MGRNO, E.LASTNAME  
FROM PROJECT P JOIN DEPARTMENT D ON P.DEPTNO = D.DEPTNO  
                  JOIN EMPLOYEE E   ON D.MGRNO = E.EMPNO  
WHERE D.DEPTNO = 'D21'  
ORDER BY P.PROJNO;
```

- ◆ La calificación solo es necesaria cuando haya que resolver ambigüedades, como el código de departamento, cuyo nombre se repite en las tablas PROJECT y DEPARTMENT. Así, se podría escribir simplemente:

```
SELECT PROJNO, DEPTNAME, MGRNO, LASTNAME  
FROM PROJECT P JOIN DEPARTMENT D ON P.DEPTNO = D.DEPTNO  
                  JOIN EMPLOYEE     ON MGRNO = EMPNO  
WHERE D.DEPTNO = 'D21'  
ORDER BY PROJNO;
```

Simplificación de correspondencias

- ◆ Si los campos de correspondencia se denominan igual en ambas tablas, se puede añadir **NATURAL** antes de **JOIN** para que el sistema aplique automáticamente la igualdad entre los campos comunes.
- ◆ En este último ejemplo:

```
SELECT PROJNO, DEPTNAME, MGRNO, LASTNAME  
FROM PROJECT NATURAL JOIN DEPARTMENT  
      JOIN EMPLOYEE ON MGRNO = EMPNO  
WHERE DEPTNO = 'D21'  
ORDER BY PROJNO;
```

- ◆ Observa que ya no es necesaria ninguna calificación porque los campos comunes solo van a aparecer una vez en la relación. Para comprobarlo, seleccionamos todos los campos:

```
SELECT * FROM PROJECT NATURAL JOIN DEPARTMENT  
      JOIN EMPLOYEE ON MGRNO = EMPNO;
```

answer(

```
PROJECT.PROJNO:char(6),  
PROJECT.PROJNAME:varchar(24),  
PROJECT.DEPTNO:string,  
PROJECT.RESPEMP:char(6),  
PROJECT.PRSTAFF:float,  
PROJECT.PRSTDATE:datetime(date),  
PROJECT.PRENDATE:datetime(date),  
PROJECT.MAJPROJ:char(6),
```

```
DEPARTMENT.DEPTNAME:varchar(36),  
DEPARTMENT.MGRNO:char(6),  
DEPARTMENT.ADMRDEPT:char(3),  
DEPARTMENT.LOCATION:char(16),
```

```
EMPLOYEE.EMPNO:char(6),  
EMPLOYEE.FIRSTNAME:varchar(12),  
EMPLOYEE.MIDINIT:char(1),  
EMPLOYEE.LASTNAME:varchar(15),  
EMPLOYEE.WORKDEPT:char(3),  
...) ->
```

DEPTNO solo aparece una vez en el esquema de la respuesta

Consultas con tablas repetidas

- ◆ Es posible que tengamos que recuperar datos de la misma tabla pero con *distintos papeles* (o *roles*).
- ◆ Por ejemplo: **queremos determinar los empleados que son mayores que el director de su departamento.**
- ◆ Necesitamos conocer dos fechas de nacimiento: la del empleado y la del director, pero el director es a su vez un empleado (habría que tomar la fecha de la misma tabla).
- ◆ El problema es que no podemos poner una condición como `EMPLOYEE.BIRTHDATE > EMPLOYEE.BIRTHDATE` porque se refiere a la misma fila de la tabla `EMPLOYEE` y, por lo tanto, esta condición es falsa para cualquier fila (un valor determinado no puede ser mayor que sí mismo).
- ◆ Así, hay que incluir dos veces la tabla `EMPLOYEE`: una para el empleado y otra para el director.
- ◆ Mediante renombramiento (alias), a una le daremos el papel **E** (de empleado) y a la otra el papel **M** (de manager - director) como se muestra en la siguiente página.

Consultas con tablas repetidas

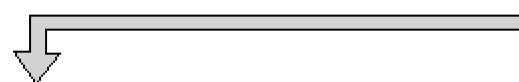
1. Recuperar la fila de un empleado de la tabla EMPLOYEE (E)

EMPNO	...	LASTNAME	WORKDEPT	...	BIRTHDATE	...
000100		SPENSER	E21		1956-12-18	
000330		LEE	E21		1941-07-18	



2. Recuperar el nº departamento de DEPARTMENT (D)

DEPTNO	DEPTNAME	MGRNO	ADMNRDEPT
E21	SOFTWARE SUPPORT	000100	E21



3. Recuperar el director en EMPLOYEE (M)

EMPNO	...	LASTNAME	WORKDEPT	...	BIRTHDATE	...
000100		SPENSER	E21		1956-12-18	
000330		LEE	E21		1941-07-18	

Consultas con tablas repetidas

- ◆ Escribimos así la consulta:

```
SELECT E.EMPNO, E.LASTNAME, E.BIRTHDATE, M.BIRTHDATE, M.EMPNO  
FROM EMPLOYEE E JOIN DEPARTMENT D ON E.WORKDEPT = D.DEPTNO  
                  JOIN EMPLOYEE M ON D.MGRNO      = M.EMPNO  
WHERE E.BIRTHDATE < M.BIRTHDATE;
```

E.EMPNO	E.LASTNAME	E.BIRTHDATE	M.BIRTHDATE	M.EMPNO
000120	O'CONNELL	1972-10-18	1973-08-24	000010
000200	BROWN	1971-05-29	1975-07-07	000060
000230	JEFFERSON	1980-05-30	2003-05-26	000070
000240	MARINO	2002-03-31	2003-05-26	000070
000250	SMITH	1979-11-12	2003-05-26	000070
000260	JOHNSON	1976-10-05	2003-05-26	000070

Pages: 1 1 of 1 Rows: 6

Consultas con tablas repetidas

- ◆ **Ejemplo:** en el siguiente esquema de base de datos, queremos obtener el nombre de los empleados con al menos dos hijos.



Crea las dos tablas,
inserta los datos y
escribe la consulta que
se pide a continuación

La tabla HIJOS expresa que el empleado identificado por DNI tiene un hijo con nombre NOMH.

```
SELECT NOM
FROM EMP JOIN HIJOS AS H1 ON EMP.DNI = H1.DNI
          JOIN HIJOS AS H2 ON EMP.DNI = H2.DNI
WHERE H1.NOMH <> H2.NOMH;
```

Esto es solo un aviso
porque DES encuentra
extraño que hagas así la
consulta (en lugar de por
ejemplo usar agregados)

Warning: [Sem] Tuple variables are always identical
for the different occurrences of "HIJOS".

- ◆ Intenta componer a mano el resultado de esta consulta con la instancia:

EMP	DNI	NOM
1	Pedro	
2	Eva	

HIJOS

Lenguaje SQL

	DNI	NOMH
1	Anita	
1	Pepín	
2	Carol	

Consultas con tablas repetidas

- ◆ **Ejemplo:** en el siguiente esquema de base de datos se piden los apellidos de cada empleado y de su supervisor.

EMP (DNI, NOM, AP, SUELDO, ND, DNISUPERV)

```
SELECT E.AP, S.AP  
FROM EMP E, EMP S  
WHERE E.DNISUPERV = S.DNI;
```

Crea la tabla, inserta algunos datos que imagines y escribe esta consulta.

Reuniones externas (**OUTER JOIN**)

- ◆ Las reuniones que hemos estudiado se denominaban *internas* (de ahí, **INNER JOIN**).
- ◆ Existe otro tipo de reunión en el estándar de SQL que se denomina *externa* (**OUTER JOIN**).
- ◆ Su objetivo es poder incluir en el resultado todas las filas de una relación aunque no encuentre correspondencia con otra con la que se reúne.
- ◆ Al igual que en la interna, donde **INNER** era opcional, **OUTER** también lo es, pero en este caso se debe incluir qué tipo de reunión necesitamos:
 - Por la izquierda (**LEFT**): incluye todas las tuplas de la relación que esté a la izquierda de **LEFT JOIN**.
 - Por la derecha (**RIGHT**): incluye todas las tuplas de la relación que esté a la derecha de **RIGHT JOIN**.
 - Completa (**FULL**): incluye todas las tuplas de la relación tanto a la derecha como a la izquierda de **FULL JOIN**.

Reuniones externas: ejemplo con LEFT JOIN

- ◆ Por ejemplo, podríamos necesitar un listado con los nombres de *todos* los departamentos y los empleados en ellos.
- ◆ Si no hubiese ningún empleado en alguno, la información de empleado debería aparecer vacía (con nulos).
- ◆ Podemos construir la relación del **FROM** con una reunión externa por la izquierda del siguiente modo:

DEPARTMENT **LEFT JOIN** EMPLOYEE **ON** WORKDEPT=**DEPTNO**

- ◆ Así, aparecerá en el resultado todas las filas de DEPARTMENT (porque es la tabla que está a la izquierda del operador).
- ◆ Lo formulamos de forma completa en SQL como aparece en la siguiente página:

Reuniones externas (OUTER JOIN)

```
SELECT DEPTNAME, EMPNO, LASTNAME  
FROM DEPARTMENT LEFT JOIN EMPLOYEE  
ON WORKDEPT=DEPTNO  
ORDER BY DEPTNAME, LASTNAME;
```

DEPTNAME	EMPNO	LASTNAME
ADMINISTRATION SYSTEMS	000230	JEFFERSON
ADMINISTRATION SYSTEMS	000260	JOHNSON
ADMINISTRATION SYSTEMS	000240	MARINO
ADMINISTRATION SYSTEMS	000270	PEREZ
ADMINISTRATION SYSTEMS	000070	PULASKI
ADMINISTRATION SYSTEMS	000250	SMITH
BRANCH OFFICE F2	null	null
BRANCH OFFICE G2	null	null
DEVELOPMENT CENTER	null	null
INFORMATION CENTER	000030	KWAN
INFORMATION CENTER	000140	NICHOLLS
INFORMATION CENTER	000130	QUINTANA

El departamento ADMINISTRATION SYSTEMS tiene seis empleados (por eso aparecen seis filas con el mismo nombre de departamento). Más abajo, INFORMATION CENTER, tiene solo tres, y así el resto.

Fíjate que en el resultado aparecen los departamentos que no tienen empleados. Por ello, las columnas que provienen de EMPLOYEE se llenan automáticamente con nulos. Si hubieras hecho esta consulta con INNER JOIN, estas tres filas simplemente no hubiesen aparecido.

Reuniones externas (**RIGHT JOIN**)

- ◆ La versión **RIGHT JOIN** es dual a **LEFT JOIN**. De hecho, se puede conseguir el mismo resultado del ejemplo anterior usando esta otra forma y simplemente intercambiado el orden de las relaciones:

```
SELECT DEPTNAME, EMPNO, LASTNAME  
FROM EMPLOYEE RIGHT JOIN DEPARTMENT  
    ON WORKDEPT=DEPTNO  
ORDER BY DEPTNAME, LASTNAME;
```

- ◆ Usar una u otra versión es potestativo: dependerá de cómo nos resulte más cómodo o intuitivo hacerlo.
- ◆ Una posible guía para seleccionar una versión u otra es fijarse en el orden en que queramos las columnas resultado y usar el mismo orden para las relaciones.
- ◆ Siguiendo esta directriz en el ejemplo, usaríamos la forma **LEFT JOIN** (porque DEPTNAME viene de DEPARTMENT, y EMPNO y LASTNAME de EMPLOYEE)

Reuniones externas (**FULL JOIN**)

- ◆ La versión **FULL JOIN** la usaremos cuando necesitemos que en el resultado aparezcan filas tanto de la relación a la izquierda como las de la derecha que no encuentren correspondencia.
- ◆ Por ejemplo, si quisiésemos listar todos los departamentos y todos los empleados (aunque aún no estén asociados a ningún departamento), usaríamos esta forma:

```
SELECT DEPTNAME, EMPNO, LASTNAME  
FROM DEPARTMENT FULL JOIN EMPLOYEE  
    ON WORKDEPT=DEPTNO  
ORDER BY DEPTNAME, LASTNAME;
```

- ◆ A diferencia de las versiones anteriores, el orden de las relaciones en **FULL JOIN** es irrelevante.
- ◆ Si pruebas esta consulta con la instancia de empleados verás que no hay ningún empleado que no esté asignado a algún departamento.

Reuniones externas

- ◆ En SQL, al igual que en RA, puedes resolver una misma pregunta con distintas consultas.
- ◆ Por ejemplo, para ver los departamentos que no tienen empleados (esta se puede resolver también con **EXISTS**, cfr. pág. 132) podemos plantear la siguiente:

```
SELECT DEPTNO, DEPTNAME
  FROM DEPARTMENT LEFT JOIN EMPLOYEE
    ON WORKDEPT=DEPTNO
 WHERE LASTNAME IS NULL
 ORDER BY DEPTNO;
```

DEPTNO	DEPTNAME
D01	DEVELOPMENT CENTER
F22	BRANCH OFFICE F2
G22	BRANCH OFFICE G2

Guía para resolver consultas con varias tablas

- ◆ A partir del enunciado de la consulta, determinar cuáles son las tablas que se necesitan:
 - Las que contengan columnas que nos pidan en el resultado (cláusula **SELECT**).
 - Las que contengan columnas que deban aparecer en la condición de filtrado (cláusula **WHERE**).
 - Las que contengan columnas que deban aparecer en el *criterio* de agrupación (cláusula **GROUP BY**).
 - Las que contengan columnas que deban aparecer en la *condición* de agrupación (cláusula **HAVING**).
 - Las que contengan columnas que deban aparecer en el orden de agrupación (cláusula **ORDER BY**).
- ◆ Construir la relación en la cláusula **FROM** uniendo todas las tablas necesarias mediante **JOIN - ON**. La condición de reunión es la igualdad de las columnas que intervienen en las restricciones de integridad referencial (claves externas).
- ◆ Incluir las cláusulas que sean necesarias (al menos, **SELECT** y **FROM**).

Resumen: ejecución de consultas SELECT

- ◆ El orden de ejecución de una consulta es el siguiente:
 1. Se calcula la relación indicada en el **FROM** .
 2. Se aplica la condición del **WHERE** a las tuplas de esta relación.
 3. Las tuplas que satisfacen el predicado del **WHERE** se reparten en grupos siguiendo el criterio de **GROUP BY**.
 4. Se ejecuta la cláusula **HAVING** para quitar cada grupo de tuplas anterior que no cumpla la condición.
 5. Los grupos obtenidos tras aplicar **HAVING** son los que serán procesados por **SELECT**, que calculará, en los casos que se incluyan, las funciones de agregación que le acompañan.
 6. A las tuplas resultantes de los pasos anteriores se le aplica la ordenación descrita en la cláusula **ORDER BY**.

Subconsultas

- ◆ Una subconsulta es una instrucción **SELECT** que está anidada (generalmente entre paréntesis) dentro de otra consulta.
- ◆ Una subconsulta puede aparecer en
 - **SELECT** : La subconsulta se utilizará como valor de la columna de la tabla resultante. La subconsulta debe devolver un único valor.
 - **FROM** : La subconsulta se utilizará como una tabla. Según el sistema, será necesario dar nombre a esta tabla con **AS**.
 - **WHERE** : La subconsulta se usa dentro de las condiciones para realizar comparaciones.
 - **HAVING** : La subconsulta se usa de forma similar que en la cláusula **WHERE**.
 - **GROUP BY, ORDER BY** : También se pueden usar, pero no es habitual.
- ◆ Las subconsultas pueden devolver uno o varios valores.
- ◆ Dependiendo del número esperado de valores se utilizarán unos operadores u otros en las condiciones.

Subconsultas

- ◆ Cuando se incluye una subconsulta en la cláusula WHERE puede tener la siguiente forma:

```
SELECT Lista_proyección
FROM   relación
WHERE  expresión operador (SELECT Lista_proyección ...);
```

- ◆ El sistema SQL resuelve la subconsulta solo una vez antes de resolver la consulta principal (aunque esto puede depender del sistema).
- ◆ Los resultados de la subconsulta se utilizan en la consulta principal.
- ◆ ***¡Las subconsultas no existen en el álgebra relacional!***

Subconsultas (cont.)

- ◆ Si estamos seguros de que la subconsulta devuelve un único valor, es posible utilizar los operadores de comparación habituales `=`, `<>`, `<=`, `<`, `>=`, `>`.
- ◆ Para subconsultas que devuelven múltiples valores se pueden utilizar los operadores:
 - `IN / NOT IN`
 - O bien los operadores de comparación habituales junto a los cuantificadores `ANY` (alias `SOME`) y `ALL`.

Obtener resultados con varias consultas

- ◆ ¿Qué empleados tienen salarios mayores que el salario medio?

1^{ra} CONSULTA →
(solo devuelve una fila)

```
SELECT AVG(SALARY)  
FROM EMPLOYEE;
```

→ 65329.81

2^a CONSULTA →

```
SELECT EMPNO, LASTNAME  
FROM EMPLOYEE  
WHERE SALARY > 65329.81;
```

Empieza el nuevo
archivo **ejercicio05.sql**
en DESweb con estas
consultas.



Integramos la primera
consulta en la segunda
como subconsulta

```
SELECT EMPNO, LASTNAME  
FROM EMPLOYEE  
WHERE SALARY > (SELECT AVG(SALARY)  
                  FROM EMPLOYEE);
```

Ejemplo de subconsulta

- ◆ ¿Qué empleados tienen el menor BONUS?

