4. CONSULTA DE BASES DE DATOS. LENGUAJE DE MANIPULACIÓN DE DATOS

4.1. INTRODUCCIÓN

A lo largo de esta unidad nos centraremos en la cláusula SELECT. Sin duda es el comando más versátil del lenguaje SQL. El comando **SELECT** permite:

- Obtener datos de ciertas columnas de una tabla (proyección).
- Obtener registros (filas) de una tabla de acuerdo con ciertos criterios (selección).
- Mezclar datos de tablas diferentes (asociación, join).

4.2. CONSULTAS

Para realizar consultas a una base de datos relacional hacemos uso de la sentencia SELECT. La sintaxis básica del comando SELECT es la siguiente:

```
SELECT * | {[ DISTINCT ] columna | expresión [[AS] alias ], ...}
FROM nombre_tabla;
```

Donde:

- *. El asterisco significa que se seleccionan todas las columnas.
- *DISTINCT*. Hace que no se muestren los valores duplicados.
- *columna*. Es el nombre de una columna de la tabla que se desea mostrar.
- *expresión*. Una expresión válida SQL.
- *alias*. Es un nombre que se le da a la cabecera de la columna en el resultado de esta instrucción.

Ejemplos:

```
-- Selección de todos los registros de la tabla CLIENTES
SELECT * FROM CLIENTES;

-- Selección de algunos campos de la tabla CLIENTES
SELECT nombre, apellido1, apellido2 FROM CLIENTES;
```

4.3. CÁLCULOS

4.3.1. Cálculos Aritméticos

Los operadores + (suma), - (resta), * ( ) y / (división), se pueden utilizar para hacer cálculos en las consultas. Cuando se utilizan como expresión en una consulta SELECT, no modifican los datos originales.

Ejemplo:

```
-- Consulta con 3 columnas
SELECT nombre, precio, precio*1.16 FROM ARTICULOS;


-- Ponemos un alias a la tercera columna.
-- Las comillas dobles en el alias hacen que se respeten mayúsculas y minúsculas,
-- de otro modo siempre aparece en mayúsculas
SELECT nombre, precio, precio*1.16 AS "Precio + IVA" FROM ARTICULOS;
```

La prioridad de esos operadores es: tienen más prioridad la multiplicación y división, después la suma y la resta. En caso de igualdad de prioridad, se realiza primero la operación que esté más a la izquierda. Como es lógico se puede evitar cumplir esa prioridad usando paréntesis; el interior de los paréntesis es lo que se ejecuta primero. Cuando una expresión aritmética se calcula sobre valores NULL, el resultado de la expresión es siempre NULL.

4.3.2. Concatenación

El operador **||** es el de la concatenación. Sirve para unir textos.

| (Alt graf + 1)

Ejemplo:

```
SELECT tipo, modelo, tipo || '-' || modelo "Clave Pieza" FROM PIEZAS;
```

En MySQL se usa el comando

```
CONCAT (Expresión1, Expresión2, Expresión3, …)
```

```
SELECT Concat (tipo, modelo, tipo, '-', modelo) “"Clave Pieza" FROM PIEZAS;
```

El resultado de esa consulta tendría esta estructura:

TIPO	MODELO	Clave Pieza
AR	6	AR-6
AR	7	AR-7
AR	8	AR-8
AR	9	AR-9
AR	12	AR-12
AR	15	AR-15
AR	20	AR-20
AR	21	AR-21
BI	10	BI-10
BI	20	BI-20
BI	22	BI-22
BI	24	BI-24

4.3.3. Condiciones o Filtros

Se puede realizar consultas que restrinjan los datos de salida de las tablas. Para ello se utiliza la cláusula WHERE. Esta cláusula permite colocar una condición que han de cumplir todos los registros que queremos que se muestren. Las filas que no la cumplan no aparecerán en la ejecución de la consulta.

Nota

Con el comando SELECT indicamos las columnas que queremos que aparezcan en nuestra consulta. Con el comando WHERE indicamos las filas que queremos que aparezcan en nuestra consulta (serán las que cumplan las condiciones que especifiquemos detrás del WHERE).

Ejemplo:

```
-- Tipo y modelo de las piezas cuyo precio es mayor que 3
SELECT tipo, modelo
FROM PIEZAS
WHERE precio > 3;
```

4.3.3.1. Operadores de comparación

Los operadores de comparación que se pueden utilizar en la cláusula WHERE son:

Operador	Significado
>	Mayor que
<	Menor que
>=	Mayor o igual que
<=	Menor o igual que
=	Igual
<>	Distinto
!=	Distinto

Se pueden utilizar tanto para comparar números como para comparar textos y fechas. En el caso de los textos, las comparaciones se hacen en orden alfabético. Sólo que es un orden alfabético estricto. Es decir el orden de los caracteres en la tabla de códigos. Así la letra Ñ y las vocales acentuadas nunca quedan bien ordenadas ya que figuran con códigos más altos. Las mayúsculas figuran antes que las minúsculas (la letra “Z” es menor que la “a”).

4.3.3.2. Operadores lógicos

Son:

Operador	Significado
AND	Devuelve verdadero si las expresiones a su izquierda y derecha son ambas verdaderas
OR	Devuelve verdadero si cualquiera de las dos expresiones a izquierda y derecha del OR, son verdaderas
NOT	Invierte la lógica de la expresión que está a su derecha. Si era verdadera, mediante NOT pasa a ser falso.

Ejemplos:

```
-- Personas entre 25 y 50 años
SELECT nombre, apellidos
FROM PERSONAS
WHERE edad >= 25 AND edad <= 50;

-- Personas de más de 60 y menos de 20
SELECT nombre, apellidos
FROM PERSONAS
WHERE edad > 60 OR edad < 20;
```

4.3.3.3. BETWEEN

El operador BETWEEN nos permite obtener datos que se encuentren entre dos valores determinados (incluyendo los dos extremos).

Ejemplo:

```
-- Selección de las piezas cuyo precio está entre 3 y 8
-- (ambos valores incluidos)
SELECT tipo, modelo, precio
FROM PIEZAS
WHERE precio BETWEEN 3 AND 8;
```

El operador NOT BETWEEN nos permite obtener los los valores que son menores (estrictamente) que el más pequeño y mayores (estrictamente) que el más grande. Es decir, no incluye los extremos.

Ejemplo:

```
-- Selección de las piezas cuyo precio sea menor que 3 o mayor que 8
-- (los de precio 3 y precio 8 no estarán incluidos)
SELECT tipo, modelo, precio
FROM PIEZAS
WHERE precio NOT BETWEEN 3 AND 8;
```

4.3.3.4. IN

El operador IN nos permite obtener registros cuyos valores estén en una lista:

Ejemplo:

```
-- Selección de las piezas cuyo precio sea igual a 3, 5 u 8
SELECT tipo, modelo, precio
FROM PIEZAS
WHERE precio IN ( 3,5,8 );

-- Selección de las piezas cuyo precio no sea igual a 3, 5 u 8
SELECT tipo, modelo, precio
FROM PIEZAS
WHERE precio NOT IN ( 3,5,8 );
```

4.3.3.5. LIKE

El operador LIKE se usa sobre todo con textos, permite obtener registros cuyo valor en un campo cumpla una condición textual. LIKE utiliza una cadena que puede contener estos símbolos:

Símbolo	Significado
%	Una serie cualquiera de caracteres
_	Un carácter cualquiera

Ejemplos:

```
-- Selección el nombre de las personas que empiezan por A
SELECT nombre
FROM PERSONAS
WHERE nombre LIKE 'A%';

-- Selección el nombre y los apellidos de las personas
cuyo primer apellido sea Jiménez, Giménez, Ximénez
SELECT nombre, apellido1, apellido2
FROM PERSONAS
WHERE apellido1 LIKE '_iménez';
```

Si queremos que en la cadena de caracteres se busquen los caracteres “%” o “_” le anteponemos el símbolo escape:

Ejemplo:

```
-- Seleccionamos el tipo, el modelo y el precio de las piezas
-- cuyo porcentaje de descuento sea 3%
SELECT tipo, modelo, precio
FROM PIEZAS
WHERE descuento LIKE '3\%' ESCAPE '\';
```

4.3.3.6. IS NULL

La cláusula IS NULL devuelve “verdadero” si una expresión contiene un nulo, y “Falso” en caso contrario. La cláusula IS NOT NULL devuelve “verdadero” si una expresión NO contiene un nulo, y “Falso” en caso contrario.