```
SELECT EMPNO, LASTNAME, BONUS  
FROM EMPLOYEE  
WHERE BONUS = (SELECT MIN(BONUS) FROM EMPLOYEE);
```



EMPNO	LASTNAME	BONUS
000290	PARKER	1227

Subconsulta con **IN**

- ◆ El operador **IN** permite determinar si un valor se encuentra en un conjunto (con **NOT IN** se comprobaría lo contrario).
- ◆ Por ejemplo, si queremos saber los **departamentos que no tienen asignados proyectos** podemos enunciar la consulta así:
 - "Por cada departamento que aparezca en la tabla **DEPARTMENT**, determinar si **no hay** una correspondencia de ese departamento en la tabla **PROJECT**".
- ◆ Lo formulamos en SQL como aparece en la siguiente página:

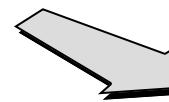
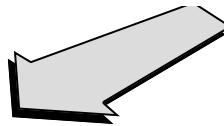
Subconsulta con IN



Tabla DEPARTMENT

<u>DEPTNO</u>	<u>DEPTNAME</u>
A00	SPIFFY COMPUTER SERVICE
B01	PLANNING
C01	INFORMATION CENTER
...	...

```
SELECT DEPTNO, DEPTNAME  
FROM DEPARTMENT  
WHERE DEPTNO NOT IN (SELECT DEPTNO  
                      FROM PROJECT);
```



Resultado final

DEPARTMENT.DEPTNO	DEPARTMENT.DEPTNAME
A00	SPIFFY COMPUTER SERVICE
F22	BRANCH OFFICE F2
G22	BRANCH OFFICE G2

Pages: 1 1 of 1 Rows: 3

Resultado subconsulta

PROJECT.DEPTNO
B01
C01
D01
D11
D21
E01
E11
E21

Pages: 1 1 of 1 Rows: 8

Modificadores ALL, SOME y ANY

- ◆ Si queremos comparar un valor con un conjunto de valores, además de **IN**, podemos usar los modificadores **ALL**, **SOME** y **ANY**, asociados a los operadores relacionales **<**, **>**, **<>**, **>=** y **<=**:
 - **valor > ALL (*subconsulta*)** es cierto si **valor** es mayor que *cada uno de* los valores que devuelve **subconsulta**
 - **valor > ANY (*subconsulta*)** es cierto si **valor** es mayor que *alguno de* los valores que devuelve **subconsulta**
 - **SOME** es sinónimo de **ANY**

Ejemplo: subconsulta con ALL

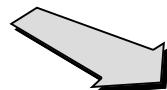
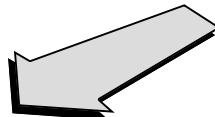
- ◆ Supongamos que necesitamos saber los **empleados cuyo salario sea mayor que la media de salarios de *cada uno de* (es decir, *de todos*) los departamentos.**
- ◆ El propio enunciado sugiere la estructura de la solución:
 - La consulta principal devolvería los datos del empleado con su salario si cumplen la condición.
 - En la condición indicamos que su salario sea mayor que todas las medias de salarios. Esta es una subconsulta en la que agrupamos por departamento para determinar las medias de salarios.
- ◆ Lo formulamos en SQL como aparece en la siguiente página:

Subconsulta con ALL

¿Qué empleados tienen un salario mayor que la media de salarios de cada uno de los departamentos?



```
SELECT LASTNAME, SALARY  
FROM EMPLOYEE  
WHERE SALARY > ALL(SELECT AVG(SALARY)  
                      FROM EMPLOYEE  
                     GROUP BY WORKDEPT);
```



Resultado final

EMPLOYEE.LASTNAME	EMPLOYEE.SALARY
HAAS	152750
KWAN	98250
PULASKI	96170

Lenguaje SQL

Resultado subconsulta

89500
94250
80156.66666666667
56900
53486.66666666664
80175
53780
86150

Ejemplo: subconsulta con ANY

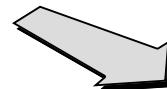
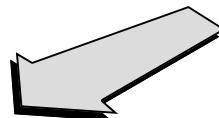
- ◆ Este ejemplo es muy parecido al anterior: en este caso necesitamos saber los **empleados del departamento E11 cuyo salario sea mayor que la media de salarios de al menos un departamento.**
- ◆ Comparado con el ejemplo anterior, hay que añadir una condición extra a la cláusula **WHERE** de la consulta principal, y usar el modificador **ANY** (algún departamento, i.e., al menos uno) en lugar de **ALL**.
- ◆ Lo formulamos en SQL como aparece en la siguiente página:

Subconsulta utilizando ANY o SOME

¿Qué empleados del departamento E11 tienen el salario mayor que la media de salarios de al menos un departamento?



```
SELECT LASTNAME, SALARY  
FROM EMPLOYEE  
WHERE WORKDEPT= 'E11' AND SALARY > ANY (  
    SELECT AVG(SALARY)  
    FROM EMPLOYEE  
    GROUP BY WORKDEPT);
```



Resultado Final

EMPLOYEE.LASTNAME	EMPLOYEE.SALARY
HENDERSON	89750

Lenguaje SQL

89500
94250
80156.666666666667
56900
53486.666666666664
80175
53780
86150

Ejemplo: subconsulta con **HAVING**

- ◆ Las subconsultas pueden aparecer prácticamente en cualquier sitio en que pueda aparecer un valor escalar; en particular en una cláusula **HAVING**.
- ◆ En este caso no usaremos modificadores porque compararemos valores escalares (números) entre si, que son resultado de una función de agregación y de una subconsulta.
- ◆ El enunciado del ejemplo sería: *Necesitamos un listado de los departamentos con un salario medio por empleado (sin incluir a los directores) mayor que el salario medio por empleado de la compañía (también sin directores). El resultado debe estar ordenado ascendente por la media.*
- ◆ Comparado con el ejemplo anterior, hay que añadir la condición **HAVING**, las condiciones sobre los directores, añadir una condición a la subconsulta, además de la ordenación de la consulta externa.
- ◆ Lo formulamos en SQL como aparece en la siguiente página:

Subconsulta en la cláusula HAVING

Necesito un listado de los departamentos con un salario medio por empleado (sin director) mayor que el salario medio por empleado de la compañía (también sin directores). El resultado lo quiero de forma que el departamento que tenga mayor media sea el primero.



```
SELECT WORKDEPT, AVG(SALARY) AS AVG_WORKDEPT  
FROM EMPLOYEE  
WHERE JOB <> 'MANAGER'  
GROUP BY WORKDEPT  
HAVING AVG(SALARY) > (  
    SELECT AVG(SALARY)  
    FROM EMPLOYEE  
    WHERE JOB <> 'MANAGER' )  
ORDER BY AVG_WORKDEPT;
```

Resultado final

EMPLOYEE.WORKDEPT	AVG_WORKDEPT
E21	86150
A00	89500
C01	98250

e SQL

Fíjate que aquí ordenamos según el alias, aunque igualmente podríamos haber ordenado por la expresión (AVG(SALARY))

Resultado subconsulta

\$a15
65469.5

Subconsultas correlacionadas

- ◆ En una subconsulta correlacionada se hace referencia a las columnas de la relación que se construye en la cláusula **FROM** de la consulta principal.
- ◆ El orden de ejecución que aplica el sistema a la consulta cambia (cfr. pág. 116):
 - Primero se resuelve la relación especificada en la consulta principal.
 - Posteriormente se resuelve la subconsulta para cada una de las filas de la consulta principal.
- ◆ Si conocemos algo de programación 😳, esto se puede asemejar al anidamiento de dos bucles donde el índice del bucle externo se usa en el interno.
- ◆ Las subconsultas correlacionadas pueden aparecer con distintos operadores (=, >, **EXISTS...**) y modificadores (**ALL**, **ANY...**)

Ejemplo: subconsulta correlacionada con >

- ◆ En este ejemplo necesito conocer los **empleados cuyo salario es mayor que la media de los salarios de su departamento**.
- ◆ En la subconsulta calculamos la media de los salarios, pero añadiendo la condición de que el departamento para el que se calcula esta media sea el mismo que el departamento del empleado que aparece en la consulta principal.
- ◆ En la siguiente página se muestra el esquema de la consulta:

Ejemplo: subconsulta correlacionada con >

EMPNO	LASTNAME	WORKDEPT	SALARY
000010	HAAS	A00	52750.00
000030	KWAN	C01	38250.00
000120	O'CONNELL	A00	29250.00
000130	QUINTANA	C01	23800.00
000140	NICHOLLS	C01	28420.00

EMPLOYEE

¿Qué empleado tiene el salario mayor que la media de salarios de su departamento?

```
SELECT EMPNO, LASTNAME, SALARY  
FROM EMPLOYEE  
WHERE SALARY >  
      Media de salarios del dept.  
correspondiente
```

En esta instancia de EMPLOYEE (luego veremos el resultado de la instancia completa en la página [132](#)) aparecen solo dos departamentos (A00 y C01), y las medias de los salarios son:



AVG(SALARY)

A00 41000.00000000



AVG(SALARY)

C01 30156.66666666

Ejemplo: subconsulta correlacionada con >

EMPNO	LASTNAME	WORKDEPT	SALARY
000010	HAAS	A00	52750.00
000030	KWAN	C01	38250.00
000120	O'CONNELL	A00	29250.00
000130	QUINTANA	C01	23800.00
000140	NICHOLLS	C01	28420.00

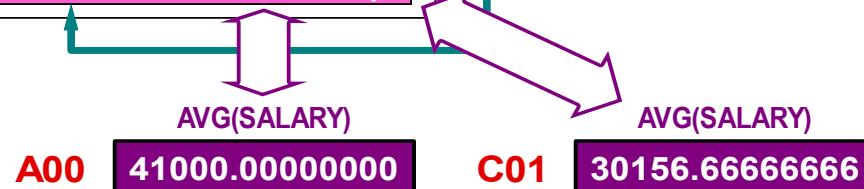
EMPLOYEE

Ahora incluimos la subconsulta indicando explícitamente que me estoy refiriendo al departamento que aparece en la consulta principal. Por ello, tengo que añadir un alias a la tabla EMPLOYEE de la consulta principal y usarlo en la subconsulta para referirme a la fila correspondiente.

```
SELECT EMPNO, LASTNAME, SALARY  
FROM EMPLOYEE E  
WHERE SALARY >
```

```
(SELECT AVG(SALARY)  
FROM EMPLOYEE  
WHERE WORKDEPT =  
E.WORKDEPT)
```

EMPNO	LASTNAME	SALARY
000010	HAAS	52750.00
000030	KWAN	38250.00



Ejemplo: subconsulta correlacionada con >

- ◆ El resultado con la instancia de EMPLOYEE que tenemos en el CV resulta ser:

```
SELECT EMPNO, LASTNAME, SALARY
FROM EMPLOYEE E
WHERE SALARY > (SELECT AVG(SALARY)
                  FROM EMPLOYEE
                  WHERE WORKDEPT = E.WORKDEPT);
```

E.EMPNO	E.LASTNAME	E.SALARY
000010	HAAS	152750
000030	KWAN	98250
000060	STERN	72250
000070	PULASKI	96170
000090	HENDERSON	89750
000160	PIANKA	62250
000200	BROWN	57740
000210	JONES	68270

Pages: 1 1 of 1 Rows: 8

Ejemplo: subconsulta correlacionada con EXISTS

- ◆ En este otro ejemplo usaremos el operador **EXISTS** para la correlación.
- ◆ Necesitamos saber los **departamentos que no tienen empleados**.
- ◆ Esto se puede reformular como sigue: determinar por cada departamento si *no podemos encontrar* ningún empleado *que trabaje en él*.
- ◆ "*no podemos encontrar*" lo entendemos como **NOT EXISTS**.
- ◆ "*que trabaje en él*." es precisamente la condición de correspondencia.
- ◆ Así, formulamos la consulta en la siguiente página con una instancia pequeña de la tabla **EMPLOYEE** para comprobar de un vistazo los resultados.

Ejemplo: subconsulta correlacionada con EXISTS

EMPNO	LASTNAME	WORKDEPT	SALARY
000010	HAAS	A00	52750.00
000030	KWAN	C01	38250.00
000120	O'CONNELL	A00	29250.00
000130	QUINTANA	C01	23800.00
000140	NICHOLLS	C01	28420.00
000400	WILSON	null	25400.00

EMPLOYEE

DEPTNO	DEPTNAME	MGRNO
A00	SPIFFY COMPUTER SERVICE DIV.	000010
C01	INFORMATION CENTER	000030
D01	DEVELOPMENT CENTER	-

DEPARTMENT

¿Qué departamentos no tienen empleados?

```
SELECT DEPTNO, DEPTNAME
  FROM DEPARTMENT D
 WHERE NOT EXISTS
    (SELECT *
      FROM EMPLOYEE
     WHERE
       WORKDEPT = D.DEPTNO)
```

DEPTNO DEPTNAME
D01 DEVELOPMENT CENTER

RESULTADO

Ejemplo: subconsulta correlacionada con EXISTS

- ◆ El resultado con la instancia de EMPLOYEE que tenemos en el CV resulta ser:

```
SELECT DEPTNO, DEPTNAME
FROM DEPARTMENT D
WHERE NOT EXISTS (SELECT *
                   FROM EMPLOYEE
                   WHERE WORKDEPT = D.DEPTNO);
```

D.DEPTNO	D.DEPTNAME
D01	DEVELOPMENT CENTER
F22	BRANCH OFFICE F2
G22	BRANCH OFFICE G2

Vistas de la base de datos

- ◆ Una vista es simplemente una consulta a la que se le proporciona un nombre y se almacena en la base de datos.
- ◆ La sintaxis es:

```
CREATE VIEW Nombre AS Consulta;
```

- ◆ El nombre también puede incluir opcionalmente los nombres de las columnas resultado de la consulta. Por ejemplo:

```
CREATE VIEW JEFES(Apellido, Nombre) AS  
SELECT LASTNAME, FIRSTNAME  
FROM EMPLOYEE JOIN DEPARTMENT  
ON WORKDEPT = DEPTNO AND EMPNO = MGRNO;
```

- ◆ Para borrar una vista usa **DROP VIEW Nombre**;
Por ejemplo: **DROP VIEW JEFES;**

Vistas de la base de datos

- ◆ Al igual que las consultas, las vistas se resuelven al ejecutarse.
- ◆ Se pueden usar en cualquier consulta **SELECT** allí donde pueda aparecer una tabla. Por ejemplo:

```
SELECT * FROM JEFES WHERE Apellido LIKE 'S%';
```

- ◆ Observa que solo se puede hacer referencia a las columnas resultado de la vista (en este caso, a **Apellido** o a **NOMBRE**).

Vistas de la base de datos

- ◆ En el caso de no proporcionar nombre a las columnas, el sistema procede a darles nombre igual que cuando se ejecuta una consulta. En nuestro ejemplo:

```
DES> SELECT LASTNAME, FIRSTNAME  
  FROM EMPLOYEE JOIN DEPARTMENT  
    ON WORKDEPT = DEPTNO AND EMPNO = MGRNO;
```

```
answer(LASTNAME:varchar(15),FIRSTNAME:varchar(12)) ->  
{  
  answer('HAAS','CHRISTINE'),  
  answer('THOMPSON','MICHAEL'),
```

- ◆ Crea la vista sin nombres de columna (borra antes la otra):

```
CREATE VIEW JEFES AS  
SELECT LASTNAME, FIRSTNAME  
  FROM EMPLOYEE JOIN DEPARTMENT  
    ON WORKDEPT = DEPTNO AND EMPNO = MGRNO;
```



Vistas locales

- ◆ Una vista local es similar a una vista, pero su existencia y ámbito se limitan a la consulta para la que se define.
- ◆ Se pueden crear varias vistas locales para una misma consulta.
La sintaxis es:

```
WITH Vista1 AS (Consulta1), ...
      Vistan AS (Consultan)
      Consulta;
```

Definiciones de
vistas locales
separadas por
comas.

Consulta
principal.

- ◆ En esta consulta (que empieza por **WITH** y termina en el punto y coma) se definen las vistas (locales) **Vista₁** hasta **Vista_n**. Estas vistas solo se pueden usar dentro de la consulta principal **Consulta**.
- ◆ Cada **Vista_i** es un nombre y deben llevar entre paréntesis el nombre de sus columnas (parecido a la definición de una tabla, pero sin tipos ni restricciones).
- ◆ Fíjate que cada una de las consultas **Consulta_i** aparece encerrada entre paréntesis.

Consulta con vista local. Ejemplo I

- ◆ Vamos a determinar los **empleados que ganan más que la media de salarios de sus respectivos departamentos**.
- ◆ Primero se define la vista local SALMEANS con los salarios medios de cada departamento, y después se reúne en la consulta principal con la tabla EMPLOYEE:

```
WITH SALMEANS(WORKDEPT, MEAN) AS (
    SELECT WORKDEPT, AVG(SALARY)
    FROM EMPLOYEE
    GROUP BY WORKDEPT
)
SELECT EMPNO, LASTNAME
FROM EMPLOYEE NATURAL JOIN SALMEANS
WHERE SALARY > MEAN;
```

Fíjate que aquí no hay punto y coma.

Tampoco la coma que separa cada una de las vistas locales (porque aquí solo se define una).

- ◆ Esta consulta calcula lo mismo que la de la página 130.

Vistas locales

- ◆ A diferencia de las vistas normales, una vista local pierde su existencia fuera de la consulta en la que se define.
 - Prueba a ejecutar `SELECT * FROM SALMEANS;`
 - ¿Qué ocurre?
- ◆ Las vistas locales también se conocen como *factorización de consultas* o *expresiones de tablas comunes* (CTE – Common Table Expressions).
- ◆ Las vistas locales tienen como objeto:
 - Mejorar la legibilidad.
 - Escribir vistas cuando no tenemos permisos administrativos (el sistema nos lo impide).
 - Especificar consultas recursivas.
- ◆ A continuación escribiremos un ejemplo con varias vistas locales que intentan dividir un problema para hacerlo más abordable (el típico "divide y vencerás").

Consulta con varias vistas locales. Ejemplo II

- ◆ Supongamos una cadena de tiendas y sus ventas en la base de datos definida por:

`STORES(NAME, ADDRESS, MANAGER)`



`SALES(NAME, ITEM, QTY)`

- ◆ La tabla `STORES` guarda el nombre de cada tienda (`NAME`), su dirección (`ADDRESS`) y su director (`MANAGER`).
- ◆ La tabla `SALES` guarda el número (`QTY`) de productos (`ITEM`) vendidos por la tienda (`NAME`).
- ◆ El objetivo es conseguir un listado con los nombres de las tiendas que han vendido más que la media total de ventas.
- ◆ Crea las tablas, inserta unas filas, y ejecuta la siguiente consulta.

Consulta con varias vistas locales. Ejemplo II

WITH

```
SUM_SALES(ALL_SALES) AS  
  ( SELECT SUM(QTY) FROM SALES ),  
NUMBER_STORES(NBR_STORES) AS  
  ( SELECT COUNT(*) FROM STORES ),  
SALES_BY_STORE(NAME, STORE_SALES ) AS  
  ( SELECT NAME, SUM(QTY)  
    FROM STORES NATURAL JOIN SALES  
   GROUP BY NAME )
```

SELECT

```
  NAME
```

FROM

```
  STORES NATURAL JOIN SALES_BY_STORE,  
SUM_SALES, NUMBER_STORES
```

WHERE

```
  STORE_SALES > (ALL_SALES / NBR_STORES);
```

Consulta con varias vistas locales. Ejemplo II

- ◆ ¿Has entendido bien cómo funciona esta consulta?
 - La vista local `SUM_SALES` calcula solo una fila con una columna: la suma de todas las ventas.
 - La vista local `NUMBER_STORES` también calcula solo una fila con una columna: el número total de tiendas.
 - La vista local `SALES_BY_STORE` devuelve una fila por cada tienda, con su nombre y su total de ventas.
 - La consulta principal reúne las tiendas con el cálculo de sus ventas (con `NATURAL JOIN`) y las ventas totales (`SUM_SALES`) y número de tiendas (`NUMBER_STORES`).
 - Así, en la relación que se construye en el `FROM` solo debe haber una fila por cada tienda, y sus columnas serán las de `STORES` (`NAME`, `ADDRESS`, `MANAGER`), `SALES_BY_STORES` (`NAME`, `STORE_SALES`), `SUM_SALES` (`ALL_SALES`) y `NUMBER_STORES` (`NBR_STORES`).
 - Es por esto que se pueden usar estas columnas en la condición del `WHERE`, y funciona precisamente porque cada una de las vistas locales que se usa en esta condición devuelve solo una fila.
 - Compruébalo ejecutando separadamente las consultas de cada vista local.

Vistas locales: particularidades en DES

- ◆ Las vistas locales en DES deben tener un nombre distinto a cualquier otra relación definida globalmente (tabla o vista) para comportarse según el estándar.
- ◆ ¿Es esto una limitación con respecto a otros sistemas?
 - No, es una característica que lo distingue.
- ◆ Reusar un nombre de relación en una vista local en DES significa recargar el significado de la relación existente con el resultado de la vista local.
- ◆ Esto se explicará en el apartado "Consultas hipotéticas" en la página 201.

Consultas recursivas

- ◆ Una aplicación muy interesante de las vistas locales son las consultas recursivas.
- ◆ SQL es un lenguaje de programación declarativo (no algorítmico como Python, C++...) que no es Turing-completo.
- ◆ Alan Turing especificó los requisitos para que un lenguaje pudiera expresar cualquier algoritmo (tesis Church-Turing) y resulta que SQL, a pesar de sus grandes ventajas, no podía.
- ◆ Sin embargo, añadir la recursión a SQL lo convierte en Turing-completo, y esto abre un nuevo abanico de posibilidades que no eran posibles sin ella.
- ◆ Pero... ¿qué es una consulta recursiva?



Alan Turing. Quizás hayas visto la película *Descifrando Enigma* 147

Consultas recursivas

- ◆ Una consulta recursiva es algo que no está muy bien visto en lingüística, en particular en los diccionarios (*lo definido no debe formar parte de la definición*), porque es una consulta que depende de sí misma; es decir, de su definición.
- ◆ Sin embargo, en matemáticas es una noción habitual que aparece por ejemplo en la inducción, lo que nos lleva a examinar el primer ejemplo sencillo de consulta recursiva:
 - a) Sabemos que el 0 es un número natural.
 - b) Además, si N es un número natural, también lo será $N+1$.
- ◆ Hagamos entonces un generador de números teniendo en cuenta estos dos casos a) y b).

Incluye todos los ejemplos y ejercicios en el **nuevo archivo ejercicio07.sql**

Consultas recursivas

- ◆ Para el caso a):

```
SELECT 0 FROM dual;
```

\$a0
0
Pages: 1 1 of 1 Rows: 1

- ◆ Aquí, `dual` es una tabla predefinida en varios sistemas y que tiene una sola fila (y en principio nos da igual qué columnas tenga). Se usa simplemente para calcular expresiones, como la raíz cuadrada, o saber qué hora es:

```
SELECT sqrt(2) FROM dual;
```

\$a0
1.4142135623730951
Pages: 1 1 of 1 Rows: 1

```
SELECT current_time FROM dual;
```

(La expresión puede ser tan complicada como necesites)

\$a0
13:06:46
Pages: 1 1 of 1 Rows: 1

- ◆ En el caso a) calculamos una expresión trivial: la consulta devuelve simplemente la constante 0, lo cual no hace nada particularmente interesante, salvo devolvernos el primer número natural que conocemos (el 0). Este es el caso *base*.
- ◆ Nos resta añadir el caso b) (que se conoce también como caso *recursivo* o *inductivo*).

Consultas recursivas

- ◆ Para el caso b) podríamos formular algo como:

```
SELECT N+1 FROM ¿qué pongo aquí? ;
```

- ◆ N sería el número natural que conocemos y que tendríamos que coger de la columna de una tabla que contenga esos números naturales, pero ¿de dónde lo obtenemos? ¿Qué tabla o vista podemos poner en la interrogación? dual no nos sirve porque no tiene esa columna N...
- ◆ Usemos entonces una vista local para el caso a):

```
WITH naturales(N) AS (
    SELECT 0 FROM dual )
    SELECT * FROM naturales;
```



naturales.N
0

- ◆ La vista local tiene ahora una columna N que va a representar a los números naturales, de momento solo el 0.

Consultas recursivas

- ◆ Para incluir el caso b) en esta consulta podríamos simplemente añadirlo como la unión con el caso a) :

```
WITH naturales(N) AS (
    SELECT 0 FROM dual
    UNION
    SELECT N+1 FROM naturales
)
SELECT * FROM naturales;
```

naturales.N
0
1
2
3
4
5

- ◆ Al usar la unión, sabemos que en la vista local **naturales** vamos a tener al menos el número **0** (porque lo proporciona la primera **SELECT**). Después el caso b) calcula **N+1** a partir de la columna **N** de la vista **naturales**. Como seguro que al menos tiene el **0**, pues también devolverá el **0+1=1**. Si devuelve el **1**, también va a devolver el **2 (1+1)**, y así sucesivamente...
 - ◆ ¿Qué ocurre si ejecutamos esta consulta? El sistema se queda colgado generando números y números sin parar (ni siquiera los llega a mostrar).
 - ◆ ¿Qué hacemos si la hemos ejecutado sin querer y el sistema se cuelga? → Pulsamos el botón **Restart** de la consola.

Consultas recursivas

- ◆ ¿Podemos controlar la terminación de esta consulta en lugar de tener que echar abajo el sistema? Sí, hay dos métodos:

- Usar la cláusula **TOP** para limitar el número de filas a calcular; e.g., 10:

```
WITH naturales(N) AS (
    SELECT 0 FROM dual
    UNION
    SELECT N+1 FROM naturales )
SELECT TOP 10 N FROM naturales;
```

Dependiendo del sistema, se usan distintas notaciones en lugar de **TOP**, como **LIMIT** (en MySQL) o **FETCH FIRST N ROWS ONLY** (el estándar), ambos al final de la consulta en lugar de en la lista de proyección (DES también admite la expresión estándar, pero es más larga de escribir...)

- Limitar en el caso recursivo hasta dónde queremos generar usando la condición **WHERE**:

```
WITH naturales(N) AS (
    SELECT 0 FROM dual
    UNION
    SELECT N+1 FROM naturales WHERE N < 9 )
SELECT N FROM naturales;
```

- ◆ En ambos casos se generan solo los 10 primeros números naturales. La ventaja del primer caso es que no tienes que saber que el décimo número es el 9.

Consultas recursivas

- ◆ Cláusulas para limitar el número de respuestas (en consultas recursivas y no recursivas):

- MS SQL Server, MS Access, DES:

```
SELECT TOP Número Expresiones  
FROM Relación  
WHERE Condición
```

- MySQL, PostgreSQL, DES:

```
SELECT Expresiones  
FROM Relación  
WHERE Condición  
LIMIT Número
```

- Oracle 11g, LearnSQL (sistemas en los laboratorios):

```
SELECT Expresiones  
FROM Relación  
WHERE Condición  
WHERE ROWNUM <= Número
```

- Oracle 12 y posteriores, IBM DB2, DES

```
SELECT Expresiones  
FROM Relación  
WHERE Condición  
FETCH FIRST Número ROWS ONLY; -- Estándar
```

Consultas recursivas: cómo funcionan

- ◆ Siguiendo este ejemplo de los números naturales, en un primer paso el sistema añade las tuplas del caso base a la relación **naturales**. Así, el resultado de naturales contendrá una sola fila con el valor 0 (la tupla (0)):

naturales(N) = {(0)}

- ◆ En el siguiente paso, aplica el caso recursivo: **SELECT N+1 FROM naturales;**
- ◆ Como **naturales** contiene solo la tupla (0), al ejecutar esta consulta recursiva se obtiene {(1)} (resultado de calcular N+1, es decir, 0+1), que se añade a la relación **naturales** porque se hace una unión (**UNION**), y queda:

naturales(N) = {(0)} UNION {(0), (1)} = {(0),(1)}

- ◆ En el siguiente paso, se aplica otra vez el caso recursivo, pero como ahora la relación **naturales** contiene dos tuplas, se calculan dos: {(1),(2)}, respectivamente de (0+1) y (1+1). De nuevo se hace la unión con lo que contenía antes:

naturales(N) = {(0),(1)} UNION {(1),(2)} = {(0),(1),(2)}

- ◆ En el siguiente paso:
- ◆ **naturales(N) = {(0),(1),(2)} UNION {(1),(2),(3)} = {(0),(1),(2),(3)}**
- ◆ Y así sucesivamente hasta que se alcance el número máximo de tuplas requerido.
- ◆ Date cuenta que si se usase **UNION ALL** se repetirían los números, de ahí que pueda ser un problema para el rendimiento (en cada paso se repiten las filas del paso anterior y el resultado crece como el trigo en el tablero de ajedrez de la leyenda).



Consultas recursivas: vistas

- ◆ Si queremos almacenar una consulta recursiva como vista normal, es posible hacerlo:

```
CREATE VIEW números_naturales AS  
  WITH naturales(N) AS (  
    SELECT 0 FROM dual  
    UNION  
    SELECT N+1 FROM naturales )  
  SELECT N FROM naturales;
```

Fíjate que debemos dar un nombre a la vista normal (números_naturales) que debe ser diferente del nombre que aparece en la vista local (naturales).

- ◆ La ventaja de crear esta definición de vista normal es que ahora podemos hacer referencia a ella y pedir los números naturales que necesitemos en cada momento. Por ejemplo:

```
SELECT TOP 5 N FROM números_naturales;
```

El operador **MOD** calcula el resto de la división entera entre dos números. Los números pares, al dividirlos por dos, queda resto 0.

- ◆ O incluso pedir los números que sean pares con:

```
SELECT TOP 20 N FROM números_naturales WHERE N MOD 2 = 0;
```

Consultas recursivas: particularidades

- ◆ Cada sistema de bases de datos tiene sus particularidades. Por ejemplo, algunos requieren usar la palabra **RECURSIVE** para las vistas locales que sean recursivas, como PostgreSQL:

```
WITH  
  RECURSIVE naturales(N) AS (  
    SELECT 0 FROM dual  
    UNION  
    SELECT N+1 FROM naturales WHERE N < 9 )  
  SELECT N FROM naturales;
```

- ◆ La mayoría de sistemas SQL también requieren conservar los duplicados en la consulta de unión (exigen escribir **UNION ALL**) como en IBM DB2 (lo cual puede ser un problema desde el punto de vista del rendimiento e incluso de terminación):

```
WITH  
  naturales(N) AS (  
    SELECT 0 FROM dual  
    UNION ALL  
    SELECT N+1 FROM naturales WHERE N < 9 )  
  SELECT N FROM naturales;
```

Consultas recursivas: particularidades

- ◆ DES (a partir de la versión 1.6.1, Nov. 2008) y PostgreSQL (a partir de la versión 9.3, Sept. 2013) permiten una formulación más concisa de las vistas recursivas evitando el uso de `WITH`:

```
CREATE VIEW naturales(N) AS
    SELECT 0 FROM dual
    UNION
    SELECT N+1 FROM naturales;
```

Compáralo con lo que especifica el estándar, que exige un identificador adicional:

```
CREATE VIEW números_naturales AS
    WITH naturales(N) AS (
        SELECT 0 FROM dual
        UNION
        SELECT N+1 FROM naturales )
    SELECT N FROM naturales;
```

- ◆ En el caso de PostgreSQL hay que incluir `RECURSIVE`:

```
CREATE RECURSIVE VIEW naturales(N) AS ...
```

- ◆ Observa finalmente que en DES la definición de vista recursiva exige la especificación del **esquema completo** (incluyendo los nombres de atributos). En este ejemplo: `naturales(N)`

Consultas recursivas: particularidades

- ◆ Los duplicados en DES provienen no solo de los valores repetidos en las tablas, sino también del caso recursivo. Por ejemplo:

```
WITH v(N) AS (
    SELECT 1 ←
    UNION ALL
    SELECT 1 ←
    UNION ALL
    UNION ALL
    SELECT 1 FROM v) ←
SELECT N FROM v;
```

answer(v.N:int) ->
{
 answer(1),
 answer(1),
 answer(1),
 answer(1)
}
Info: 4 tuples computed.

- ◆ En el caso recursivo se obtendrán tantos valores diferentes como caminos diferentes se hayan usado para determinar las tuplas respuesta.
- ◆ Si quieres eliminar estas repeticiones del caso recursivo puedes usar **DISTINCT**:

```
WITH v(N) AS (
    SELECT 1 ←
    UNION ALL
    SELECT 1 ←
    UNION ALL
    UNION ALL
    SELECT DISTINCT 1 FROM v) ←
SELECT N FROM v;
```

answer(nat.x:int) ->
{
 answer(1),
 answer(1),
 answer(1)
}
Info: 3 tuples computed.

Consultas recursivas: particularidades

- ◆ Los duplicados en PostgreSQL y la mayoría de SGBD pueden provocar no terminación. El mismo ejemplo anterior es no terminante:

```
WITH v(N) AS (
    SELECT 1
    UNION ALL
    SELECT N FROM v)
SELECT N FROM v;
```

- ◆ En este caso puedes usar la cláusula **LIMIT**: al final de la consulta par