Ejemplos:

```
-- Devuelve el nombre y los apellidos de las personas que NO tienen teléfono
SELECT nombre, apellido1, apellido2
FROM PERSONAS
WHERE telefono IS NULL;

-- Devuelve el nombre y los apellidos de las personas que SÍ tienen teléfono
SELECT nombre, apellido1, apellido2
FROM PERSONAS
WHERE telefono IS NOT NULL;
```

4.3.3.7. Precedencia de operadores

A veces las expresiones que se producen en los SELECT son muy extensas y es difícil saber que parte de la expresión se evalúa primero, por ello se indica la siguiente tabla de precedencia:

Orden de precedencia	Operador
1	*(Multiplicar) / (dividir)
2	+ (Suma) - (Resta)
3	|| (Concatenación)
4	Comparaciones (>, <, !=, ...)
5	IS [NOT] NULL, [NOT]LIKE, IN
6	NOT
7	AND
8	OR

4.4. SUBCONSULTAS

Se trata de una técnica que permite utilizar el resultado de una tabla SELECT en otra consulta SELECT. Permite solucionar problemas en los que el mismo dato aparece dos veces. La sintaxis es:

```
SELECT lista_expresiones
FROM tablas
WHERE expresión OPERADOR
   ( SELECT lista_expresiones
     FROM tablas );
```

Se puede colocar el SELECT dentro de las cláusulas WHERE, HAVING o FROM. El operador puede ser >,<,>=,<=,!=, = o IN.

Ejemplo:

```
-- Obtiene los empleados cuyas pagas sean inferiores a lo que gana Martina.
SELECT nombre_empleado, paga
FROM EMPLEADOS
WHERE paga < ( SELECT paga
               FROM EMPLEADOS
               WHERE nombre_empleado='Martina');
```

Lógicamente el resultado de la subconsulta debe incluir el campo que estamos analizando.

Se pueden realizar esas subconsultas las veces que haga falta:

```
SELECT nombre_empleado, paga
FROM EMPLEADOS
WHERE paga < ( SELECT paga
            FROM EMPLEADOS
            WHERE nombre_empleado='Martina')
AND   paga > ( SELECT paga
            FROM EMPLEADOS
            WHERE nombre_empleado='Luis');
```

La última consulta obtiene los empleados cuyas pagas estén entre lo que gana Luis y lo que gana Martina. Una subconsulta que utilice los valores >,<,>=,… tiene que devolver un único valor, de otro modo ocurre un error. Pero a veces se utilizan consultas del tipo: mostrar el sueldo y nombre de los empleados cuyo sueldo supera al de cualquier empleado del departamento de ventas.

La subconsulta necesaria para ese resultado mostraría los sueldos del departamento de ventas. Pero no podremos utilizar un operador de comparación directamente ya que compararíamos un valor con muchos valores. La solución a esto es utilizar instrucciones especiales entre el operador y la consulta. Esas instrucciones son:

Instrucción	Significado
ANY	Compara con cualquier registro de la subconsulta. La instrucción es válida si hay un registro en la subconsulta que permite que la comparación sea cierta
ALL	Compara con todos los registros de la consulta. La instrucción resulta cierta si es cierta toda comparación con los registros de la subconsulta
IN	No usa comparador, ya que sirve para comprobar si un valor se encuentra en el resultado de la subconsulta
NOT IN	Comprueba si un valor no se encuentra en una subconsulta

Ejemplo:

```
-- Obtiene el empleado que más cobra.
SELECT nombre, sueldo
FROM EMPLEADOS
WHERE sueldo >= ALL ( SELECT sueldo
                FROM EMPLEADOS );
```

Ejemplo:

```
-- Obtiene los nombres de los empleados cuyos DNI están en la tabla de directivos.
-- Es decir, obtendrá el nombre de los empleados que son directivos.
SELECT nombre, sueldo
FROM EMPLEADOS
```

```
WHERE DNI IN ( SELECT DNI
               FROM DIRECTIVOS );
```

4.5. ORDENACIÓN

El orden inicial de los registros obtenidos por un SELECT guarda una relación con al orden en el que fueron introducidos. Para ordenar en base a criterios más interesantes, se utiliza la cláusula ORDER BY. En esa cláusula se coloca una lista de campos que indica la forma de ordenar. Se ordena primero por el primer campo de la lista, si hay coincidencias por el segundo, si ahí también las hay por el tercero, y así sucesivamente. Se puede colocar las palabras ASC O DESC (por defecto se toma ASC). Esas palabras significan en ascendente (de la A a la Z, de los números pequeños a los grandes) o en descendente (de la Z a la a, de los números grandes a los pequeños) respectivamente.

Sintaxis completa de SELECT:

```
SELECT * | {[DISTINCT] columna | expresión [[AS] alias], ... }
FROM nombre_tabla
[WHERE condición]
[ORDER BY columna1[{ASC|DESC}]][,columna2[{ASC|DESC}]]...;
```

Ejemplo:

```
-- Devuelve el nombre y los apellidos
-- de las personas que tienen teléfono, ordenados por
-- apellido1, luego por apellido2 y finalmente por nombre
SELECT nombre, apellido1, apellido2
FROM PERSONAS
WHERE telefono IS NOT NULL
ORDER BY apellido1, apellido2, nombre;
```

4.6. FUNCIONES

Oracle incorpora una serie de instrucciones que permiten realizar cálculos avanzados, o bien facilitar la escritura de ciertas expresiones. Todas las funciones reciben datos para poder operar (parámetros) y devuelven un resultado (que depende de los parámetros enviados a la función. Los argumentos se pasan entre paréntesis:

```
NOMBRE_FUNCIÓN [ ( parámetro1 [, parámetros2] ... ) ];
```

Si una función no precisa parámetros (como SYSDATE) no hace falta colocar los paréntesis.

Las funciones pueden ser de dos tipos:

- Funciones que operan con una sola fila
- Funciones que operan con varias filas.

En este apartado, solo veremos las primeras. Más adelante se estudiarán las que operan sobre varias filas.

Nota

Oracle proporciona una tabla llamada **DUAL** con la que se permiten hacer pruebas. Esa tabla tiene un solo campo (llamado DUMMY) y una sola fila de modo que es posible hacer pruebas.

Por ejemplo la consulta:

SELECT SQRT(5) FROM DUAL;

Muestra una tabla con el contenido de ese cálculo (la raíz cuadrada de 5). DUAL es una tabla interesante para hacer pruebas.

4.6.1. Funciones de caracteres

Para convertir el texto a mayúsculas o minúsculas:

Función	Descripción
LOWER(texto)	Convierte el texto a minúsculas (funciona con los caracteres españoles)
UPPER(texto)	Convierte el texto a mayúsculas
INITCAP(texto)	Coloca la primera letra de cada palabra en mayúsculas

En la siguiente tabla mostramos las llamadas funciones de transformación:

Función	Descripción
RTRIM(texto)	Elimina los espaciosa la derecha del texto
LTRIM(texto)	Elimina los espacios a la izquierda que posea el texto
TRIM(texto)	Elimina los espacios en blanco a la izquierda y la derecha del texto y los espacios dobles del interior.
TRIM(caracteres FROM texto)	Elimina del texto los caracteres indicados. Por ejemplo TRIM('h' FROM nombre) elimina las haches de la columna *nombre* que estén a la izquierda y a la derecha
SUBSTR(texto,n[,m])	Obtiene los *m* siguientes caracteres del texto a partir de la posición *n* (si *m* no se indica se cogen desde *n* hasta el final).
LENGTH(texto)	Obtiene el tamaño del texto
INSTR(texto, textoBuscado [,posInicial [, nAparición]])	Obtiene la posición en la que se encuentra el texto buscado en el texto inicial. Se puede empezar a buscar a partir de una posición inicial concreta e incluso indicar el número de aparición del texto buscado. Ejemplo, si buscamos la letra *a* y ponemos 2 en *nAparición*, devuelve la posición de la segunda letra a del texto). Si no lo encuentra devuelve 0
REPLACE(texto, textoABuscar, [textoReemplazo])	Buscar el texto a buscar en un determinado texto y lo cambia por el indicado como texto de reemplazo. Si no se indica texto de reemplazo, entonces está función elimina el texto a buscar
LPAD(texto, anchuraMáxima, [caracterDeRelleno]) RPAD(texto, anchuraMáxima, [caracterDeRelleno])	Rellena el texto a la izquierda (LPAD) o a la derecha (RPAD) con el carácter indicado para ocupar la anchura indicada. Si el texto es más grande que la anchura indicada, el texto se recorta. Si no se indica carácter de relleno se rellenaré el espacio marcado con espacios en blanco.
REVERSE(texto)	Invierte el texto (le da la vuelta)

4.6.2. Funciones numéricas

Funciones para redondear el número de decimales o redondear a números enteros:

Función	Descripción
ROUND(n,decimales)	Redondea el número al siguiente número con el número de decimales indicado más cercano. ROUND(8.239,2) devuelve 8.24
TRUNC(n,decimales)	Los decimales del número se cortan para que sólo aparezca el número de decimales indicado