```
WITH v(N) AS (
    SELECT 1
    UNION ALL
    SELECT N FROM v)
SELECT N FROM v;
```

- ◆ DES termina en este ejemplo porque distingue los casos base como fuente de los duplicados.

Consultas recursivas: ejercicio R1

- ◆ Escribe una consulta recursiva que calcule solo los primeros 10 números impares.
- ◆ Crea una vista normal (denominada `números_impares`) con `CREATE VIEW` a partir de la consulta anterior y sin limitar el número de resultados.
- ◆ Escribe una consulta `SELECT` que devuelva los 15 primeros números impares a partir de la vista normal `números_impares`.

Consultas recursivas: ejemplo

- ◆ Tenemos una lista de títulos de canciones y su número de copias vendidas de todos los tiempos en la tabla:
`hits(theme string, copies int)`
- ◆ Queremos escribir una consulta SQL que asigne la posición de cada canción por su número de ventas (la que tenga mayor número de ventas tendrá posición 1, la que le sigue en ventas la posición 2, y así sucesivamente); es decir, un *ranking*.
- ◆ Una idea para resolverlo es "pegar" el ranking (un número a partir del 1) a cada canción, que lleva asociada un número de copias.
- ◆ El primer paso es poner un 1 a todas las canciones (todas podrían ser el *top one* de la lista de éxitos hasta que averigüemos cuál es en realidad). Denominaremos `rec` a esta relación. Este es el caso base:

ranking	copies	theme
1	50	White Christmas
1	31	In the Summertime
1	30	Silent Night
1	25	My Heart will Go On
1	25	Rock Around the Clock
1	20	I Will Always Love You
1	20	It's Now or Never
1	20	We Are the World
1	19	If I Didn't Care

Puedes descargar el fichero `hits.sql` del CV.

Consultas recursivas: ejemplo

- ◆ Después ponemos un número consecutivo (el 2 en este siguiente paso) a todas las canciones cuyo número de copias sea menor que las que hubiese en `rec`. Así, a la que tuviese el mayor número de copias no le pondríamos un 2. Este es el caso recursivo.
- ◆ Los pasos sucesivos considerarían el 3, 4... así hasta que no hubiese más posibilidades. Esto se ilustra en la siguiente figura:

ranking	copies	theme
1	50	White Christmas
1	31	In the Summertime
1	30	Silent Night
1	25	My Heart will Go On
1	25	Rock Around the Clock
1	20	I Will Always Love You
1	20	It's Now or Never
1	20	We Are the World
1	19	If I Didn't Care

Caso base

2	31	In the Summertime
2	30	Silent Night
2	25	My Heart will Go On
2	25	Rock Around the Clock
2	20	I Will Always Love You
2	20	It's Now or Never
2	20	We Are the World
2	19	If I Didn't Care

Primer paso del caso recursivo

3	30	Silent Night
3	25	My Heart will Go On
3	25	Rock Around the Clock
3	20	I Will Always Love You
3	20	It's Now or Never
3	20	We Are the World
3	19	If I Didn't Care

Segundo paso del caso recursivo

4	25	My Heart will Go On
4	25	Rock Around the Clock
4	20	I Will Always Love You
4	20	It's Now or Never
4	20	We Are the World
4	19	If I Didn't Care

Tercer paso del caso recursivo

...

- ◆ Piensa cuáles serían los siguientes pasos. El último sería:

Consultas recursivas: ejemplo

- ◆ La consulta recursiva que genera esta salida se puede formular con la siguiente consulta, que incluye la vista local a la que llamamos `rec`:

WITH

```
rec(ranking, copies, theme) AS (
    SELECT 1, copies, theme FROM hits
    UNION
    SELECT ranking+1, hits.copies, hits.theme
    FROM hits, rec
    WHERE hits.copies < rec.copies )
```

```
SELECT *
FROM rec;
```

Caso base

Caso recursivo

Consulta principal

- ◆ Pruébala. Quizás quieras añadir una cláusula `ORDER BY` en la consulta principal para ordenar los resultados.

Consultas recursivas: ejemplo

- ◆ Ahora solo nos queda filtrar cada grupo (caracterizado por el número de ranking) y quedarnos con la fila que tenga el mayor número de copias. Es decir, devolver solo las filas encuadradas en verde en la figura:



- ◆ Fíjate que en algunos grupos podemos tener más de una fila con el número máximo de descargas. En estos casos queremos ser equitativos y darles la misma posición en el ranking.

Consultas recursivas: ejemplo

- ◆ Añadamos la agrupación y el cálculo del máximo:

WITH

```
rec(ranking, copies, theme) AS (
    SELECT 1, copies, theme FROM hits
    UNION
    SELECT ranking+1, hits.copies, hits.theme
    FROM hits, rec
    WHERE hits.copies < rec.copies )
SELECT ranking, MAX(copies)
FROM rec
GROUP BY ranking;
```



ranking	copies
1	50
2	31
3	30
4	25
5	20
6	19

- ◆ Bueno, no está mal, pero nos falta añadir el nombre de la canción. No podemos añadirla a la salida porque es una columna por la que no se agrupa (tampoco deberíamos hacerlo porque el grupo debe depender solo del valor del ranking).
- ◆ ¿Se te ocurre cómo hacerlo?

Consultas recursivas: ejemplo

- ◆ Una forma es usar otra vista local (digamos `ranks`) para obtener lo que calculamos con la principal de antes, y después reunirla en la nueva principal con `hits` para devolver también el título de la canción:

WITH

```
rec(ranking, copies) AS (
    SELECT 1, copies FROM hits
    UNION
    SELECT ranking+1, hits.copies
    FROM hits, rec
    WHERE hits.copies < rec.copies ),
ranks(ranking, copies) AS (
    SELECT ranking, MAX(copies)
    FROM rec
    GROUP BY ranking )
SELECT ranking, copies, theme
FROM hits NATURAL JOIN ranks
ORDER BY ranking, theme;
```

Fíjate que hemos quitado la columna `theme` de la vista local recursiva: simplemente no es necesaria aquí.



ranking	copies	theme
1	50	White Christmas
2	31	In the Summertime
3	30	Silent Night
4	25	My Heart will Go On
4	25	Rock Around the Clock
5	20	I Will Always Love You
5	20	It's Now or Never
5	20	We Are the World
6	19	If I Didn't Care

Consultas recursivas: ejercicio R2

- ◆ Piensa al menos otra forma de añadir el título de la canción sin usar una vista local como `ranks`. Añádelo como siempre al archivo de ejercicios.
- ◆ ¿Cuál es la versión que te parece más intuitiva, elegante o fácil de escribir? Responde en el archivo de ejercicios como un comentario (recuerda: con `--` al principio de la línea).
- ◆ ¿Se puede resolver este ejercicio sin usar la recursión?
(Acertijo)

Consultas recursivas: ejercicio R3

- ◆ Obtén los 10 mejores empleados (de la tabla EMPLOYEE) según el valor de sus incentivos (campo BONUS).
- ◆ El mejor debe recibir un *ranking* de 1 y el 10º (menos bueno) un 10.
- ◆ El resultado debería ser:

RANKS.RANKING	BONUS	NAME
1	4220	CHRISTINE HAAS
2	3720	VINCENZO LUCCHESSI
3	3300	MICHAEL THOMPSON
4	3214	JOHN GEYER
5	3060	SALLY KWAN
6	2893	EVA PULASKI
7	2580	IRVING STERN
8	2387	JENNIFER LUTZ
9	2380	EILEEN HENDERSON
10	2340	SEAN O'CONNELL

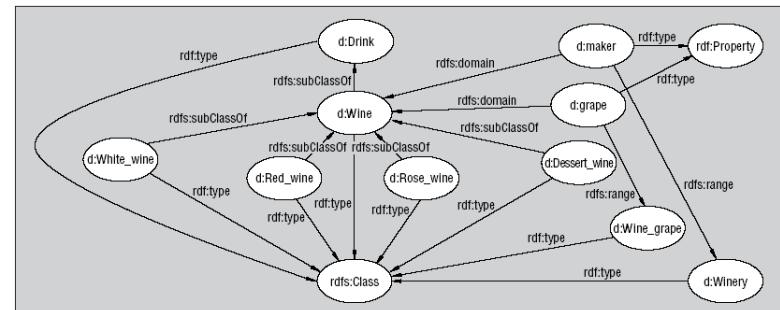
Consultas recursivas: el cierre transitivo

- ◆ ¿Qué más se puede hacer con las consultas recursivas?
- ◆ Algo muy interesante: calcular el cierre transitivo.
- ◆ ¿Qué es un cierre transitivo? Quizás os acordéis de las relaciones simétricas, reflexivas y... transitivas.
 - Puede que también recuerdes la película *Lo mejor que le puede pasar a un crudasán* y de unas líneas de diálogo en las que se decía algo así como: "Los amigos de mis amigos son también mis amigos", a lo que Pablo Carbonell respondía: "No se crea que las relaciones transitivas se cumplen en todos los casos".
 - Recuerda también que las relaciones son binarias: están definidas por pares de elementos (como dos amigos).
- ◆ En una relación transitiva, si un elemento "a" está relacionado con otro "b", y "b" con "c", entonces "a" también lo estará con "c":
$$(a,b) \text{ y } (b,c) \Rightarrow (a,c)$$
- ◆ El cierre transitivo es simplemente determinar todas estas relaciones. En el ejemplo anterior solo se deduce una, pero puede haber más:
$$(a,b), (b,c) \text{ y } (c,d) \Rightarrow (a,c), (a,d), (b,d)$$



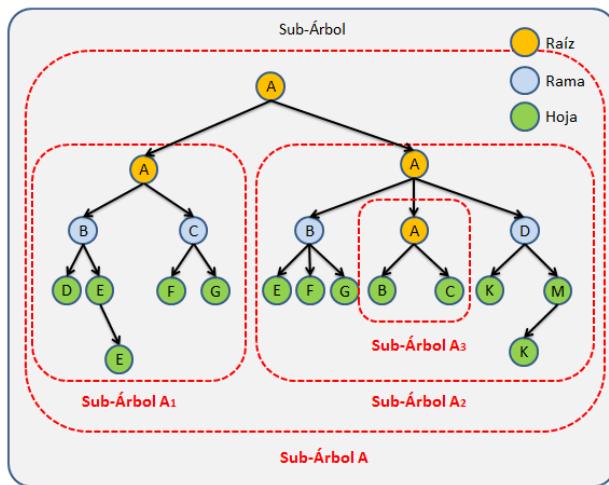
Consultas recursivas: aplicaciones del cierre

- ◆ ¿Qué se puede hacer con este concepto de cierre transitivo en las consultas recursivas?
- ◆ Por ejemplo, en la base de datos `buses.sql` podemos determinar de dónde a dónde se puede viajar. Con datos de distancia y velocidad entre paradas también se podría determinar el tiempo de viaje entre cualquiera dos paradas (sean de la misma línea o de distintas).
- ◆ En una relación de sinonimia se pueden averiguar todos los sinónimos a partir de una representación binaria: si "coche" es sinónimo de "auto", y "auto" lo es de "carro", entonces "coche" también lo será de "carro".
- ◆ Otras relaciones lingüísticas también se pueden tratar del mismo modo, como la hipernimia y la hiponimia.
- ◆ En un diccionario se podrían detectar referencias circulares si resultase que una palabra hace referencia a otra y en el camino de referencias encontrásemos de nuevo la palabra original.
- ◆ En general el cierre es útil para aplicaciones en las que encontramos árboles o grafos, algo habitual en las representaciones de ontologías. (En la imagen, una ontología de vinos en OWL con formato RDF).

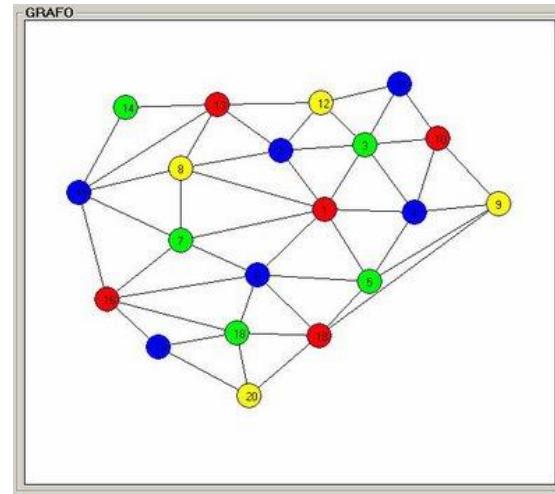


Consultas recursivas: árboles y grafos

- ◆ En general podemos hablar de dos estructuras susceptibles de la aplicación de la recursión en SQL: los árboles y los grafos.



Los árboles relacionan elementos entre sí y generalmente con un sentido que se ve gráficamente en la flecha



Los grafos admiten conexiones más liberales, y pueden tener sentido (flechas) o no.

- ◆ Hay toda una teoría alrededor de los grafos (que no vamos a estudiar aquí, claro), y los árboles se pueden ver como casos particulares de ellos.

Consultas recursivas: árboles

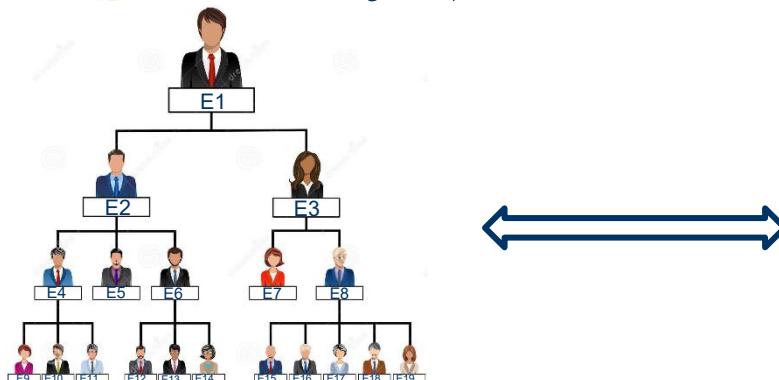
- ◆ Los árboles tienen una estructura jerárquica y dotada de orden; es decir, disfrutan además de la propiedad de antisimetría.
 - Por si no lo recuerdas o no lo sabías, la *antisimetría* viene a decir que si un elemento "a" está relacionado con otro "b", entonces no es cierto que "b" lo esté con "a".
 - La *asimetría*, por otra parte, dice que si "a" está relacionado con "b", entonces es posible (pero no es obligatorio como en la simetría) que "b" lo esté con "a".
- ◆ Por ejemplo, la relación "jefe" puede representar que un empleado es jefe de otro: si Ana es jefa de Pedro, no puede darse el caso de que Pedro lo sea de Ana (a menos que lo asciendan, claro 😊).
- ◆ Los tesauros son también un caso de árbol: organizan taxonómicamente conceptos o palabras.

Descriptor	Descripción	Nº de Fichas
CORTES		0
CORTES, DERECHO DE VOTO		0
PROCURADORES DE DIETAS		0
DIPUTACIÓN DE ARAGÓN		0
DIPUTACIÓN DE CATALUÑA		0
DIPUTACIÓN DE EXTREMADURA		0
DIPUTACIÓN DE VALENCIA		0
DIPUTACIONES		0
DIPUTADOS		0
GENERALITAT DE CATALUNYA		0
GENERALITAT DE VALENCIA		0
OFICIALES DEL REINO	No del rey.	0
OFICIALES DEL REINO, SALARIO	Retribución de los que no lo son del rey.	0

Tesoro del sistema de legislación histórica de España. Ministerio de Cultura

Consultas recursivas: ejemplo con árboles

- ◆ Dada la tabla: `jefes(empleado, jefe*)`, que puedes descargar del CV (`jefes.sql`).
- ◆ Se trata de determinar quién es jefe de quién, tanto directa como indirectamente.
- ◆ La tabla `jefes` contiene realmente una relación binaria entre empleados: la segunda columna (`jefe`) indica quién es el jefe del empleado en la primera columna (`empleado`).
 - Recuerda que el asterisco indica que el campo es opcional (en este caso, el "jefe supremo" -también llamado "gran jefe" 😊 - no tiene jefe).



empleado	jefe
E1	null
E2	E1
E3	E1
E4	E2
E5	E2
E6	E2
E7	E3
E8	E3
E9	E4
E10	E4
E11	E4
...	

Consultas recursivas: ejemplo con árboles

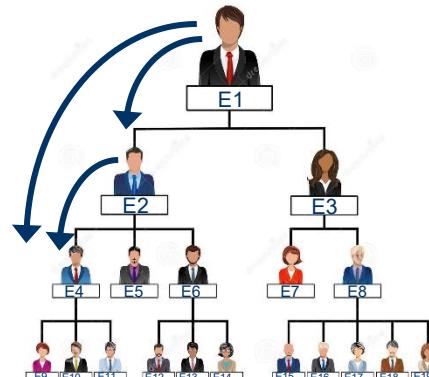
- ◆ La tabla `jefes` contiene la relación "jefe directo de".
- ◆ Vamos a averiguar cuáles son todos los jefes, tanto directos como indirectos. En particular, dado que `E1` es jefe directo de `E2`, y `E2` es jefe directo de `E4`, entonces `E1` también es jefe (indirecto) de `E3`.