En el siguiente cuadro mostramos la sintaxis SQL de funciones matemáticas habituales:

Función	Descripción
MOD(n1,n2)	Devuelve el resto resultado de dividir *n1* entre *n2*
POWER(valor,exponente)	Eleva el valor al exponente indicado
SQRT(n)	Calcula la raíz cuadrada de n
SIGN(n)	Devuelve 1 si *n* es positivo, cero si vale cero y -1 si es negativo
ABS(n)	Calcula el valor absoluto de *n*
EXP(n)	Calcula e^n, es decir el exponente en base *e* del número n
LN(n)	Logaritmo neperiano de *n*
LOG(n)	Logaritmo en base 10 de *n*
SIN(n)	Calcula el seno de *n* (*n* tiene que estar en radianes)
COS(n)	Calcula el coseno de *n* (*n* tiene que estar en radianes)
TAN(n)	Calcula la tangente de *n* (*n* tiene que estar en radianes)
ACOS(n)	Devuelve en radianes el arco coseno de *n*
ASIN(n)	Devuelve en radianes el arco seno de *n*
ATAN(n)	Devuelve en radianes el arco tangente de *n*
SINH(n)	Devuelve el seno hiperbólico de *n*
COSH(n)	Devuelve el coseno hiperbólico de *n*
TANH(n)	Devuelve la tangente hiperbólica de *n*

4.6.3. Funciones de fecha

Las fechas se utilizan muchísimo en todas las bases de datos. Oracle proporciona dos tipos de datos para manejar fechas, los tipos DATE y TIMESTAMP. En el primer caso se almacena una fecha concreta (que incluso puede contener la hora), en el segundo caso se almacena un instante de tiempo más concreto que puede incluir incluso fracciones de segundo. Hay que tener en cuenta que a los valores de tipo fecha se les pueden sumar números y se entendería que esta suma es de días. Si tiene decimales entonces se suman días, horas, minutos y segundos. La diferencia entre dos fechas también obtiene un número de días.

Funciones para obtener la fecha y hora actual

Función	Descripción
SYSDATE	Obtiene la fecha y hora actuales
SYSTIMESTAMP	Obtiene la fecha y hora actuales en formato TIMESTAMP

Funciones para calcular fechas:

Función	Descripción
ADD_MONTHS(fecha,n)	Añade a la fecha el número de meses indicado por *n*
MONTHS_BETWEEN(fecha1, fecha2)	Obtiene la diferencia en meses entre las dos fechas (puede ser decimal)
NEXT_DAY(fecha,día)	Indica cual es el día que corresponde a añadir a la fecha el día indicado. El día puede ser el texto '*Lunes*', '*Martes*', '*Miércoles*',... (si la configuración está en español) o el número de día de la semana (1=lunes, 2=martes,...)
LAST_DAY(fecha)	Obtiene el último día del mes al que pertenece la fecha. Devuelve un valor DATE
EXTRACT(valor FROM fecha)	Extrae un valor de una fecha concreta. El valor puede ser **day** (día), **month** (mes), **year** (año), etc.
GREATEST(fecha1, fecha2,..)	Devuelve la fecha más moderna la lista
LEAST(fecha1, fecha2,..)	Devuelve la fecha más antigua la lista
ROUND(fecha [,'formato']	Redondea la fecha al valor de aplicar el formato a la fecha. El formato puede ser: 'YEAR' Hace que la fecha refleje el año completo 'MONTH' Hace que la fecha refleje el mes completo más cercano a la fecha 'HH24' Redondea la hora a las 00:00 más cercanas 'DAY' Redondea al día más cercano
TRUNC(fecha [formato])	Igual que el anterior pero trunca la fecha en lugar de redondearla.

4.6.4. Funciones de conversión

Oracle es capaz de convertir datos automáticamente a fin de que la expresión final tenga sentido. En ese sentido son fáciles las conversiones de texto a número y viceversa.

Ejemplos:

```
-- El resultado es 8
SELECT 5+'3'
```

```
FROM DUAL;

-- El resultado es 53
SELECT 5||'3'
FROM DUAL;
```

Pero en determinadas ocasiones querremos realizar conversiones explícitas. Para hacerlo utilizaremos las funciones que se detallan a continuación.

4.6.4.1. Función de conversión TO_CHAR

Obtiene un texto a partir de un número o una fecha. En especial se utiliza con fechas (ya que de número a texto se suele utilizar de forma implícita). En el caso de las fechas se indica el formato de conversión, que es una cadena que puede incluir estos símbolos (en una cadena de texto):

Símbolo	Significado
YY	Año en formato de dos cifras
YYYY	Año en formato de cuatro cifras
MM	Mes en formato de dos cifras
MON	Las tres primeras letras del mes
MONTH	Nombre completo del mes
DY	Día de la semana en tres letras
DAY	Día completo de la semana
D	Día de la semana (del 1 al 7)
DD	Día del mes en formato de dos cifras (del 1 al 31)
DDD	Día del año
Q	Semestre
WW	Semana del año
AM	Indicador AM
PM	Indicador PM
HH12	Hora de 1 a 12
HH24	Hora de 0 a 23
MI	Minutos (0 a 59)
SS	Segundos (0 a 59)
SSSS	Segundos desde medianoche
/ . ,:;'	Posición de los separadores, donde se pongan estos símbolos aparecerán en el resultado

Ejemplo:

```
-- Si esta consulta se ejecuta el 20 de Febrero de 2014 a las 14:15 horas,
-- devuelve:  20/FEBRERO/2014, JUEVES 14:15:03
```

```
SELECT TO_CHAR(SYSDATE, 'DD/MONTH/YYYY, DAY HH:MI:SS')
FROM DUAL;
```

Para convertir números a textos se usa esta función cuando se desean características especiales. En este caso en el formato se pueden utilizar estos símbolos:

4.6.4.2. Función de conversión TO_NUMBER

Convierte textos en números. Se indica el formato de la conversión.

Símbolo	Significado
9	Posición del número
0	Posición del número (muestra ceros)
$	Formato dólar
L	Símbolo local de la moneda
S	Hace que aparezca el símbolo del signo
D	Posición del símbolo decimal (en español, la coma)
G	Posición del separador de grupo (en español el punto)

4.6.4.3. Función de conversión TO_DATE

Convierte textos en fechas. Como segundo parámetro se utilizan los códigos de formato de fechas comentados anteriormente.

4.6.5. Función DECODE

Se evalúa una expresión y se colocan a continuación pares valor,resultado de forma que si se la expresión equivale al valor, se obtiene el resultado indicado. Se puede indicar un último parámetro con el resultado a efectuar en caso de no encontrar ninguno de los valores indicados. Sintaxis:

```
DECODE (
   expresión, valor1, resultado1
         [, valor2, resultado2] ...
         [, valorPorDefecto]
);
```

Ejemplo:

```
SELECT
  DECODE (cotización, 1, salario*0.85,
                   2, salario*0.93,
                   3, salario*0.96,
                   salario)
```

```
FROM EMPLEADOS;
```

Este ejemplo es idéntico al mostrado con una expresión CASE.

4.6.6. Expresión CASE

Es una instrucción incorporada a la versión 9 de Oracle que permite establecer condiciones de salida (al estilo if-then-else de muchos lenguajes).

```
CASE expresión
  WHEN valor1 THEN resultado1
  [WHEN valor2 THEN resultado2] ...
  [ELSE resultado_por_defecto]
END;
```

El funcionamiento es el siguiente:

1. Se evalúa la expresión indicada.
2. Se comprueba si esa expresión es igual al valor del primer WHEN, de ser así se devuelve el primer resultado (cualquier valor excepto nulo).
3. Si la expresión no es igual al valor 1, entonces se comprueba si es igual al segundo. De ser así se escribe el resultado 2. De no ser así se continua con el siguiente WHEN.
4. El resultado indicado en la zona ELSE sólo se escribe si la expresión no vale ningún valor de los indicados.