```
WITH tiene_jefe(empleado, jefe) AS (
    SELECT empleado, jefe FROM jefes
    UNION
    SELECT tiene_jefe.empleado, jefes.jefe
    FROM tiene_jefe JOIN jefes ON tiene_jefe.jefe = jefes.empleado
    WHERE jefes.jefe IS NOT NULL
)
SELECT * FROM tiene_jefe;
```

Diagrama de un organigrama jerárquico:

- Caso base (jefes directos): El primer `SELECT` en la consulta.
- Caso recursivo (jefes indirectos): El segundo `SELECT` en la consulta.
- Consulta principal: La consulta completa que incluye el `WITH` y el `SELECT *`.

Fíjate que por ejemplo la fila `(E4,E1)` se deduce de esta consulta porque hay una fila `(E2,E1)` y también otra `(E4,E2)` en jefe. Como la vista local recursiva `tiene_jefe` incluye en el caso base estas dos, al aplicarse la `JOIN` entre `tiene_jefe` y `jefes`, la condición `tiene_jefe.jefe = jefes.empleado` tiene éxito para la fila `(E4,E2)` de `tiene_jefe` y la fila `(E2,E1)` de `jefe`, con lo que la consulta devuelve `(E4,E1)` en este caso. El resto de tuplas se generan de forma similar.



empleado	jefe
E1	null
E2	E1
E3	E1
E4	E1
E4	E2
E5	E1
E5	E2
E6	E1
E6	E2
E7	E1
E7	E3
E8	E1
E8	E3
E9	E1
E9	E2
E9	E4
...	

Consultas recursivas: ejemplo con árboles

- ◆ También resulta interesante conocer los niveles que separan a un empleado de sus jefes.
- ◆ Se puede modificar la consulta anterior añadiendo ese nivel: en el caso base será un nivel 1 (jefe directo) y en el recursivo un nivel más que el de la relación `tiene_jefe`:

```
WITH tiene_jefe(empleado, jefe, nivel) AS (
    SELECT empleado, jefe, 1 FROM jefes
    UNION
    SELECT tiene_jefe.empleado, jefes.jefe, tiene_jefe.nivel+1
    FROM tiene_jefe JOIN jefes ON tiene_jefe.jefe = jefes.empleado
    WHERE jefes.jefe IS NOT NULL )
SELECT * FROM tiene_jefe;
```

Extracto del resultado: hay tres niveles entre los empleados que no son jefes y el "jefe supremo".

empleado	jefe	nivel
E14	E1	3
E15	E1	3
E16	E1	3
E17	E1	3
E18	E1	3
E19	E1	3

Consultas recursivas: ejercicio R4

- ◆ Crea una vista normal denominada `jefes_de(jefe, empleado)` (al revés de como lo tenemos puesto en la salida de la `WITH` anterior) definida por la consulta de la página [174](#).
- ◆ Selecciona con una consulta `SELECT` quiénes son los jefes del empleado E5.
- ◆ El resultado debería ser:

jefe	empleado
E2	E5
E1	E5

Pages: 1 1 of 1 Rows: 2

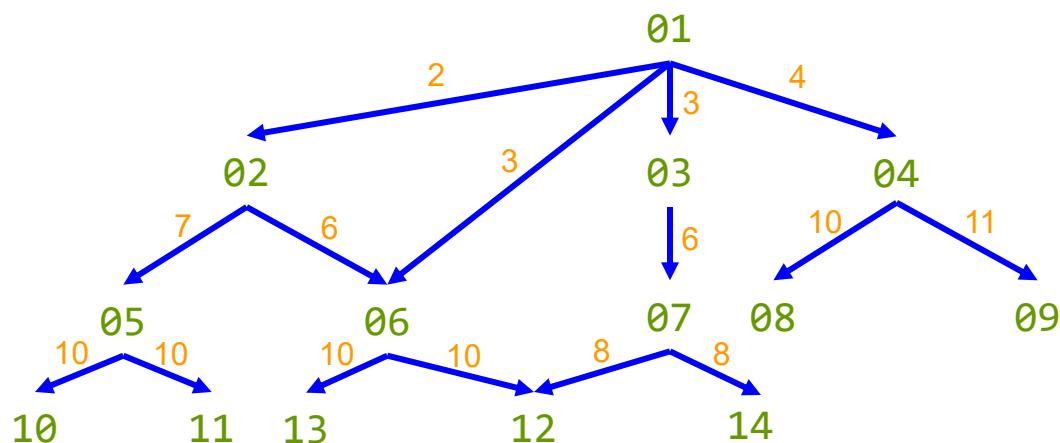
Consultas recursivas: ejemplo

- ◆ Otro ejemplo habitual de consultas recursivas con árboles son las aplicaciones BOM (Bill Of Materials - Lista de materiales).
- ◆ Se parte de una relación que indica de que piezas (o partes) está compuesta otra.
- ◆ Por ejemplo, una bicicleta puede estar compuesta de marco, manillar, ruedas y sillín. A su vez, una rueda puede estar compuesta por una llanta, una cubierta y una cámara. Una llanta puede también estar compuesta por un buje y unos radios. Y así sucesivamente.
- ◆ Consideremos una tabla PARTLIST(PART, SUBPART, QUANTITY), que indica que una parte (PART) tiene a otra como componente (SUBPART) en cierta cantidad (QUANTITY). Por ejemplo, una llanta podría tener 50 radios. En esta tabla se almacenan solo números de referencia (descarga el archivo `partlist.sql`).
- ◆ El objetivo es obtener un listado de la cantidad de partes componente de otra a partir de la pieza de referencia '01' (por ejemplo, este código podría representar la rueda, y solo se listarían componentes de rueda; pero no la bicicleta, que podría tener por ejemplo el código '00').

PART	SUBPART	QUANTITY
00	01	5
00	05	3
01	02	2
01	03	3
01	04	4
01	06	3
02	05	7
02	06	6
...		

Consultas recursivas: ejemplo

- ◆ Para resolver esta consulta, el caso base serían las filas cuyo código de pieza sea '**01**', devolviendo los tres campos de la tabla PARTLIST.
- ◆ El caso recursivo sería la unión de la vista local (que denominaremos RPL - por Recursive Part List) con la propia tabla PARTLIST tal que la subparte de RPL coincida con la parte de PARTLIST.



Calcula a mano la cantidad de piezas de tipo '**06**' necesarias para construir la pieza '**01**': por un lado necesitamos 3 (por la flecha directa entre ellas) y, por otro lado, necesitamos 2 de la '**02**', que a su vez necesita 6 de la '**06**' (esta pieza podrían ser tornillos), es decir, $2 \times 6 = 12$. Esto hace un total de $3 + 12 = 15$ piezas '**06**' para construir la '**01**'.

PART	SUBPART	QUANTITY
00	01	5
00	05	3
01	02	2
01	03	3
01	04	4
01	06	3
02	05	7
02	06	6

Consultas recursivas: ejemplo

- ◆ Razonando así, la consulta se puede escribir:

```
WITH RPL(PART, SUBPART, QUANTITY) AS (
    SELECT PART, SUBPART, QUANTITY
    FROM PARTLIST
    WHERE PART = '01'
```

El caso base es sencillo: solo filtra las filas para la pieza '01'.

UNION ALL

```
SELECT PARENT.PART, CHILD.SUBPART,
       CHILD.QUANTITY * PARENT.QUANTITY
    FROM RPL AS PARENT, PARTLIST AS CHILD
   WHERE PARENT.SUBPART = CHILD.PART )
```

Dos comentarios sobre el caso recursivo:
1) Renombramos respectivamente con PARENT y CHILD las relaciones que representan al arco superior y al inmediatamente inferior. Esto te puede ser útil para escribir otras consultas parecidas (solo tienes que cambiar una vez el nombre de la tabla del FROM). 2) Fíjate que para calcular la cantidad de piezas en el caso recursivo tenemos que multiplicar la cantidad de piezas del padre por las del hijo, como hicimos en el ejemplo de la página anterior.

```
SELECT PART, SUBPART, SUM(QUANTITY) AS QUANTITY
  FROM RPL
 GROUP BY PART, SUBPART;
```

En este caso, UNION ALL es importante: no podemos escribir simplemente UNION. ¿Podrías decir por qué? Piensa que dos filas para la misma pieza y subpieza (calculadas por distintas ramas del árbol) podrían tener el mismo valor para la cantidad. Imagina que el arco directo entre '01' y '06' tuviese cantidad 12 en lugar de 3. Haz la prueba con este otro dato, con ALL y sin ALL.

¿Por qué no devolvemos simplemente PART, SUBPART, QUANTITY en lugar de esta agregación con SUM? Date cuenta que se puede calcular más de una fila en RPL para la misma pieza y subpieza. Por ejemplo: ('01','06',3) (conexión directa), y ('01','06',12) (conexión indirecta pasando por la pieza '02'). Por tanto, hay que sumarlas todas.

Cómo resuelve el sistema este ejemplo

```
SELECT PART, SUBPART, QUANTITY  
FROM PARTLIST  
WHERE PART = '01'
```

PART	SUBPART	QUANTITY
00	01	5
00	05	3
01	02	2
01	03	3
01	04	4
01	06	3
02	05	7
02	06	6
03	07	6
04	08	10
04	09	11
05	10	10
05	11	10
06	12	10
06	13	10
07	12	8
07	14	8

Tabla PARTLIST

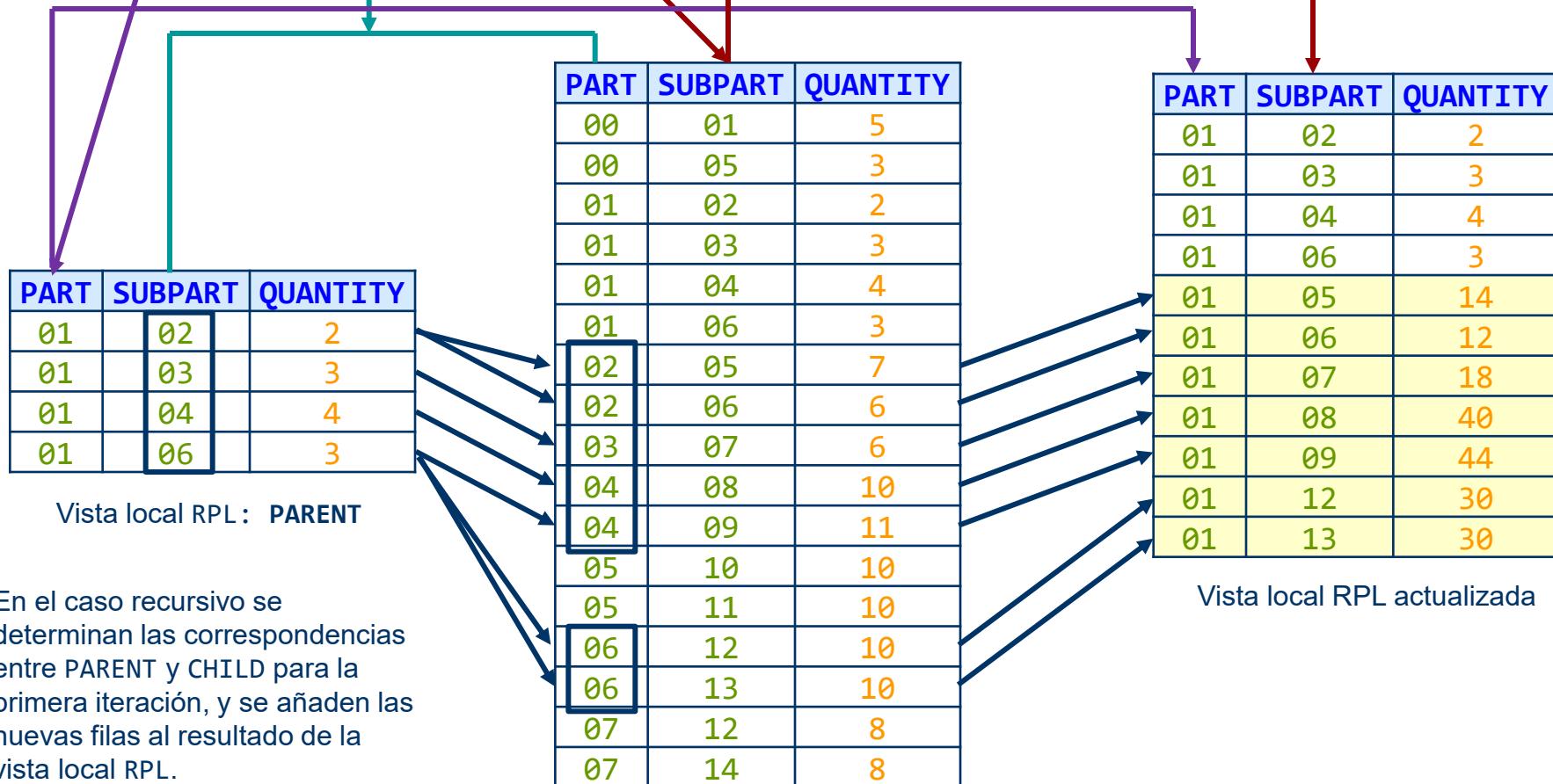
En un primer caso se calcula el caso base (a partir de PARTLIST) y las filas se añaden al resultado de la vista local RPL.

PART	SUBPART	QUANTITY
01	02	2
01	03	3
01	04	4
01	06	3

Vista local RPL

SQL Recursivo. Iteración 1

```
SELECT PARENT.PART, CHILD.SUBPART, CHILD.QUANTITY * PARENT.QUANTITY
FROM RPL AS PARENT, PARTLIST AS CHILD
WHERE PARENT.SUBPART = CHILD.PART )
```



En el caso recursivo se determinan las correspondencias entre PARENT y CHILD para la primera iteración, y se añaden las nuevas filas al resultado de la vista local RPL.

SQL Recursivo. Iteración 2

```
SELECT PARENT.PART, CHILD.SUBPART, CHILD.QUANTITY * PARENT.QUANTITY
FROM RPL AS PARENT, PARTLIST AS CHILD
WHERE PARENT.SUBPART = CHILD.PART )
```

PART	SUBPART	QUANTITY
01	02	2
01	03	3
01	04	4
01	06	3
01	05	14
01	06	12
01	07	18
01	08	40
01	09	44
01	12	30
01	13	30

Vista local RPL: PARENT

En la segunda iteración se aplica de nuevo el caso recursivo a las nuevas filas (las anteriores ya se tuvieron en cuenta en la iteración anterior).

PART	SUBPART	QUANTITY
00	01	5
00	05	3
01	02	2
01	03	3
01	04	4
01	06	3
02	05	7
02	06	6
03	07	6
04	08	10
04	09	11
05	10	10
05	11	10
06	12	10
06	13	10
07	12	8
07	14	8

Tabla PARTLIST: CHILD
Lenguaje SQL

PART	SUBPART	QUANTITY
01	02	2
01	03	3
01	04	4
01	06	3
01	05	14
01	06	12
01	07	18
01	08	40
01	09	44
01	12	30
01	13	30
01	10	140
01	11	140
01	12	120
01	13	120
01	12	144
01	14	144

Vista local RPL actualizada
182

SQL Recursivo. Iteración 3

```
SELECT PARENT.PART, CHILD.SUBPART, CHILD.QUANTITY * PARENT.QUANTITY  
FROM RPL AS PARENT, PARTLIST AS CHILD  
WHERE PARENT.SUBPART = CHILD.PART )
```



PART	SUBPART	QUANTITY
01	02	2
01	03	3
01	04	4
01	06	3
01	05	14
01	06	12
01	07	18
01	08	40
01	09	44
01	12	30
01	13	30
01	10	140
01	11	140
01	12	120
01	13	120
01	12	144
01	14	144

Vista local RPL: PARENT

PART	SUBPART	QUANTITY
00	01	5
00	05	3
01	02	2
01	03	3
01	04	4
01	06	3
02	05	7
02	06	6
03	07	6
04	08	10
04	09	11
05	10	10
05	11	10
06	12	10
06	13	10
07	12	8
07	14	8

Tabla PARTLIST: CHILD
Lenguaje SQL

En la tercera iteración no encontramos más correspondencias (en CHILD no hay ningún valor del 10 al 14 en PART) y acabamos.

PART	SUBPART	QUANTITY
01	02	2
01	03	3
01	04	4
01	06	3
01	05	14
01	06	12
01	07	18
01	08	40
01	09	44
01	12	30
01	13	30
01	10	140
01	11	140
01	12	120
01	13	120
01	12	144
01	14	144

Vista local RPL final 183

SQL Recursivo. Consulta principal

Ya solo queda aplicar la consulta principal:

```
SELECT PART, SUBPART, SUM(QUANTITY) AS QUANTITY  
FROM RPL  
GROUP BY PART, SUBPART;
```

PART	SUBPART	QUANTITY
01	02	2
01	03	3
01	04	4
01	06	3
01	05	14
01	06	12
01	07	18
01	08	40
01	09	44
01	12	30
01	13	30
01	10	140
01	11	140
01	12	120
01	13	120
01	12	144
01	14	144

PART	SUBPART	QUANTITY
01	02	2
01	03	3
01	04	4
01	05	14
01	06	15
01	07	18
01	08	40
01	09	44
01	10	140
01	11	140
01	12	294
01	13	150
01	14	144

Resultado de la consulta principal

Consultas recursivas: ejercicio R5

- ◆ Esto puede que sea muy interesante, pero yo no soy ingeniero en bicicletas. ¿Para qué me sirve esto?
- ◆ Esta estructura de consulta es útil en muchas aplicaciones. Imagina una ontología con una relación binaria de hiponimia. ¿Podrías calcular el número de hipónimos de una palabra dada?
- ◆ Con el esquema anterior no supondría ningún problema. Es más, podríamos obviar las etiquetas de cantidades.
- ◆ Crea una tabla para esta relación de hiponimia y calcula el número de hipónimos para una palabra que elijas.
- ◆ Modifica la consulta para calcular todos los hiperónimos de una palabra.

Consultas recursivas: control de la recursión

- ◆ A veces no es necesario devolver todos los niveles de una jerarquía.
- ◆ Si queremos controlar el número de niveles podemos añadir un campo a la vista local recursiva que vaya contando el número de niveles, y después acotando al límite que necesitemos en el caso recursivo (esto es parecido a la generación de números naturales cuando los limitábamos la propia vista local).
- ◆ La siguiente consulta limita a 2 niveles el ejemplo anterior:

```
WITH RPL(PART, SUBPART, QUANTITY, LEVEL) AS (
  SELECT PART, SUBPART, QUANTITY, 1
  FROM PARTLIST
  WHERE PART = '01'

  UNION ALL

  SELECT PARENT.PART, CHILD.SUBPART,
         CHILD.QUANTITY * PARENT.QUANTITY,
         PARENT.LEVEL + 1
    FROM RPL AS PARENT, PARTLIST AS CHILD
   WHERE PARENT.SUBPART = CHILD.PART AND LEVEL < 2 )

  SELECT PART, SUBPART, SUM(QUANTITY) AS QUANTITY
  FROM RPL
 GROUP BY PART, SUBPART;
```

Fíjate cómo cambian algunas cantidades.
¿Sabrías explicar por qué?

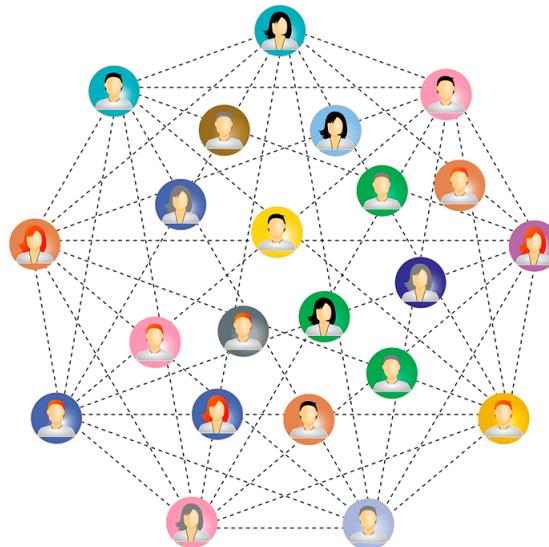


PART	SUBPART	QUANTITY
01	02	2
01	03	3
01	04	4
01	05	14
01	06	15
01	07	18
01	08	40
01	09	44
01	12	30
01	13	30

Pages: 1 1 of 1 Rows: 10

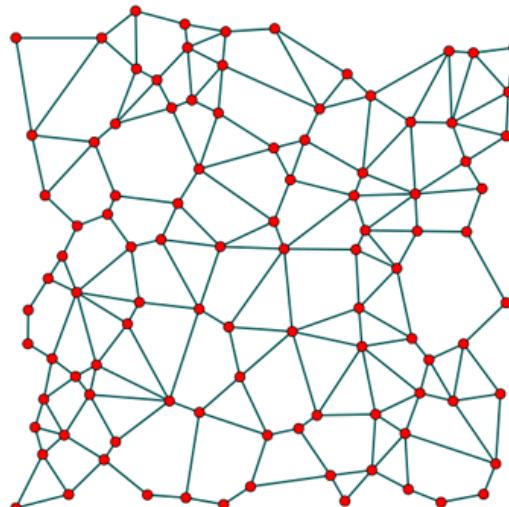
Consultas recursivas: grafos

- ◆ En un grafo, cualquier par de elementos puede estar conectado entre sí.
- ◆ La relación de orden es, a diferencia de los árboles, opcional. Se habla entonces de grafos dirigidos o no dirigidos según exista orden (con flechas) o no (solo con arcos).
- ◆ Los grafos, al admitir una estructura más flexible, pueden incluir ciclos (los árboles no pueden).
- ◆ También son muy adecuados para representar relaciones en las redes sociales: unas personas conocen a (o están en contacto con) otras. (Al final, gracias a los grafos voy a poder llegar a conocerme a mí mismo 😊).



Consultas recursivas: alcanzabilidad en grafos

- ◆ Otro ejemplo paradigmático de la aplicación de las consultas recursivas es la alcanzabilidad en grafos.
- ◆ A partir de una red de nodos interconectados, se trata de determinar los nodos que se pueden alcanzar a partir de otros.
- ◆ La red puede representar un plano del metro, la red de carreteras, una red social o una red semántica, por ejemplo.

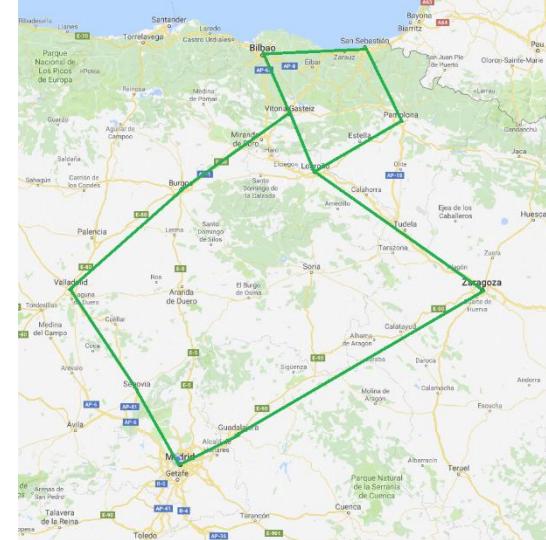


Alcanzabilidad en grafos: ejemplo

- ◆ Consideremos una red de carreteras y la tabla `vías(origen, destino, km, kph)` que determina que hay una vía entre sus dos primeros argumentos separadas por una ciertos kilómetros (km) y en la que se circula a una cierta velocidad media en km/h (kph).
- ◆ Descarga la base de datos `vías.sql` que representa las vías de la figura y ejecuta el archivo.
- ◆ Escribe la consulta de abajo que calcula el cierre transitivo de la relación `vías`.

```
WITH conecta(origen, destino) AS (
    SELECT origen, destino
    FROM vías
    UNION
    SELECT conecta.origen, vías.destino
    FROM conecta JOIN vías ON conecta.destino=vías.origen )
SELECT * FROM conecta;
```

Observa que la estructura de esta consulta es similar a las anteriores (caso base, recursivo y consulta principal), calculando el cierre transitivo de la relación `vías`.



Databases	Answer	PARTLIST
origen	destino	
Bilbao	San Sebastián	
Burgos	Bilbao	
Burgos	San Sebastián	
Burgos	Vitoria	
Guadalajara	Bilbao	
Guadalajara	Logroño	
Guadalajara	Pamplona	
Guadalajara	San Sebastián	
Guadalajara	Vitoria	
Guadalajara	Zaragoza	
Logroño	Bilbao	
Logroño	Pamplona	
Logroño	San Sebastián	
Logroño	Vitoria	
Madrid	Bilbao	
Madrid	Burgos	
Madrid	Guadalajara	
Madrid	Logroño	
Madrid	Pamplona	
Madrid	San Sebastián	

Pages: 1 2 3 Next Last 1 of 3

Rows: 41

Alcanzabilidad en grafos: ejercicio R6

- ◆ ¿Es posible viajar de Madrid a San Sebastián?
- ◆ Crea una vista normal denominada **rutas** con la consulta anterior.
- ◆ Emite una consulta sobre esta vista indicando que el origen es Madrid y el destino San Sebastián.
- ◆ ¿Se puede viajar de San Sebastián a Madrid? Si escribes:
`SELECT * FROM rutas WHERE origen = 'San Sebastián' AND destino = 'Madrid';`
- ◆ El resultado será vacío porque los arcos son dirigidos y no hay camino en ese sentido.
- ◆ Para resolverlo hay que añadir simetría a las conexiones, por ejemplo en la consulta principal de **rutas** con:
`SELECT origen, destino FROM conecta
UNION
SELECT destino, origen FROM conecta`
- ◆ Ahora sí podrías determinar que se puede viajar de San Sebastián a Madrid. Compruébalo.

Alcanzabilidad en grafos: ejercicio R7

- ◆ Además de saber si puedes ir de un sitio a otro, puedes conocer la distancia.
- ◆ Modifica la vista local de alcanzabilidad **conecta** para añadirle un tercer argumento: **km**, que serán los kilómetros entre dos puntos:
 - Para el caso base, este tercer argumento viene directamente de la tabla **vías**.
 - Para el caso recursivo hay que sumar los kilómetros de la tabla **vías** y los de la vista local **conecta**. Date cuenta que la distancia es simplemente una medida acumulativa (lo que llevabas de **conecta** más el nuevo arco de **vías**).

conecta.orig	conecta.desl	conecta.km
Bilbao	Burgos	150
Bilbao	Guadalajara	360
Bilbao	Logroño	100
Bilbao	Madrid	410
Bilbao	Madrid	440
Bilbao	San Sebastiá	80
Bilbao	Segovia	370
Bilbao	Valladolid	270
Bilbao	Vitoria	50
Bilbao	Zaragoza	260
Burgos	Bilbao	150
Burgos	Madrid	290
Burgos	San Sebastiá	230
Burgos	Segovia	220
Burgos	Valladolid	120
Burgos	Vitoria	100
Guadalajara	Bilbao	360
Guadalajara	Logroño	260
Guadalajara	Madrid	50
Guadalajara	Pamplona	340

Pages: 1 2 3 4 5 Next Last
1 of 5 Rows: 96

Alcanzabilidad en grafos: ejercicio R8

- ◆ Calcula también el tiempo: modifica la vista local `conecta` para que el tercer argumento indique las horas (`horas`). Quédate con dos decimales en la consulta principal.
 - Para el caso base, este tercer argumento se calcula como la división entre los kilómetros (`km`) y la velocidad (`kph`), es decir, simplemente km/kph .
 - Para el caso recursivo, hay que sumar el tiempo de la tabla `vías` (que se calcula igual que en el caso base) y el tiempo de la vista local `conecta` (`horas`). Al igual que antes, es una medida acumulativa.
- ◆ Calcula el tiempo mínimo para viajar de Madrid a San Sebastián. Habrás observado que hay más de una ruta y con distintos tiempos. Escribe una consulta con una función de agregación (**MIN**) para quedarte con el tiempo mínimo. →

horas
4.77

origen	destino	horas
Bilbao	Burgos	1.46
Bilbao	Guadalajara	3.58
Bilbao	Logroño	1.14
Bilbao	Madrid	4.03
Bilbao	Madrid	4.2
Bilbao	San Sebastiá	1.07
Bilbao	Segovia	3.62
Bilbao	Valladolid	2.51
Bilbao	Vitoria	0.56
Bilbao	Zaragoza	2.74
Burgos	Bilbao	1.46
Burgos	Madrid	2.74
Burgos	San Sebastiá	2.53
Burgos	Segovia	2.15
Burgos	Valladolid	1.04
Burgos	Vitoria	0.91
Guadalajara	Bilbao	3.58
Guadalajara	Logroño	2.43
Guadalajara	Madrid	0.45
Guadalajara	Pamplona	3.5

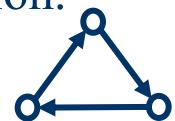
Pages: [1](#) [2](#) [3](#) [4](#) [5](#) [Next](#) [Last](#)
1 of 5 Rows: 96

Riesgos en los grafos: ciclos

- ◆ Los ciclos en un grafo pueden representar un problema de no terminación.
- ◆ Considera la siguiente situación para **vías**, que incluye un ciclo:
- ◆ En ciertos sistemas, como DB2, la siguiente consulta no terminará:

```
WITH conecta(origen, destino) AS (
    SELECT origen, destino
    FROM vías
    UNION ALL
    SELECT conecta.origen, vías.destino
    FROM conecta, vías WHERE conecta.destino = vías.origen )
SELECT * FROM conecta;
```

- ◆ Observa que DB2 exige **UNION ALL**, y en cada iteración irá añadiendo un nuevo arco independientemente de que ya lo haya añadido antes.
- ◆ Para evitar este problema se puede imponer un límite al número de arcos visitados (véase el control de la recursión, pág. [183](#)).
- ◆ También puedes usar otro gestor, como PostgreSQL, que permite usar simplemente **UNION** (además parece ser es el más rápido para este tipo de consultas recursivas).



Consultas recursivas: limitaciones

- ◆ La mayoría de implementaciones imponen las siguientes limitaciones en las consultas recursivas:
 - Solo se acepta recursión lineal (solo una llamada recursiva)
 - No se acepta la recursión mutua.
 - La consulta debe tener al menos un caso base y un caso recursivo unidos por **UNION ALL**
 - PostgreSQL es una excepción a esto y permite **UNION**
 - Los casos base deben ir antes que los recursivos.
 - No se permite anidar consultas **WITH**.
 - El caso recursivo no puede contener:
 - **SELECT DISTINCT**
 - **GROUP BY**
 - **HAVING**
 - Agregados
 - **TOP**
 - Reuniones externas
 - Subconsultas
 - **ORDER BY**

Consultas recursivas: limitaciones en DES

- ◆ DES evita muchas de las limitaciones anteriores. Si embargo, no todas:
 - El caso recursivo no puede contener:
 - GROUP BY
 - HAVING
 - Agregados
 - Reuniones externas
 - ORDER BY
 - Referencia al caso recursivo en:
 - Subconsultas:
 - ◆ NOT IN.
 - ◆ <Op> ALL, con <Op> $\in \{ =, \neq, <, >, \leq, \geq \}$
 - Operando derecho de EXCEPT

Consultas recursivas: grafo de dependencias y estratificación

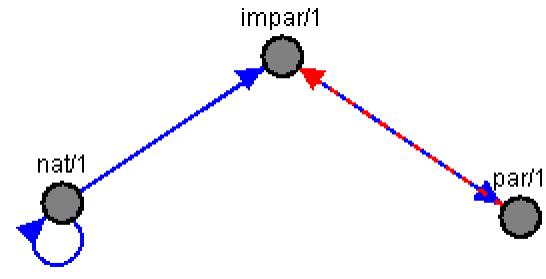
- ◆ El sistema DES, al igual que otros sistemas deductivos, implementa el concepto de *negación estratificada*.
- ◆ Intuitivamente, este concepto es una limitación sintáctica de los programas y consultas con respecto a la negación.
- ◆ Cuando un programa o consulta no cumple esta limitación se dice que no es estratifiable (**Warning:** Non stratifiable program).
- ◆ La negación aparece, entre otras situaciones, en el cálculo de la diferencia de conjuntos $A - B$: "El resultado contiene los elementos de A que *no están* en B ".
- ◆ Por ejemplo, la consulta de la siguiente página no es estratifiable.

Consultas recursivas: grafo de dependencias y estratificación

```
WITH
nat(x) AS (
    SELECT 1
    UNION
    SELECT x+1 FROM nat),
par(x) AS (
    SELECT 0
    UNION ALL
    SELECT x+1 FROM impar),
impar(x) AS (
    SELECT x FROM nat
    EXCEPT
    SELECT x FROM par)
SELECT x FROM par;
```

Consultas recursivas: grafo de dependencias y estratificación

```
WITH
  nat(x) AS (
    SELECT 1
    UNION
    SELECT x+1 FROM nat),
  par(x) AS (
    SELECT 0
    UNION ALL
    SELECT x+1 FROM impar),
  impar(x) AS (
    SELECT x FROM nat
    EXCEPT
    SELECT x FROM par)
SELECT x FROM par;
```



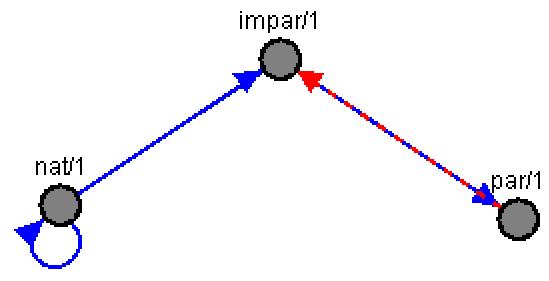
Grafo de dependencias entre relaciones

- El grafo de dependencias tiene arcos etiquetados como positivos (azules) y negativos (rojos).
- Un nodo *a* con un arco *positivo* entrante desde *b* representa que el cálculo de *a* se hace a partir de *b*.
- Un nodo *a* con un arco *negativo* entrante desde *b* representa que *b* se debe de calcular *antes e independientemente* de *a*.

Consultas recursivas: grafo de dependencias y estratificación

- ◆ Algo similar ocurre en el mismo ejemplo usando **NOT IN** en lugar de **EXCEPT**:

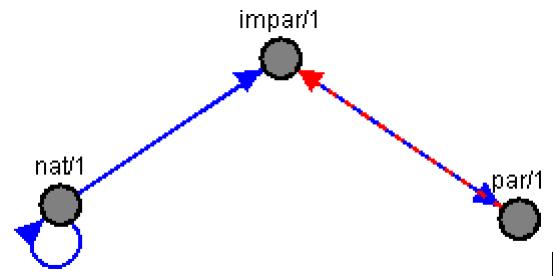
```
WITH
  nat(x) AS (
    SELECT 1
    UNION
    SELECT x+1 FROM nat),
  par(x) AS (
    SELECT 0
    UNION ALL
    SELECT x+1 FROM impar),
  impar(x) AS (
    SELECT x FROM nat
    WHERE x NOT IN (
      SELECT x FROM par))
SELECT x FROM par;
```



Consultas recursivas: estratificación

- ◆ La estratificación consiste en clasificar las relaciones por estratos (números consecutivos) para aplicar un mecanismo ordenado de resolución.
- ◆ Esta clasificación se puede especificar a partir de la función $\text{str}(\Delta, r)$, que asigna un número de estrato a una relación r en la base de datos Δ :
 - Para un arco positivo $r_1 \xleftarrow{\text{blue}} r_2 \Rightarrow \text{str}(\Delta, r_1) \geq \text{str}(\Delta, r_2)$
 - Para un arco negativo $r_1 \xleftarrow{\text{red}} r_2 \Rightarrow \text{str}(\Delta, r_1) > \text{str}(\Delta, r_2)$
- ◆ En el ejemplo anterior se tiene
 - $\text{par} \xleftarrow{\text{blue}} \text{impar} \Rightarrow \text{str}(\Delta, \text{par}) \geq \text{str}(\Delta, \text{impar})$
 - $\text{impar} \xleftarrow{\text{red}} \text{par} \Rightarrow \text{str}(\Delta, \text{impar}) > \text{str}(\Delta, \text{par})$

Lo cual no se puede satisfacer y, por tanto,
el grafo no es estratificable.
- ◆ De forma equivalente se puede enunciar que una consulta es estratifiable si no hay ningún ciclo que involucre un arco negativo.
 - En este ejemplo hay un ciclo formado por las vistas par e impar con un arco negativo que apunta de par a impar.



Consultas hipotéticas (*fueras del estándar*)

- ◆ Algunos sistemas de bases de datos permiten razonamiento hipotético (OLAP, HR-SQL, DES...).
- ◆ Idea: formular una hipótesis y examinar lo que se puede deducir de ella.
- ◆ Aplicación: sistemas de ayuda a la toma de decisiones ("*what-if*" applications).
- ◆ Algun sistema SQL incluye construcciones **ASSUME** para formular estas hipótesis.
- ◆ Hay dos tipos de hipótesis:
 - Extensionales: se asumen tuplas (**SELECT** sin **FROM**).
 - Intensionales: se asume intensionalmente un conjunto de tuplas definidas por una instrucción **SELECT** (con **FROM**).

Consultas hipotéticas. Ejemplo extensional

```
CREATE TABLE flight(origin STRING, destination STRING, time real);
```

```
INSERT INTO flight VALUES
('lon','ny',9.0), ('mad','par',1.5), ('par','ny',10.0);
```

```
CREATE OR REPLACE VIEW travel(origin,destination,time) AS
WITH connected(origin,destination,time) AS
    SELECT * FROM flight
UNION
    SELECT flight.origin,connected.destination,
           flight.time+connected.time
    FROM flight,connected
    WHERE flight.destination = connected.origin
SELECT * FROM connected;
```

```
DES> SELECT * FROM travel;
answer ->
{ answer(lon,ny,9.0),
  answer(mad,ny,11.5),
  answer(mad,par,1.5),
  answer(par,ny,10.0) }
```



Corredores aéreos con tiempos definido en un habitual cierre transitivo

Consultas hipotéticas. Ejemplo

```
CREATE TABLE flight(origin STRING, destination STRING, time real);
```

```
INSERT INTO flight VALUES  
('lon','ny',9.0), ('mad','par',1.5), ('par','ny',10.0);
```

```
CREATE OR REPLACE VIEW travel(origin,destination,time) AS  
WITH connected(origin,destination,time) AS  
    SELECT * FROM flight
```

```
UNION  
    SELECT flight.origin,connected.o  
          ,flight.time+connected.ti  
         ,flight.destination  
        FROM flight,connected  
       WHERE flight.destination = connected.origin  
SELECT * FROM connected;
```

```
DES> SELECT * FROM travel;  
answer ->  
{ answer(lon,ny,9.0),  
  answer(mad,ny,11.5),  
  answer(mad,par,1.5),  
  answer(par,ny,10.0) }
```



DES>

Hipótesis extensional

```
ASSUME SELECT 'mad','lon',2.0 IN  
           flight(origin,destination,time)
```

```
SELECT * FROM travel;
```

answer ->

```
{ answer(lon,ny,9.0),  
  answer(mad,lon,2.0),  
  answer(mad,ny,11.0),  
  answer(mad,ny,11.5),  
  answer(mad,par,1.5),  
  answer(par,ny,10.0) }
```

Deducción

Consultas hipotéticas. Hipótesis intensionales

- ◆ Partiendo de una BD con ciertas conexiones de vuelos, determinar cuáles serían todos los posibles trayectos.