Ejemplo:

```
SELECT
  CASE cotización
    WHEN 1 THEN salario*0.85
    WHEN 2 THEN salario*0.93
    WHEN 3 THEN salario*0.96
    ELSE salario
  END
FROM EMPLEADOS;
```

4.7. AGRUPACIONES

Es muy común utilizar consultas en las que se desee agrupar los datos a fin de realizar cálculos en vertical, es decir calculados a partir de datos de distintos registros. Para ello se utiliza la cláusula GROUP BY que permite indicar en base a qué registros se realiza la agrupación. Con GROUP BY la instrucción SELECT queda de esta forma:

```
SELECT lista_expresiones
FROM lista_tablas
```

[WHERE condiciones]
[GROUP BY grupos]
[HAVING condiciones_de_grupos]
[ORDER BY columnas];

En el apartado GROUP BY, se indican las columnas por las que se agrupa. La función de este apartado es crear un único registro por cada valor distinto en las columnas del grupo. Vamos a ver un ejemplo de como funciona GROUP BY. Supongamos que tenemos la siguiente tabla EXISTENCIAS:

TI	MODELO	N_ALMACEN	CANTIDAD
AR	6	1	2500
AR	6	2	5600
AR	6	3	2430
AR	9	1	250
AR	9	2	4000
AR	9	3	678
AR	15	1	5667
AR	20	3	43
BI	10	2	340
BI	10	3	23
BI	38	1	1100
BI	38	2	540
BI	38	3	

Si por ejemplo agrupamos en base a las columnas tipo y modelo, la sintaxis sería la siguiente:

SELECT tipo, modelo
FROM EXISTENCIAS
GROUP BY tipo, modelo;

Al ejecutarla, en la tabla de existencias, se creará un único registro por cada tipo y modelo distintos, generando la siguiente salida:

TI	MODELO
AR	6
AR	9
AR	15
AR	20
BI	10
BI	38

Es decir es un resumen de los datos anteriores. Pero observamos que los datos n_almacen y cantidad no están disponibles directamente ya que son distintos en los registros del mismo grupo.

Así si los hubiésemos seleccionado también en la consulta habríamos ejecutado una consulta ERRÓNEA. Es decir al ejecutar:

```
SELECT tipo, modelo, cantidad
FROM EXISTENCIAS
GROUP BY tipo, modelo;
```

Habríamos obtenido un mensaje de error.

```
ERROR en línea 1:
ORA-00979: no es una expresión GROUP BY
```

Es decir esta consulta es errónea, porque GROUP BY sólo se pueden utilizar desde funciones.

4.7.1. Funciones de cálculo con grupo (o funciones colectivas)

Lo interesante de la creación de grupos es la posibilidad de cálculo que ofrece. Para ello se utilizan las funciones que permiten trabajar con los registros de un grupo. Estas son:

Función	Significado
COUNT(*)	Cuenta los elementos de un grupo. Se utiliza el asterisco para no tener que indicar un nombre de columna concreto, el resultado es el mismo para cualquier columna
SUM(*expresión*)	Suma los valores de la expresión
AVG(*expresión*)	Calcula la media aritmética sobre la expresión indicada
MIN(*expresión*)	Mínimo valor que toma la expresión indicada
MAX(*expresión*)	Máximo valor que toma la expresión indicada
STDDEV(*expresión*)	Calcula la desviación estándar
VARIANCE(*expresión*)	Calcula la varianza

Las funciones anteriores se aplicarán sobre todos los elementos del grupo. Así, por ejemplo, podemos calcular la suma de las cantidades para cada tipo y modelo de la tabla EXISTENCIAS. (Es como si lo hiciéramos manualmente con las antiguas fichas sobre papel: primero las separaríamos en grupos poniendo juntas las que tienen el mismo tipo y modelo y luego para cada grupo sumaríamos las cantidades). La sintaxis SQL de dicha consulta quedaría:

```
SELECT tipo, modelo, SUM(cantidad)
FROM EXISTENCIAS
```

GROUP BY tipo, modelo;

Y se obtiene el siguiente resultado, en el que se suman las cantidades para cada grupo.

TI	MODELO	SUM(CANTIDAD)
AR	6	10530
AR	9	4928
AR	15	5667
AR	20	43
BI	10	363
BI	38	1740

4.7.2. Condiciones HAVING

A veces se desea restringir el resultado de una función agrupada y no aplicarla a todos los grupos. Por ejemplo, imaginemos que queremos realizar la consulta anterior, es decir queremos calcular la suma de las cantidades para cada tipo y modelo de la tabla EXISTENCIAS, pero queremos que se muestren solo los registros en los que la suma de las cantidades calculadas sean mayor que 500. si planteáramos la consulta del modo siguiente:

```
SELECT tipo, modelo, SUM(cantidad)
FROM EXISTENCIAS
WHERE SUM(cantidad) > 500
GROUP BY tipo, modelo;
```

Habríamos ejecutado una consulta ERRÓNEA.

```
ERROR  en línea 3:
ORA-00934: función de grupo no permitida aquí
```

La razón es que Oracle calcula primero el WHERE y luego los grupos; por lo que esa condición no la puede realizar al no estar establecidos los grupos. Es decir, no puede saber que grupos tienen una suma de cantidades mayor que 500 cuando todavía no ha aplicado los grupos. Por ello se utiliza la cláusula HAVING, cuya ejecución se efectúa una vez realizados los grupos. Se usaría de esta forma:

```
SELECT tipo, modelo, SUM(cantidad)
FROM EXISTENCIAS
GROUP BY tipo, modelo
HAVING SUM(cantidad) > 500;
```

Ahora bien, esto no implica que con la cláusula GROUP BY no podamos emplear un WHERE. Esta expresión puede usarse para imponer condiciones sobre las filas de la tabla antes de agrupar. Por ejemplo, la siguiente expresión es correcta:

```
SELECT tipo, modelo, SUM(cantidad)
FROM EXISTENCIAS
WHERE tipo='AR'
GROUP BY tipo, modelo
HAVING SUM(cantidad) > 500;
```

De la tabla EXISTENCIAS tomará solo aquellas filas cuyo tipo sea AR, luego agrupará según tipo y modelo y dejará sólo aquellos grupos en los que SUM(cantidad)>500 y por último mostrará tipo, modelo y la suma de las cantidades para aquellos grupos que cumplan dicha condición. En definitiva, el orden de ejecución de la consulta marca lo que se puede utilizar con WHERE y lo que se puede utilizar con HAVING.

Pasos en la ejecución de una consulta SELECT el gestor de bases de datos sigue el siguiente orden:

1. Se aplica la cláusula FROM, de manera que determina sobre que tablas se va a ejecutar la consulta.
2. Se seleccionan las filas deseadas utilizando WHERE. (Solo quedan las filas que cumplen las condiciones especificadas en el WHERE).
3. Se establecen los grupos indicados en la cláusula GROUP BY.
4. Se calculan los valores de las funciones de totales o colectivas que se especifiquen en el HAVING (COUNT, SUM, AVG,…)
5. Se filtran los registros que cumplen la cláusula HAVING
6. Se aplica la cláusula SELECT que indica las columnas que mostraremos en la consulta.
7. El resultado se ordena en base al apartado ORDER BY.

4.8. OBTENER DATOS DE MÚLTIPLES TABLAS

Es más que habitual necesitar en una consulta datos que se encuentran distribuidos en varias tablas. Las bases de datos relacionales se basan en que los datos se distribuyen en tablas que se pueden relacionar mediante un campo. Ese campo es el que permite integrar los datos de las tablas. A continuación veremos como se pueden realizar las consultas entre varias tablas. Para ello partiremos del siguiente ejemplo: Supongamos que disponemos de una tabla de EMPLEADOS cuya clave principal es el DNI y otra tabla de TAREAS que se refiere a las tareas realizadas por los empleados. Suponemos que cada empleado realizará múltiples tareas, pero que cada tarea es realizada por un único empleado. Si el diseño está bien hecho, en la tabla de TAREAS aparecerá el DNI del empleado (como clave foránea) para saber qué empleado realizó la tarea.

4.8.1. Producto cruzado o cartesiano de tablas

En el ejemplo anterior si quiere obtener una lista de los datos de las tareas y los empleados, se podría hacer de esta forma:

```
SELECT cod_tarea, descripción_tarea, dni_empleado, nombre_empleado
FROM TAREAS, EMPLEADOS;
```

Aunque la sintaxis es correcta ya que, efectivamente, en el apartado FROM se pueden indicar varias tareas separadas por comas, al ejecutarla produce un producto cruzado de las tablas. Es decir, aparecerán todos los registros de las tareas relacionados con todos los registros de empleados (y no para cada empleado sus tareas específicas). El producto cartesiano pocas veces es útil para realizar consultas. Nosotros necesitamos discriminar ese producto para que sólo aparezcan los registros de las tareas relacionadas con sus empleados correspondientes. A eso se le llama asociar tablas (join) y se ve en el siguiente apartado.