```
CREATE TABLE flight(origin STRING, destination STRING, time real);
```

```
INSERT INTO flight VALUES
('lon','ny',9.0), ('mad','par',1.5), ('par','ny',10.0);
```

Hipótesis intensional

```
DES> ASSUME (
    SELECT * FROM flight
    UNION
    SELECT f1.origin, f2.destination,
           f1.time+f2.time
    FROM flight f1, flight f2
    WHERE f1.destination = f2.origin )
    IN flight(origin,destination,time)
```

```
SELECT * FROM flight;
```

```
answer ->
{ answer(lon,ny,9.0),
  answer(mad,ny,11.5),
  answer(mad,par,1.5),
  answer(par,ny,10.0) }
```

Consultas hipotéticas. Hipótesis negativas

- ◆ Las hipótesis negativas permiten descartar tuplas en el proceso de deducción, tanto extensional como intensionalmente.

```
CREATE TABLE flight(origin STRING, destination STRING, time real);
```

```
INSERT INTO flight VALUES
('lon','ny',9.0), ('mad','par',1.5), ('par','ny',10.0);
```

```
CREATE OR REPLACE VIEW travel(origin,destination,time) AS ...
```

```
DES> ASSUME
```

```
    SELECT 'mad','lon',2.0 IN flight,
```

```
    SELECT 'mad','par',1.5 NOT IN flight
```

Hipótesis negativa

```
SELECT * FROM travel;
```

```
answer ->
{ answer(lon,ny,9.0),
  answer(mad,lon,2.0),
  answer(mad,ny,11.0),
  answer(par,ny,10.0) }
```

vs.

```
DES> SELECT * FROM travel;
answer ->
{ answer(lon,ny,9.0),
  answer(mad,ny,11.5),
  answer(mad,par,1.5),
  answer(par,ny,10.0) }
```

Consultas hipotéticas. Hipótesis negativa intensional

- ◆ Siguiendo el mismo ejemplo anterior:

```
CREATE TABLE flight(origin STRING, destination STRING, time real);
```

```
INSERT INTO flight VALUES
```

```
('lon','ny',9.0),('mad','par',1.5),('mad','lon',2.0),('par','ny',10.0);
```

```
CREATE OR REPLACE VIEW travel(origin,destination,time) AS ...
```

ASSUME

```
SELECT * FROM flight WHERE origin='par'  
NOT IN travel(origin,destination,time)
```

```
SELECT * FROM travel;
```

```
answer ->  
{ answer(lon,ny,9.0),  
  answer(mad,lon,2.0),  
  answer(mad,ny,11.0),  
  answer(mad,ny,11.5),  
  answer(mad,par,1.5) }
```

vs.

```
DES> SELECT * FROM travel;  
answer ->  
{ answer(lon,ny,9.0),  
  answer(mad,lon,2.0),  
  answer(mad,ny,11.0),  
  answer(mad,ny,11.5),  
  answer(mad,par,1.5) ,  
  answer(par,ny,10.0) }
```

Consultas hipotéticas. Semántica

- ◆ Dada una base de datos Δ y consulta Q con una serie de hipótesis positivas $H_1..H_n$, otras negativas $H_{n+1}..H_m$ sobre relaciones $R_1..R_m$ respectivamente, el resultado de calcular Q bajo estas hipótesis se puede expresar como:

$$\| Q \|_{\Delta^+ - \Delta^-}$$

donde:

$$\Delta^+ = \| H_1 \|_{\Delta} \cup \dots \cup \| H_n \|_{\Delta}$$

$$\Delta^- = \| H_{n+1} \|_{\Delta} \cup \dots \cup \| H_m \|_{\Delta}$$

Esta diferencia de conjuntos va a imponer una dependencia negativa en el grafo de dependencias

Δ es el conjunto de tuplas definidas extensionalmente (de la forma $E_i(\bar{X})$ para cada relación E_i definidas con **INSERT**) más las vistas, y $\| Q \|_{\Delta}$ representa el resultado extensional de la consulta Q (como tuplas $R_i(\bar{X})$) en el contexto de la base de datos Δ .

Consultas hipotéticas en DES. Sintaxis WITH

- ◆ DES usa consultas hipotéticas (positivas) para resolver consultas WITH.
- ◆ Para una relación R y consultas Q y Q' , las siguientes dos consultas son equivalentes:
 $\text{ASSUME } Q \text{ IN } R \ Q'; \quad \longleftrightarrow \quad \text{WITH } R \text{ AS } Q \ Q';$
- ◆ Si la relación R no existe, se crea localmente y funciona como el estándar.
- ◆ Si existe (como vista local o normal), su significado se sobrecarga en lugar de reemplazar la vista existente, como en el estándar.
- ◆ Por lo tanto, se puede dar el caso de que vistas que dependan de otras y usen el mismo nombre de vista local arrojen resultados "inesperados".

Consultas hipotéticas en DES. Sintaxis WITH

- ◆ Ejemplo de la diferencia entre DES y el estándar.

```
CREATE VIEW v1(a) AS  
  WITH r(a) AS SELECT 1  
    SELECT * FROM r;
```

```
CREATE VIEW v2(a) AS  
  WITH r(a) AS SELECT 2  
    SELECT * FROM r NATURAL JOIN v1;
```

```
DES> SELECT * FROM v2;  
answer(v2.a:int) ->  
{ answer(2) }
```

```
DES-postgresql>  
SELECT * FROM v2;  
Info: 0 tuples computed.
```

Consultas hipotéticas en DES. Sintaxis WITH

- ◆ Al no reemplazar vistas existentes sino sobrecargar su significado, en DES se admiten usos como:

WITH

```
r(a) AS SELECT 1,  
r(a) AS SELECT 2  
SELECT * FROM r;
```



DES>

```
answer(v2.a:int) ->  
{ answer(1),  
  answer(2) }
```

- ◆ ... que no se admiten en el estándar:

DES-postgresql>

```
Error: ODBC Code 1: ERROR: el nombre de consulta WITH  
«r» fue especificado más de una vez;
```

- ◆ Una aplicación puede ser separar el caso base y el recursivo:

WITH

```
r(a,b) AS SELECT 1,  
r(a,b) AS SELECT r1.a,r2.b FROM r r1 JOIN r r2 ON r1.b=r2.a  
SELECT * FROM r;
```

Modificación de datos de la BD

- ◆ Las instrucciones SQL que permiten cambiar los datos de la BD son:
 - **DELETE** → Elimina filas de una tabla
 - **INSERT** → Añade filas a una tabla
 - **UPDATE** → Modifica filas de una tabla
- ◆ Los ejemplos a continuación puedes añadirlos al archivo **ejercicio06.sql**.
- ◆ Las preguntas que se formulen hay que responderlas en el propio archivo como un comentario (recuerda que un comentario es una línea que empieza con **--**).

La instrucción DELETE

- ◆ La **eliminación** de filas se realiza con la sentencia **DELETE**.

Tiene dos formas:

- **DELETE FROM TABLA;**
 - **¡Cuidado!** Borra todas las filas
- **DELETE FROM TABLA WHERE *CONDICIÓN*;**
 - Esta forma solo borra las filas que cumplan la condición.
 - La parte *CONDICIÓN* puede incluir subconsultas.

Ejemplos de la instrucción DELETE

- ◆ A partir de la base de datos de empleados vamos a borrar todas las filas de empleados, pero antes las guardaremos: pulsa el botón **Commit** en DESweb.
- ◆ Con una consulta **SELECT**, averigua el número total de empleados.
- ◆ Ejecuta **DELETE FROM EMPLOYEE;**
- ◆ ¿Cuántas filas te informa DES que se han borrado?
- ◆ Comprueba si queda alguna fila con una consulta **SELECT**.
- ◆ Recupera los datos pulsando el botón **Rollback** en DESweb y comprueba que se han recuperado las filas.

Ejemplos de la instrucción DELETE

- ◆ Ahora vamos a borrar *solo* un empleado: el que tiene código **000010** (ahora no guardaremos, ya lo hemos hecho antes).
- ◆ Ejecuta:
`DELETE FROM EMPLOYEE WHERE EMPNO='000010';`
- ◆ Comprueba el resultado: ¿se ha borrado este empleado?
- ◆ Fíjate que, como EMPNO es un campo clave, solo es posible borrar una fila con esta instrucción (no se puede repetir el valor de EMPNO).
- ◆ Recupera la fila pulsando el botón **Rollback** en DESweb y compruébalo.

Ejemplos de la instrucción DELETE

- ◆ Vamos a borrar todos los empleados que no tengan inicial (MIDINIT).
- ◆ Ejecuta:
`DELETE FROM EMPLOYEE WHERE MIDINIT IS NULL;`
- ◆ ¿Cuántas filas se han borrado? (DES te informa en la consola).
- ◆ Recupera las filas pulsando el botón **Rollback** en DESweb.

Ejemplos de la instrucción DELETE

- ◆ Finalmente vamos a borrar los empleados del departamento PLANNING.
- ◆ Ejecuta:

```
DELETE FROM EMPLOYEE WHERE WORKDEPT =
(SELECT DEPTNO
  FROM DEPARTMENT
 WHERE DEPTNAME = 'PLANNING');
```
- ◆ Fíjate que hemos escrito una subconsulta de la sentencia DELETE:
 - Por cada fila de EMPLOYEE se comprueba que el valor de DEPTNO sea el del departamento de nombre PLANNING. Si es así, se borra.
- ◆ Comprueba que no quede ningún empleado que pertenezca a ese departamento y finalmente recupera los datos con el botón Rollback.

La instrucción **INSERT**

◆ La **inserción** de filas se realiza con la sentencia **INSERT**:

- Es posible insertar directamente valores (usando el modificador **VALUES**).
- También se puede insertar el conjunto de resultados de una consulta.
- En cualquier caso, los valores que se insertan deben pertenecer al tipo de cada uno de las columnas de la tabla.

Ejemplo de la instrucción INSERT

- ◆ Crea la tabla CLIENTES(DNI,NOMBRE,DIR) con tipos **STRING**, y prueba las siguientes consultas.

- ◆ La inserción de una fila añade los datos en el orden en que aparecen en la definición de la tabla (primero el DNI, luego NOMBRE y luego DIR):

```
INSERT INTO CLIENTES VALUES ('1111','Mario','C/. Mayor, 3');
```

- ◆ Es equivalente a las siguientes sentencias en las que se explicita el orden de las columnas (borra con **DELETE** para probarlas cuando sea necesario):

```
INSERT INTO CLIENTES (NOMBRE,DIR,DNI)  
VALUES ('Mario','C/. Mayor, 3','1111');
```

```
INSERT INTO CLIENTES (DNI,DIR,NOMBRE)  
VALUES ('1111','C/. Mayor, 3','Mario');
```

- ◆ También es posible insertar un subconjunto de campos. Los que no se especifiquen recibirán un valor **NULL** (a menos que la columna tenga una restricción **DEFAULT**, como veremos más adelante). Compruébalo con una **SELECT** tras la siguiente inserción:

```
INSERT INTO CLIENTES (DNI,NOMBRE) VALUES ('2222','Jesús');
```

Ejemplo de la instrucción INSERT

```
INSERT INTO EMPLOYEE  
VALUES ('111111', 'JOHN', NULL, 'SMITH', 'C01', NULL,  
'2019-06-25', NULL, 19,  
NULL, NULL, 45000, NULL, NULL);
```



Estas dos consultas son
equivalentes

```
INSERT INTO EMPLOYEE(EMPNO, FIRSTNAME, LASTNAME, WORKDEPT,  
HIREDATE, EDLEVEL, SALARY)  
VALUES ('111111', 'JOHN', 'SMITH', 'C01',  
'2019-06-25', 19, 45000);
```



Comprueba el resultado con
una SELECT que liste solo
este empleado.

EMPNO	FIRSTNAME	LASTNAME	WORKDEPT	HIREDATE	EDLEVEL	SALARY
111111	JOHN	null	SMITH	C01	null	2019-06-25

Pages: 1 1 of 1 Rows: 1

La instrucción **INSERT** y la restricción **DEFAULT**

- ◆ ¿Recuerdas la restricción **DEFAULT** cuando se crea una tabla?
 - (Páginas 10 y 11)
- ◆ ¿Recuerdas también que podíamos añadir restricciones a una tabla?
 - (Página 16)
- ◆ Añade una restricción a la tabla **CLIENTES** con:

```
ALTER TABLE CLIENTES ALTER
DIR STRING DEFAULT 'Falta la dirección';
```

 - Esto redefine el campo DIR con su tipo y añade la restricción de columna **DEFAULT** que no habíamos establecido antes. En definitiva, redefine una columna con la misma sintaxis que hubiésemos usado en la instrucción **CREATE TABLE**.
- ◆ Inserta una nueva fila en **CLIENTES**:

```
INSERT INTO CLIENTES(DNI, NOMBRE) VALUES ('3333', 'Carlos');
```

 - ◆ Observa que el campo DIR se rellena con el valor por defecto que hemos definido en lugar de con **NULL**.

Instrucción **INSERT**: añadir varias filas

- ◆ Con una única instrucción **INSERT** se puede añadir más de una fila a la vez:
 - Con el modificador **VALUES**, separando con comas las tuplas a insertar.
 - No obstante, no todos los SGBD permiten incluir varias tuplas en la misma **INSERT** (Oracle, por ejemplo, no lo admite).
 - Con el resultado de una consulta **SELECT**.

Ejemplo para añadir varias filas con **VALUES**

- ◆ Borra las filas de **CLIENTES** y añade las dos filas a la vez:

```
INSERT INTO CLIENTES(NOMBRE, DIR, DNI) VALUES  
  ('Mario', 'C/. Mayor, 3', '1111'),  
  ('Jesús', NULL, '2222');
```

- ◆ Observa que ahora hemos tenido que incluir el valor **NULL** explícitamente en la segunda tupla. No es posible dar dos listas de campos distintos para insertar estas filas en la misma instrucción.
- ◆ No todos los sistemas admiten encadenar tuplas y tendremos que repetir la instrucción **INSERT** por cada una de ellas.

Ejemplo para añadir varias filas con SELECT

- ◆ Copiaremos algunos datos de la tabla EMPLOYEE a una nueva (TESTEMP)
- ◆ Crea la tabla TESTEMP(EMPNO CHAR(6), LASTNAME VARCHAR(15), WORKDEPT CHAR(3), HIREDATE DATE, SALARY FLOAT, BONUS FLOAT)
- ◆ Con la siguiente instrucción copiaremos datos de los empleados cuyo identificador sea menor o igual que 000050:

```
INSERT INTO TESTEMP
```

```
SELECT EMPNO, LASTNAME, WORKDEPT, HIREDATE, SALARY, BONUS  
FROM EMPLOYEE  
WHERE EMPNO <= '000050';
```

EMPLOYEE.EN	EMPLOYEE.LA	EMPLOYEE.WC	EMPLOYEE.HI	EMPLOYEE.SA	EMPLOYEE.BC
000010	HAAS	A00	1995-01-01	152750	4220
000020	THOMPSON	B01	2003-10-10	94250	3300
000030	KWAN	C01	2005-04-05	98250	3060
000050	GEYER	E01	1979-08-17	80175	3214
000060	STERN	D11	2003-09-14	72250	2580
000070	PULASKI	D21	2005-09-30	96170	2893
000090	HENDERSON	E11	2000-08-15	89750	2380
000100	SPENSER	E21	2000-06-19	86150	2092
000110	LUCCHESI	A00	1988-05-16	66500	3720
000120	O'CONNELL	A00	1993-12-05	49250	2340

En varios SQL se prohíbe que la subconsulta haga referencia a la misma tabla en la que se quieren insertar las tuplas (DES, Oracle y DB2 sí lo permiten).

La instrucción UPDATE

- ◆ La **modificación** de filas se realiza con la sentencia **UPDATE**

- Es posible elegir el conjunto de filas que se van a actualizar usando la cláusula **WHERE**.
- Si no se usa **WHERE**, se actualizarían *todas* las filas.
- Al igual que en las instrucciones anteriores, se pueden usar subconsultas.

- ◆ La sintaxis de esta instrucción es:

```
UPDATE TABLA SET COL1=EXPR1, ..., COLN=EXPRN  
WHERE COND;
```

- ◆ Esta consulta modifica la columna **COLi** para que tome el valor resultado de **EXPRi** de cada fila para la que se cumpla la condición **COND**.
- ◆ Si no se añade **WHERE**, se asume que **COND** es **true**.

Ejemplos de la instrucción UPDATE

- ◆ Aumenta el sueldo de los empleados en TESTEMP en un 5%, pero sin decimales (redondeando).

```
UPDATE TESTEMP SET SALARY = ROUND(SALARY * 1.05);
```

- ◆ Aumenta la bonificación (BONUS) en un 1% (con dos decimales) a aquellos empleados cuyo sueldo sea inferior a 100.000 \$.

```
UPDATE TESTEMP SET BONUS = ROUND(BONUS * 1.01, 2)  
WHERE SALARY < 100000.0;
```

- ◆ Modifica el código de empleado y su sueldo simultáneamente para el empleado con código 30.

```
UPDATE TESTEMP SET  
    EMPNO = '000040',  
    SALARY = 99000  
WHERE EMPNO = '000030';
```

Ejemplo de la instrucción UPDATE con subconsulta

- ◆ Se debe aumentar en un 3% el sueldo de los empleados del departamento INFORMATION CENTER (redondeando sin decimales):

```
UPDATE TESTEMP
SET SALARY = ROUND(SALARY * 1.03)
WHERE WORKDEPT =
(SELECT DEPTNO
  FROM DEPARTMENT
 WHERE DEPTNAME = 'INFORMATION CENTER');
```

Ejemplo de la instrucción UPDATE con subconsulta y agregados

- ◆ Para la siguiente base de datos, queremos incluir en la tabla GRUPOS a todos los grupos musicales, junto con su número de álbumes publicados:
 - GRUPOS (NOMBRE, ALBUMES) LP (TIT, GRUPO, AÑO, NUM_CANC)

- ◆ Crea las dos tablas con las claves primarias y externa.
- ◆ Inserta las filas GRUPOS = {('Mecano', 0), ('Queen', 0)} y LP = {('Mecano', 'Mecano', 1982, 12), ('Ya viene el Sol', 'Mecano', 1984, 10), ('The Game', 'Queen', 1980, 10)}.
- ◆ Ejecuta la siguiente consulta para actualizar el número de álbumes en GRUPOS:

```
UPDATE GRUPOS G
SET ALBUMES = (SELECT COUNT(DISTINCT TIT)
                FROM LP
               WHERE G.NOMBRE = GRUPO
                 GROUP BY GRUPO);
```

INSERT, DELETE y UPDATE en vistas

- ◆ En la mayoría de los sistemas es posible aplicar estas instrucciones a una vista para modificarla.
- ◆ Por ejemplo, añadir una nueva fila a la vista DEPT:

```
CREATE VIEW DEPT AS
```

```
    SELECT DEPTNO, DEPTNAME, ADMRDEPT  
    FROM DEPARTMENT;
```

```
INSERT INTO DEPT VALUES
```

```
    ('999', 'AI', '99');
```

- ◆ Que es equivalente a la inserción en la tabla DEPARTMENT de la tupla:

```
    ('999', 'AI', NULL, '99', NULL)
```

```
CREATE TABLE DEPARTMENT  
    (DEPTNO CHARACTER(3) PRIMARY KEY  
     ,DEPTNAME VARCHAR(36) NOT NULL  
     ,MGRNO CHARACTER(6)  
     ,ADMRDEPT CHARACTER(3) NOT NULL  
     ,LOCATION CHARACTER(16));
```

Problemas en la actualización mediante vistas

```
CREATE VIEW Professional_Data AS  
SELECT EMPNO, FIRSTNME, LASTNAME, EDLEVEL, DEPTNAME  
FROM EMPLOYEE JOIN DEPARTMENT ON WORKDEPT = DEPTNO;
```

```
INSERT INTO Professional_Data VALUES ('01000', 'Pepe',  
'Gotera y Otilio', 1, 'Chapuzas a domicilio');
```

Error SQL: ORA-01776: no se puede modificar más de una tabla base a través de una vista de **unión**

JOIN, Oracle, JOIN, no unión....

- ¿A qué departamento, si hay varios para Pepe?
- ¿Y si no hay ninguno para Pepe?



Restricciones sobre vistas actualizables

- ◆ Las vistas actualizables tienen que cumplir ciertas restricciones en la instrucción de selección que define la vista:
 - No puede contener ninguna reunión, es decir, la vista debe estar definida sobre una única tabla o vista (que también debe ser actualizable).
 - Todas las columnas obligatorias (**NOT NULL**) deben aparecer en la definición de la vista.
 - No puede contener operadores de conjunto como **UNION**, **EXCEPT** o **INTERSECT**.
 - La cláusula **DISTINCT** no está permitida.
 - La cláusula **SELECT** no puede contener funciones de agregación.
 - No se pueden utilizar las cláusulas **GROUP BY** ni **HAVING**.
 - Vistas aparentemente simples también dan problemas:

Otras modificaciones no se pueden traducir

```
CREATE VIEW EMP_A00 AS  
    SELECT EMPNO, FIRSTNAME, LASTNAME,  
          WORKDEPT, EDLEVEL  
     FROM EMPLOYEE  
    WHERE DEPT = 'A00';
```

- ◆ ¿Qué ocurriría si intentásemos insertar ('007', 'Bond', 'James', 'KILL', 'LICENCE') en EMP_A00?
- ◆ La mayoría de las implementaciones SQL permiten modificaciones solo sobre vistas realmente simples (DES, ni eso)

Vistas materializadas

- ◆ La **materialización** de una vista consiste en crear una tabla física que contenga todas las tuplas del resultado de la consulta que define a la vista.
- ◆ Si se actualizan las relaciones que aparecen en esa consulta, la vista materializada deja de estar actualizada.
 - Es necesario mantener la vista actualizándola bien automática o manualmente.

CREATE MATERIALIZED VIEW

history_instructors AS

SELECT *

FROM instructor

WHERE dept_name = 'History';

Las vistas materializadas no están soportadas en DES.

Transacciones

- ◆ Hemos estado usando los botones **Commit** y **Rollback** en DESweb. Ahora estudiaremos el concepto de transacción para entender la utilidad completa de estos botones.
- ◆ Una **transacción** es un conjunto de instrucciones que se ejecutan formando una *unidad de trabajo*, es decir, de forma indivisible o atómica.
- ◆ Por regla general, todas las operaciones relacionadas entre sí que se ejecuten dentro una misma unidad de trabajo, deben ejecutarse como un bloque.
- ◆ Así, si todas funcionan, la operación conjunta del bloque tiene éxito, pero si falla cualquiera de ellas, deberán retrocederse todas las anteriores que ya se hayan realizado evitando que el sistema quede en un estado inconsistente.

Ejemplo de transacción

- ◆ Supongamos que deseamos transferir dinero de una cuenta a otra, este proceso se podría detallar en las operaciones:
 1. Quitar el dinero de la cuenta origen.
 2. Poner el dinero en la cuenta destino.
- ◆ Si se retira el dinero de la cuenta origen en el paso 1 y hay algún problema que evite que pueda continuar el proceso y se realice el paso 2, el dinero habrá salido de la cuenta origen pero no se habrá añadido a la cuenta destino. Habría un dinero que ha desaparecido, por lo que la base de datos se encontraría en un estado inconsistente.
- ◆ Para evitar este tipo de situaciones existen las transacciones que marcan bloques completos de operaciones y comprueban que, o bien se realizan todas, o ninguna.

Control de transacciones

- ◆ Las transacciones se pueden comprometer o cancelar.
- ◆ Cuando una transacción se compromete:
 - La transacción finaliza.
 - Los cambios provocados por ella se hacen permanentes y visibles para todos los usuarios de la base de datos.
- ◆ Cuando una transacción se cancela:
 - Todos los cambios hechos por las instrucciones de la transacción se cancelan.
- ◆ Cada sistema de bases de datos ofrece en unas instrucciones particulares para este control porque dependen del lenguaje de programación anfitrión que proporcionen (PL/SQL, Transact-SQL...). No obstante, dos de ellas son estándar y comunes a la mayoría de sistemas:

Control de transacciones

◆ La sentencia:

COMMIT WORK;

compromete una transacción y hace permanentes y visibles los cambios (**WORK** es opcional).

En DESweb puedes pulsar también el botón **Commit** con el mismo efecto (si no guardas los cambios con **COMMIT**, al restaurar la consola los perderás).

◆ La sentencia:

ROLLBACK WORK;

(**WORK** es también opcional aquí) descarta todos los cambios realizados por la transacción realizados desde el último **COMMIT** o **ROLLBACK**.

En DESweb puedes pulsar también el botón **Rollback** con el mismo efecto.

COMMIT se entiende vulgarmente como "*¡Ok! Voy a guardar los cambios, no quiero perderlos*";
ROLLBACK como "*¡Vaya! He borrado lo que no debía, voy a deshacer los cambios*".

Control de transacciones

- ◆ Las sentencias **COMMIT** y **ROLLBACK** se pueden incluir en un programa o *script* (una secuencia de instrucciones escrita en el lenguaje anfitrión) que permita decidir las acciones necesarias.
- ◆ Un programa es una secuencia de instrucciones que incluye consultas SQL, e instrucciones y comandos del lenguaje anfitrión.
- ◆ En DES se puede usar por ejemplo el comando del lenguaje anfitrión:
/IF Condición /GOTO Lugar
 - Donde Lugar es un punto del programa definido por la etiqueta :Lugar, (observa que las etiquetas llevan ":" antes del nombre) y **/GOTO** es un comando que transfiere el control de programa a ese punto.
 - Si se cumple Condición, entonces la próxima instrucción a ejecutar será la que preceda a la línea en que se encuentre :Lugar.
- ◆ Para comunicar SQL con el lenguaje anfitrión se usan instrucciones como:
SELECT Expresión INTO Variable FROM ...
 - Esta es una instrucción **SELECT** habitual a la que se le añade **INTO Variable** para almacenar en la variable **Variable** del lenguaje anfitrión el resultado de la consulta SQL.
 - Aunque esta versión de **SELECT** sí es estándar, el resto de instrucciones (o comandos) es en general particular de cada sistema.

Transacciones. Ejemplo I

- ◆ Volviendo al ejemplo inicial de transferencia monetaria entre cuentas, se pretende transferir 500 euros de la cuenta número '3207' a la '3208'.
- ◆ La tabla CUENTAS tiene el esquema:

CUENTAS(NÚMERO , TIPO, SALDO)

- ◆ Crea la la tabla CUENTAS con tipos **STRING** salvo el saldo que será **FLOAT**.
- ◆ Inserta las filas:
`{('3207', 'Corriente', 1500), ('3208', 'Ahorro', 2500)}`
- ◆ Crea un nuevo archivo denominado **transferencia.sql** con el contenido de la siguiente página.
- ◆ Ejecútalo y observa el contenido de CUENTAS. Escribe los cambios en el archivo **ejercicio06.sql** como comentarios.
- ◆ Borra la fila correspondiente a la cuenta '3208'.
- ◆ Ejecuta de nuevo el archivo **transferencia.sql**
- ◆ Anota lo que hayas observado en cuanto a los cambios de la tabla CUENTAS. ¿Por qué aparece de nuevo la cuenta '3208'?

Ejemplo de transacción

Archivo transferencia.sql

```
UPDATE CUENTAS SET SALDO = SALDO - 500  
WHERE NÚMERO = '3207';
```

```
SELECT COUNT(*) INTO existe  
FROM CUENTAS  
WHERE NÚMERO = '3208';
```

```
/IF ($existe$ \= 1) /GOTO deshacer
```

```
UPDATE CUENTAS SET SALDO = SALDO + 500  
WHERE NÚMERO = '3208';
```

```
COMMIT;  
/GOTO fin
```

```
:deshacer  
ROLLBACK;
```

```
:fin
```

Aquí se traslada el valor de recuento a la variable existe.

Operador "distinto de" en la sintaxis del lenguaje anfitrión.

Así se accede a las variables en este lenguaje: poniéndolas entre símbolos de dólar.

Asegúrate de dejar una línea en blanco al final.

Transacciones. Ejemplo II

```
CREATE TABLE EMPLEADO
(NIF      VARCHAR(9) NOT NULL PRIMARY KEY,
NOMBRE    VARCHAR(20),
SALARIO   NUMBER(6,2),
APELLIDOS VARCHAR(40));
    tabla EMPLEADO creada
COMMIT;
    Se guarda el cambio de la BD (nueva tabla)
INSERT INTO EMPLEADO VALUES('10000000A','Jorge',3000.11,'Pérez Sala');
    1 fila insertada
ROLLBACK;
    Transacción retrocedida (cancelada)
INSERT INTO EMPLEADO VALUES('30000000C','Javier',2000.22,'Sala Rodríguez');
    1 fila insertada
INSERT INTO EMPLEADO VALUES('30000000C','Soledad',2000.33,'López J.');
    Error: clave repetida
INSERT INTO EMPLEADO VALUES('40000000D','Sonia',1800.44,'Moldes R.');
    1 fila insertada
INSERT INTO EMPLEADO VALUES('50000000E','Antonio',1800.44,'López A.');
    1 fila insertada
COMMIT;
    Transacción comprometida
```

Ejecuta estas instrucciones para probar este otro ejemplo.

Transacciones. Ejemplo

- Estado de las tablas tras la ejecución anterior:

30000000C	Javier	2000.22	Sala Rodríguez
40000000D	Sonia	1800.44	Moldes R.
50000000E	Antonio	1800.44	López A.

Comprueba que consigues los mismos resultados.

Puntos de control (**SAVEPOINT**)

- ◆ Es posible declarar puntos intermedios en una transacción que permiten guardar el trabajo realizado hasta un punto.
- ◆ Un punto de control debe tener un nombre asociado, y se establece mediante la instrucción:

SAVEPOINT Nombre

- ◆ Para cancelar una transacción hasta un punto de control determinado se utiliza la sentencia:

ROLLBACK TO SAVEPOINT Nombre

- Solo se deshacen las instrucciones realizadas entre el punto de control y el **ROLLBACK**.
 - Preserva el **SAVEPOINT** correspondiente y los **SAVEPOINT** anteriores a este.
- ◆ Una transacción que se retrocede a un punto de control permanece activa.

Transacciones. Ejemplo (III)

```
UPDATE EMPLEADO SET SALARIO = 7000  
WHERE NIF = '30000000C';
```

```
SAVEPOINT after_salario;
```

```
UPDATE EMPLEADO SET SALARIO = 12000  
WHERE NIF = '40000000D';
```

```
ROLLBACK TO SAVEPOINT after_salario;
```

```
UPDATE EMPLEADO SET SALARIO = 11000  
WHERE NIF = '40000000D';
```

```
COMMIT;
```

```
INSERT INTO EMPLEADO VALUES ('70000000C','Soledad',2000.33,'Lopez J.');
```

Ejecuta estas instrucciones para probar este otro ejemplo.

Transacciones. Ejemplo IV

- Estado de las tablas tras la ejecución anterior:

30000000C	Javier	7000	Sala Rodríguez
40000000D	Sonia	11000	Moldes R.
50000000E	Antonio	1800.44	López A.
70000000C	Soledad	2000.33	López J.

- Comprueba que consigues los mismos resultados .
- El último registro (**70000000C**) solo será permanente en la base de datos (y visible al resto de los usuarios) si se compromete la transacción.
- Comprueba que efectivamente lo pierdes si pulsas el botón **Restart** de la consola.

Finale: conclusiones

- ◆ Llegar a esta transparencia habiendo practicado todos los ejercicios propuestos te debería haber proporcionado un conocimiento de SQL que, aunque básico, es bastante extenso.
- ◆ Por recordar algunas de las nociones aprendidas has estudiado:
 - El lenguaje LDD para crear, borrar y modificar tablas y vistas.
 - El lenguaje LMD para:
 - Insertar, borrar y modificar datos de las tablas.
 - Recuperar datos de las tablas con consultas:
 - ◆ Sobre una o múltiples tablas.
 - ◆ Usando expresiones y condiciones con muchas posibilidades.
 - ◆ Incluyendo subconsultas
 - ◆ Con funciones de agregación y agrupaciones
 - En particular has llegado a comprender las consultas recursivas, un concepto que más de un experimentado programador de SQL desconoce incluso su existencia.
 - ◆ Y con esto...

The End

¡Ya era hora!