4.8.2. Asociando tablas

La forma de realizar correctamente la consulta anterior (asociado las tareas con los empleados que la realizaron) sería:

```
SELECT cod_tarea, descripción_tarea, dni_empleado, nombre_empleado
FROM TAREAS, EMPLEADOS
WHERE TAREAS.dni_empleado = EMPLEADOS.dni_empleado;
```

Nótese que se utiliza la notación tabla.columna para evitar la ambigüedad, ya que el mismo nombre de campo se puede repetir en ambas tablas. Para evitar repetir continuamente el nombre de la tabla, se puede utilizar un alias de tabla:

```
SELECT T.cod_tarea, T.descripción_tarea, E.dni_empleado, E.nombre_empleado
FROM TAREAS T, EMPLEADOS E
WHERE T.dni_empleado = E.dni_empleado;
```

A la sintaxis WHERE se le pueden añadir condiciones sin más que encadenarlas con el operador AND.

Ejemplo:

```
SELECT T.cod_tarea, T.descripción_tarea
FROM TAREAS T, EMPLEADOS E
WHERE T.dni_empleado = E.dni_empleado AND E.nombre_empleado = 'Javier';
```

Finalmente indicar que se pueden enlazar más de dos tablas a través de sus claves principales y foráneas. Por cada relación necesaria entre tablas, aparecerá una condición (igualando la clave principal y la foránea correspondiente) en el WHERE.

Ejemplo:

```
SELECT T.cod_tarea, T.descripción_tarea, E.nombre_empleado, U.nombre_utensilio
FROM TAREAS T, EMPLEADOS E, UTENSILIOS U
WHERE T.dni_empleado = E.dni_empleado AND T.cod_tarea = U.cod_tarea;
```

4.8.3. Relaciones sin igualdad

A las relaciones descritas anteriormente se las llama relaciones en igualdad (equijoins), ya que las tablas se relacionan a través de campos que contienen valores iguales en dos tablas. A veces esto no ocurre, en las tablas:

EMPLEADOS	
Empleado	Sueldo
Antonio	18000
Marta	21000
Sonia	15000

CATEGORIAS		
categoría	Sueldo mínimo	Sueldo máximo
D	6000	11999
C	12000	17999
B	18000	20999
A	20999	80000

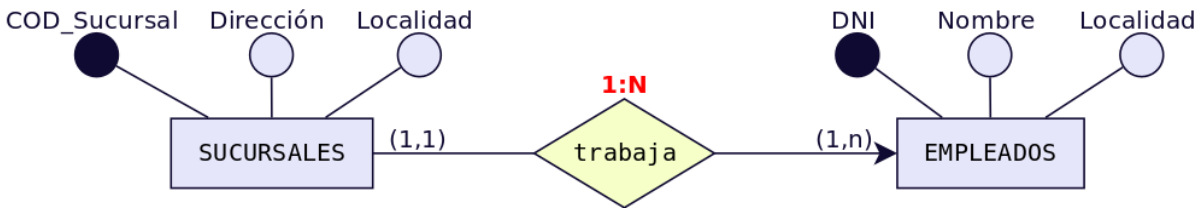
En el ejemplo anterior podríamos averiguar la categoría a la que pertenece cada empleado, pero estas tablas poseen una relación que ya no es de igualdad.

La forma sería:

```
SELECT E.empleado, E.sueldo, C.categoria
FROM EMPLEADOS E, CATEGORÍAS C
WHERE E.sueldo BETWEEN C.sueldo_mínimo AND C.sueldo_máximo;
```

4.8.4. Combinación de tablas (JOIN)

Existe otra forma más moderna e intuitiva de trabajar con varias tablas. Para ello se utiliza la clausula JOIN. Supongamos que tenemos una base de datos de una entidad bancaria. Disponemos de una tabla con sus empleados y otra tabla con sus sucursales. En una sucursal trabajan varios empleados. Los empleados viven en una localidad y trabajan en una sucursal situada en la misma localidad o en otra localidad. El esquema E-R es el siguiente:



Los datos de las tablas son:

DNI	NOMBRE	LOCALIDAD	COD_SUCURSAL

11111111A	ANA	ALMERÍA	0001
22222222B	BERNARDO	GRANADA	0001
33333333C	CARLOS	GRANADA	●
44444444D	DAVID	JEREZ	0003

COD_SUCURSAL	DIRECCIÓN	LOCALIDAD
0001	C/ ANCHA, 1	ALMERÍA
0002	C/ NUEVA, 1	GRANADA
0003	C/ CORTÉS, 33	CÁDIZ

Se observa que Ana vive en Almería y trabaja en la sucursal 0001 situada en Almería. Bernardo vive en Granada pero trabaja en la sucursal 0001 de Almería. Carlos es un empleado del que no disponemos el dato acerca de la sucursal en la que trabaja. David es un empleado que vive en Jerez de la Frontera y trabaja en la sucursal 0003 en Cádiz. Existe otra sucursal 0002 en Granada donde no aparece registrado ningún empleado.

Existen diversas formas de combinar (JOIN) las tablas según la información que deseemos obtener. Los tipos de JOIN se clasifican en:

- INNER JOIN (o simplemente JOIN): Combinación interna.
  - JOIN
  - SELF JOIN
  - NATURAL JOIN
- OUTER JOIN: Combinación externa.
  - LEFT OUTER JOIN (o simplemente LEFT JOIN)
  - RIGHT OUTER JOIN (o simplemente RIGHT JOIN)
  - FULL OUTER JOIN (o simplemente FULL JOIN)
- CROSS JOIN: Combinación cruzada.

Pasamos a continuación a explicar cada uno de ellos.

4.8.4.1. INNER JOIN

También se conoce como EQUI JOIN o combinación de igualdad. Esta combinación devuelve todas las filas de ambas tablas donde hay una coincidencia. Este tipo de unión se puede utilizar en la situación en la que sólo debemos seleccionar las filas que tienen valores comunes en las columnas que se especifican en la cláusula ON.

4.8.4.1.1. JOIN

En lugar de INNER JOIN es más frecuente encontrarlo escrito como JOIN simplemente:

Su sintaxis es:

```
SELECT TABLA1.columna1, TABLA1.columna2, ...
    TABLA2.columna1, TABLA2.columna2, ...
FROM TABLA1 JOIN TABLA2
ON TABLA1.columnaX = TABLA2.columnaY;
```

Por ejemplo, para ver los empleados con sucursal asignada:

```
SELECT E.*, S.LOCALIDAD
FROM EMPLEADOS E JOIN SUCURSALES S
ON E.COD_SUCURSAL = S.COD_SUCURSAL;
```

DNI	NOMBRE	LOCALIDAD	COD_SUCURSAL	LOCALIDAD
22222222B	BERNARDO	GRANADA	0001	ALMERÍA
11111111A	ANA	ALMERÍA	0001	ALMERÍA
44444444D	DAVID	JEREZ	0003	CÁDIZ

En esta consulta, utilizamos la combinación interna basada en la columna «COD_SUCURSAL» que es común en las tablas «EMPLEADOS» y «SUCURSALES». Esta consulta dará todas las filas de ambas tablas que tienen valores comunes en la columna «COD_SUCURSAL»

4.8.4.1.2. SELF JOIN

En algún momento podemos necesitar unir una tabla consigo mísma. Este tipo de combinación se denomina SELF JOIN. En este JOIN, necesitamos abrir dos copias de una misma tabla en la memoria. Dado que el nombre de tabla es el mismo para ambas instancias, usamos los alias de tabla para hacer copias idénticas de la misma tabla que se abran en diferentes ubicaciones de memoria.

Nota

Observa que no existe la clausula SELF JOIN, solo JOIN.

Sintaxis:

```
SELECT ALIAS1.columna1, ALIAS1.columna2, ..., ALIAS2.columna1, ...
FROM TABLA ALIAS1 JOIN TABLA ALIAS2
ON ALIAS1.columnaX = ALIAS2.columnaY;
```

Ejemplo:

```
SELECT E1.NOMBRE, E2.NOMBRE, E1.LOCALIDAD
FROM EMPLEADOS E1 JOIN EMPLEADOS E2 ON E1.LOCALIDAD = E2.LOCALIDAD;
```

NOMBRE	NOMBRE	LOCALIDAD
ANA	ANA	ALMERÍA
CARLOS	BERNARDO	GRANADA
BERNARDO	BERNARDO	GRANADA
CARLOS	CARLOS	GRANADA
BERNARDO	CARLOS	GRANADA
DAVID	DAVID	JEREZ

Esto muestra las combinaciones de los empleados que viven en la misma localidad. En este caso no es de mucha utilidad, pero el SELF JOIN puede ser muy útil en relaciones reflexivas.

4.8.4.1.3. NATURAL JOIN

NATURAL JOIN establece una relación de igualdad entre las tablas a través de los campos que tengan el mismo nombre en ambas tablas. Su sintaxis es:

```
SELECT TABLA1.columna1, TABLA1.columna2, ...
    TABLA2.columna1, TABLA2.columna2, ...
FROM TABLA1 NATURAL JOIN TABLA2;
```

En este caso no existe clausula ON puesto que se realiza la combinación teniendo en cuenta las columnas del mismo nombre. Por ejemplo:

```
SELECT *
FROM EMPLEADOS E NATURAL JOIN SUCURSALES S;
```

LOCALIDAD	COD_SUCURSAL	DNI	NOMBRE	DIRECCIÓN

ALMERÍA	0001	11111111A	ANA	C/ ANCHA, 1

En el resultado de la consulta nos aparece la combinación donde la (LOCALIDAD, COD_SUCURSAL) de EMPLEADOS es igual a (LOCALIDAD, COD_SUCURSAL) de SUCURSALES. Es decir estamos mostrando todos los empleados que tienen asignada una sucursal y dicha sucursal está en la localidad donde vive el empleado. El NATURAL JOIN elimina columnas duplicadas, por eso no aparecen los campos LOCALIDAD ni SUCURSAL duplicados. Este tipo de consulta no permite indicar estos campos en la cláusula SELECT. Por ejemplo: SELECT E.LOCALIDAD o SELECT E.COD_SUCURSAL sería incorrecto.

4.8.4.2. OUTER JOIN

La combinación externa o OUTER JOIN es muy útil cuando deseamos averiguar que campos están a NULL en un lado de la combinación. En nuestro ejemplo, podemos ver qué empleados no tienen sucursal asignada; también podemos ver que sucursales no tienen empleados asignados.

4.8.4.2.1. LEFT JOIN

También conocido como LEFT OUTER JOIN, nos permite obtener todas las filas de la primera tabla asociadas a filas de la segunda tabla. Si no existe correspondencia en la segunda tabla, dichos valores aparecen como NULL. Su sintaxis es:

```
SELECT TABLA1.columna1, TABLA1.columna2, ...
    TABLA2.columna1, TABLA2.columna2, ...
FROM TABLA1 LEFT JOIN TABLA2
ON TABLA1.columnaX = TABLA2.columnaY;
```

Por ejemplo:

```
SELECT E.*, S.LOCALIDAD
FROM EMPLEADOS E LEFT JOIN SUCURSALES S
ON E.COD_SUCURSAL = S.COD_SUCURSAL;
```

DNI	NOMBRE	LOCALIDAD	COD_SUCURSAL	LOCALIDAD
11111111A	ANA	ALMERÍA	0001	ALMERÍA
22222222B	BERNARDO	GRANADA	0001	ALMERÍA
33333333C	CARLOS	GRANADA	●	●
44444444D	DAVID	JEREZ	0003	CÁDIZ

Todos los empleados tienen una sucursal asignada salvo el empleado Carlos.

4.8.4.2.2. RIGHT JOIN

También conocido como RIGHT OUTER JOIN, nos permite obtener todas las filas de la segunda tabla asociadas a filas de la primera tabla. Si no existe correspondencia en la primera tabla, dichos valores aparecen como NULL. Su sintaxis es:

```
SELECT TABLA1.columna1, TABLA1.columna2, ...
    TABLA2.columna1, TABLA2.columna2, ...
FROM TABLA1 RIGHT JOIN TABLA2
ON TABLA1.columnaX = TABLA2.columnaY;
```

Por ejemplo:

```
SELECT E.DNI, E.NOMBRE, S.*
FROM EMPLEADOS E RIGHT JOIN SUCURSALES S
ON E.COD_SUCURSAL = S.COD_SUCURSAL;
```

DNI	NOMBRE	COD_SUCURSAL	DIRECCIÓN	LOCALIDAD
11111111A	ANA	0001	C/ ANCHA, 1	ALMERÍA
22222222B	BERNARDO	0001	C/ ANCHA, 1	ALMERÍA
44444444D	DAVID	0003	C/ CORTÉS, 33	CÁDIZ
●	●	0002	C/ NUEVA, 1	GRANADA

Todas las sucursales tienen algún empleado asignado salvo la sucursal 0002.

4.8.4.2.3. FULL JOIN

También conocido como FULL OUTER JOIN, nos permite obtener todas las filas de la primera tabla asociadas a filas de la segunda tabla. Si no existe correspondencia en alguna de las tablas, dichos valores aparecen como NULL.

Su sintaxis es:

```
SELECT TABLA1.columna1, TABLA1.columna2, ...
    TABLA2.columna1, TABLA2.columna2, ...
FROM TABLA1 FULL JOIN TABLA2
ON TABLA1.columnaX = TABLA2.columnaY;
```

Por ejemplo:

```
SELECT E.DNI, E.NOMBRE, S.COD_SUCURSAL, S.LOCALIDAD
FROM EMPLEADOS E FULL JOIN SUCURSALES S
ON E.COD_SUCURSAL = S.COD_SUCURSAL;
```

DNI	NOMBRE	COD_SUCURSAL	LOCALIDAD	
22222222B	BERNARDO	0001	ALMERÍA	
11111111A	ANA	0001	ALMERÍA	
●	●	0002	GRANADA	
44444444D	DAVID	0003	CÁDIZ	
33333333C	CARLOS	●	●	

Como puede observarse fácilmente, vemos que en la sucursal 0002 no hay ningún empleado asignado y que el empleado Carlos no tiene asignada ninguna sucursal.

4.8.4.3. CROSS JOIN

El CROSS JOIN o combinación cruzada produce el mismo resultado del producto cartesiano, es decir nos da todas las combinaciones posibles. Su sintaxis es:

```
SELECT TABLA1.columna1, TABLA1.columna2, ...
    TABLA2.columna1, TABLA2.columna2, ...
FROM TABLA1 CROSS JOIN TABLA2;
```

Por ejemplo:

```
SELECT E.DNI, E.NOMBRE, E.LOCALIDAD, S.COD_SUCURSAL, S.LOCALIDAD
FROM EMPLEADOS E CROSS JOIN SUCURSALES S;
```

DNI	NOMBRE	LOCALIDAD	COD_SUCURSAL	LOCALIDAD
11111111A	ANA	ALMERÍA	0001	ALMERÍA

11111111A	ANA	ALMERÍA	0002	GRANADA
11111111A	ANA	ALMERÍA	0003	CÁDIZ
22222222B	BERNARDO	GRANADA	0001	ALMERÍA
22222222B	BERNARDO	GRANADA	0002	GRANADA
22222222B	BERNARDO	GRANADA	0003	CÁDIZ
33333333C	CARLOS	GRANADA	0001	ALMERÍA
33333333C	CARLOS	GRANADA	0002	GRANADA
33333333C	CARLOS	GRANADA	0003	CÁDIZ
44444444D	DAVID	JEREZ	0001	ALMERÍA
44444444D	DAVID	JEREZ	0002	GRANADA
44444444D	DAVID	JEREZ	0003	CÁDIZ

En el ejemplo que estamos viendo nos mostraría 12 filas (4x3: 4 filas de empleados x 3 filas de sucursales). El primer cliente se combina con todas las sucursales. El segundo cliente igual. Y así sucesivamente. Esta combinación asocia todas las filas de la tabla izquierda con cada fila de la tabla derecha. Este tipo de unión es necesario cuando necesitamos seleccionar todas las posibles combinaciones de filas y columnas de ambas tablas. Este tipo de unión no es generalmente preferido ya que toma mucho tiempo y da un resultado enorme que no es a menudo útil

4.9. COMBINACIONES ESPECIALES

4.9.1. Uniones

La palabra UNION permite añadir el resultado de un SELECT a otro SELECT. Para ello ambas instrucciones tienen que utilizar el mismo número y tipo de columnas.

Ejemplo:

```
-- Tipos y modelos de piezas que se encuentren el almacén 1, en el 2 o en ambos.
```

```
SELECT tipo,modelo FROM existencias
WHERE n_almacen = 1

UNION

SELECT tipo,modelo FROM existencias
WHERE n_almacen = 2;
```

Nota

Observa que sólo se pone punto y coma al final.

Es decir, UNION crea una sola tabla con registros que estén presentes en cualquiera de las consultas. Si están repetidas sólo aparecen una vez, para mostrar los duplicados se utiliza UNION ALL en lugar de la palabra UNION.

4.9.2. Intersecciones

De la misma forma, la palabra INTERSECT permite unir dos consultas SELECT de modo que el resultado serán las filas que estén presentes en ambas consultas. Ejemplo: tipos y modelos de piezas que se encuentren en los almacenes 1 y 2 (en ambos).

```
-- Tipos y modelos de piezas que se encuentren en los almacenes 1 y 2 (en ambos).

SELECT tipo,modelo FROM existencias
WHERE n_almacen = 1

INTERSECT

SELECT tipo,modelo FROM existencias
WHERE n_almacen = 2;
```

Nota

Observa que sólo se pone punto y coma al final.

4.9.3. Diferencia

Con MINUS también se combinan dos consultas SELECT de forma que aparecerán los registros del primer SELECT que no estén presentes en el segundo. Ejemplo:

```
-- Tipos y modelos de piezas que se encuentren el almacén 1 y no en el 2.

SELECT tipo,modelo FROM existencias
WHERE n_almacen = 1

MINUS
```

```
SELECT tipo,modelo FROM existencias
WHERE n_almacen = 2;
```

Nota

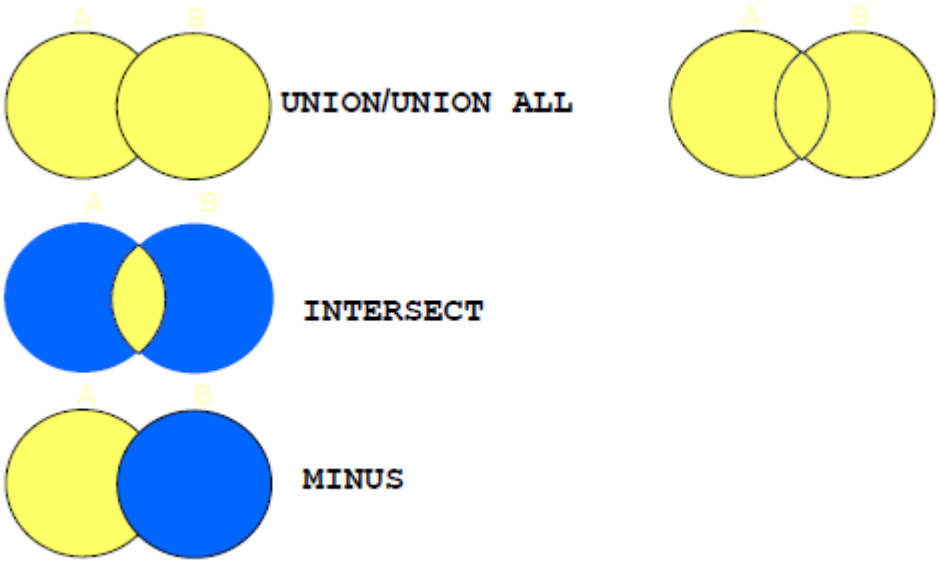
Observa que sólo se pone punto y coma al final.

Nota

La palabra MINUS es utiliza en el SGBD Oracle. El estándar SQL establece la palabra EXCEPT.

Se podrían hacer varias combinaciones anidadas (una unión a cuyo resultado se restará de otro SELECT por ejemplo), en ese caso es conveniente utilizar paréntesis para indicar qué combinación se hace primero:

```
(
SELECT ...

UNION

SELECT ...
)

MINUS

SELECT ... ;

-- Primero se hace la unión y luego la diferencia
```



4.10. CONSULTA DE VISTAS

A efectos de uso, la consulta de una vista es idéntica a la consulta de una tabla.
Supongamos que tenemos la siguiente vista:

```
CREATE VIEW RESUMEN
  -- a continuación indicamos los alias
  (id_localidad, localidad, poblacion,
   n_provincia, provincia, superficie,
   id_comunidad, comunidad)
AS
  SELECT L.IdLocalidad, L.Nombre, L.Poblacion,
         P.IdProvincia, P.Nombre, P.Superficie,
         C.IdComunidad, C.Nombre
  FROM LOCALIDADES L JOIN PROVINCIAS P ON L.IdProvincia = P.IdProvincia
                     JOIN COMUNIDADES C ON P.IdComunidad = C.IdComunidad;
```

Podemos consultar ahora sobre la vista como si de una tabla se tratase

```
SELECT DISTINCT (provincia, comunidad) FROM resumen;
```

« 4. CONSULTA DE BASES DE DATOS. LENGUAJE DE MANIPULACIÓN DE DATOS :: Contenidos :: 4.11. ACTIVIDADES PROPUESTAS »

5. MODIFICACIÓN DE BASES DE DATOS. LENGUAJE DE MANIPULACIÓN DE DATOS

5.1. INTRODUCCIÓN

En esta unidad veremos 3 sentencias SQL DML para la modificación de datos. Además aprenderemos el funcionamiento de las transacciones y realizaremos una introducción al lenguaje **PL/SQL**.

5.2. LENGUAJE DE MANIPULACIÓN DE DATOS: DML

Una vez que se ha creado de forma conveniente las tablas, el siguiente paso consiste en insertar datos en ellas, es decir, añadir tuplas. Durante la vida de la base de datos será necesario, además, borrar determinadas tuplas o modificar los valores que contienen. Los comandos de SQL que se van a estudiar en este apartado son **INSERT, UPDATE y DELETE**. Estos comandos pertenecen al **DML**.

5.2.1. Inserción de datos

El comando INSERT de SQL permite introducir datos en una tabla o en una vista de la base de datos. La sintaxis del comando es la siguiente:

```
INSERT INTO {nombre_tabla | nombre_vista } [(columna1 [, columna2]...)]
VALUES (valor1 [, valor2] ... );
```

Indicando la tabla se añaden los datos que se especifiquen tras el apartado VALUES en un nuevo registro. Los valores deben corresponderse con el orden de las columnas. Si no es así se puede indicar tras el nombre de la tabla y entre paréntesis.
Ejemplos:
Supongamos que tenemos el siguiente diseño físico de una tabla:

```
CREATE TABLE EMPLEADOS (
 COD      NUMBER(2)    PRIMARY KEY,
 NOMBRE    VARCHAR2(50) NOT NULL,
 LOCALIDAD VARCHAR2(50) DEFAULT 'Écija',
 FECHANAC  DATE
);
```

La forma más habitual de introducir datos es la siguiente:

```
INSERT INTO EMPLEADOS VALUES (1, 'Pepe', 'Osuna', '01/01/1970');
INSERT INTO EMPLEADOS VALUES (2, 'Juan', DEFAULT, NULL);
INSERT INTO EMPLEADOS VALUES (3, 'Sara', NULL, NULL);
```

Es obligatorio introducir valores para los campos COD y NOMBRE. Dichos campos no pueden tener valor NULL. Podemos insertar sólo el valor de ciertos campos. En este caso hay que indicar los campos a insertar y el orden en el que los introducimos:

```
INSERT INTO EMPLEADOS(NOMBRE, COD) VALUES ('Ana', 5);
```

Inserción de datos obtenidos de una consulta

También es posible insertar datos en una tabla que hayan sido obtenidos de una consulta realizada a otra tabla/vista u otras tablas/vistas. Su forma es:

```
INSERT INTO tabla
SELECT …
```

Debe respetarse lo dicho anteriormente respecto a los campos. La consulta SELECT debe devolver la misma cantidad y tipo de campos que los definidos en la tabla.
Por ejemplo, suponiendo que disponemos de una tabla SOLICITANTES con el siguiente diseño:

```
CREATE TABLE SOLICITANTES (
 NUM       NUMBER(2) PRIMARY KEY,
 NOMBRE     VARCHAR2(50),
 CIUDAD     VARCHAR2(50),
 NACIMIENTO DATE,
 ESTUDIOS   VARCHAR2(50)
);
```

```
INSERT INTO EMPLEADOS
SELECT NUM, NOMBRE, CIUDAD, NACIMIENTO
FROM SOLICITANTES
WHERE ESTUDIOS='CFGS ASIR';
```

También podemos indicar los campos a insertar, teniendo en cuenta que, en este caso los campos COD y NOMBRE de la tabla EMPLEADO no aceptan valores NULL, por tanto es obligatorio introducir valores para ellos:

```
INSERT INTO EMPLEADOS(FECHANAC, NOMBRE, COD)
SELECT NACIMIENTO, NOMBRE, NUM
FROM SOLICITANTES
WHERE ESTUDIOS='CFGS ASIR';
```

5.2.2. Modificación de datos

Para la modificación de registros dentro de una tabla o vista se utiliza el comando UPDATE. La sintaxis del comando es la siguiente:

```
UPDATE {nombre_tabla | nombre_vista}
SET columna1=valor1 [, columna2=valor2] ...
[WHERE condición];
```

Se modifican las columnas indicadas en el apartado SET con los valores indicados. La cláusula WHERE permite especificar qué registros serán modificados.

Ejemplos:

```
-- Ponemos todos los nombres a mayúsculas
-- y todas las localidades a Estepa
UPDATE EMPLEADOS
SET NOMBRE=UPPER(NOMBRE), LOCALIDAD='Estepa';
```

```
-- Para los empleados que nacieron a partir de 1970
-- ponemos nombres con inicial mayúscula y localidades Marchena
UPDATE EMPLEADOS
SET NOMBRE=INITCAP(NOMBRE), LOCALIDAD='Marchena'
WHERE FECHANAC >= '01/01/1970';
```

Actualización de datos usando una subconsulta

También se admiten subconsultas. Por ejemplo:

```
UPDATE empleados
SET sueldo=sueldo*1.10
WHERE id_seccion = (SELECT id_seccion FROM secciones
                    WHERE nom_seccion='Producción');
```

WHERE=SIEMPRE 1 VALOR

SI ES CON ANY,ALL… SE PUEDE PERMITIR MAS DE 1 VALOR

Esta instrucción aumenta un 10% el sueldo de los empleados que están dados de alta en la sección llamada Producción.

5.2.3. Eliminación de datos

Es más sencilla que el resto, elimina los registros de la tabla que cumplan la condición indicada. Se realiza mediante la instrucción DELETE:

```
DELETE [ FROM ] {nombre_tabla|nombre_vista}
[WHERE condición] ;
```

Ejemplos:

```
-- Borramos empleados de Estepa
DELETE EMPLEADOS
WHERE LOCALIDAD='Estepa';

-- Borramos empleados cuya fecha de nacimiento sea anterior a 1970
-- y localidad sea Osuna
DELETE EMPLEADOS
WHERE FECHANAC < '01/01/1970' AND LOCALIDAD = 'Osuna';

-- Borramos TODOS los empleados;
DELETE EMPLEADOS;
```

Hay que tener en cuenta que el borrado de un registro no puede provocar fallos de integridad y que la opción de integridad ON DELETE CASCADE (clave secundaria o foránea) hace que no sólo se borren los registros indicados sino todos los relacionados. En la práctica esto significa que no se pueden borrar registros cuya clave primaria sea referenciada por alguna clave foránea en otra tabla, a no ser que dicha tabla secundaria tenga activada la clausula ON DELETE CASCADE en su clave foránea, en cuyo caso se borraría el/los registro/s de la tabla principal y los registros de tabla secundaria cuya clave foránea coincide con la clave primaria eliminada en la tabla primera.

Eliminación de datos usando una subconsulta

Al igual que en el caso de las instrucciones INSERT o SELECT, DELETE dispone de cláusula WHERE y en dicha cláusula podemos utilizar subconsultas. Por ejemplo:

```
DELETE empleados
WHERE id_empleado IN (SELECT id_empleado FROM operarios);
```

En este caso se trata de una subconsulta creada con el operador IN, se eliminarán los empleados cuyo identificador esté dentro de la tabla operarios.

5.2.4. Las Vistas - Ejecución de comandos DML sobre vistas.

Las instrucciones DML ejecutadas sobre las vistas permiten añadir o modificar los datos de las tablas relacionados con las filas de la vista. Ahora bien, no es posible ejecutar instrucciones DML sobre vistas que:

Utilicen funciones de grupo (SUM, AVG,…)

Usen GROUP BY o DISTINCT

Posean columnas con cálculos (P. ej: PRECIO * 1.16)

Además no se pueden añadir datos a una vista si en las tablas referencias en la consulta SELECT hay campos NOT NULL que no aparecen en la consulta (es lógico ya que al añadir el dato se tendría que añadir el registro colocando el valor NULL en el campo).

Si tenemos la siguiente vista:

```
CREATE VIEW resumen (id_localidad, localidad, poblacion,
                     n_provincia, provincia, superficie,
                     id_comunidad, comunidad)
AS SELECT L.IdLocalidad, L.Nombre, L.Poblacion,
          P.IdProvincia, P.Nombre, P.Superficie,
          C.IdComunidad, C.Nombre
   FROM LOCALIDADES L JOIN PROVINCIAS P ON L.IdProvincia=P.IdProvincia
                      JOIN COMUNIDADES C ON P.IdComunidad=C.IdComunidad;
```

Si realizamos la siguiente inserción

```
INSERT INTO resumen (id_localidad, localidad, poblacion)
VALUES (10000, 'Sevilla', 750000);
```

Se producirá un error, puesto que estamos insertando un registro dentro de la vista donde muchos de sus campos no tienen especificado un valor y por tanto serán insertados a NULL. El problema es que no puede insertarse un NULL en n_provincia ni id_comunidad puesto que son claves primarias de las tablas subyacentes PROVINCIAS y COMUNIDADES. La solución al problema anterior se soluciona creando un disparador (trigger) de sustitución, que veremos en el apartado de triggers.

5.3. GESTIÓN DE TRANSACCIONES

En términos teóricos, una transacción es un conjunto de tareas relacionadas que se realizan de forma satisfactoria o incorrecta como una unidad. En términos de procesamiento, las transacciones se confirman o se anulan. Para que una transacción se confirme se debe garantizar la permanencia de los cambios efectuados en los datos. Los cambios deben conservarse aunque el sistema se bloquee o tengan lugar otros eventos imprevistos. Existen 4 propiedades necesarias, que son conocidas como **propiedades ACID**:

atomicidad (Atomicity)

coherencia (Consistency)

aislamiento (Isolation)

permanencia (Durability).

Estas propiedades garantizan un comportamiento predecible, reforzando la función de las transacciones como proposiciones de todo o nada.

Atomicidad: Una transacción es una unidad de trabajo el cual se realiza en su totalidad o no se realiza en ningún caso. Las operaciones asociadas a una transacción comparten normalmente un objetivo común y son interdependientes. Si el sistema ejecutase únicamente una parte de las operaciones, podría poner en peligro el objetivo final de la transacción.

Coherencia: Una transacción es una unidad de integridad porque mantiene la coherencia de los datos, transformando un estado coherente de datos en otro estado de datos igualmente coherente.

Aislamiento: Una transacción es una unidad de aislamiento, permitiendo que transacciones concurrentes se comporten como si cada una fuera la única transacción que se ejecuta en el sistema. El aislamiento requiere que parezca que cada transacción sea la única que manipula el almacén de datos, aunque se puedan estar ejecutando otras transacciones al mismo tiempo. Una transacción nunca debe ver las fases intermedias de otra transacción.

Permanencia: Una transacción también es una unidad de recuperación. Si una transacción se realiza satisfactoriamente, el sistema garantiza que sus actualizaciones se mantienen aunque el equipo falle inmediatamente después de la confirmación. El registro especializado permite que el procedimiento de reinicio del sistema complete las operaciones no finalizadas, garantizando la permanencia de la transacción.

En términos más prácticos, una transacción está formada por una serie de instrucciones DML. Una transacción comienza con la primera instrucción DML que se ejecute y finaliza con una operación COMMIT (si la transacción se confirma) o una operación ROLLBACK (si la operación se cancela). Hay que tener en cuenta que cualquier instrucción DDL o DCL da lugar a un COMMIT implícito, es decir todas las instrucciones DML ejecutadas hasta ese instante pasan a ser definitivas.

Para poder hacer uso de transacciones en SQL*Plus debemos tener desactivado el modo AUTOCOMMIT. Podemos ver su estado con la orden:

```
SHOW AUTOCOMMIT
```

Para desactivar dicho modo, usamos la orden:

```
SET AUTOCOMMIT OFF
```

5.3.1. COMMIT

La instrucción COMMIT hace que los cambios realizados por la transacción sean definitivos, irrevocables. Se dice que tenemos una **transacción confirmada**. Sólo se debe utilizar si estamos de acuerdo con los cambios, conviene asegurarse mucho antes de realizar el COMMIT ya que las instrucciones ejecutadas pueden afectar a miles de registros. Además el cierre correcto de la sesión da lugar a un COMMIT, aunque siempre conviene ejecutar explícitamente esta instrucción a fin de asegurarnos de lo que hacemos.

5.3.2. ROLLBACK

Esta instrucción regresa a la instrucción anterior al inicio de la transacción, normalmente el último COMMIT, la última instrucción DDL o DCL o al inicio de sesión. Anula definitivamente los cambios, por lo que conviene también asegurarse de esta operación. Un abandono de sesión incorrecto o un problema de comunicación o de caída del sistema dan lugar a un ROLLBACK implícito.

5.3.3. SAVEPOINT

Esta instrucción permite establecer un punto de ruptura. El problema de la combinación ROLLBACK/COMMIT es que un COMMIT acepta todo y un ROLLBACK anula todo. SAVEPOINT permite señalar un punto intermedio entre el inicio de la transacción y la situación actual. Su sintaxis es:

```
-- ... instrucciones DML ...

SAVEPOINT nombre

-- ... instrucciones DML ...
```

Para regresar a un punto de ruptura concreto se utiliza ROLLBACK TO SAVEPOINT seguido del nombre dado al punto de ruptura. También es posible hacer ROLLBACK TO nombre de punto de ruptura. Cuando se vuelve a un punto marcado, las instrucciones que siguieron a esa marca se anulan definitivamente.

Ejemplo de uso:

```
SET AUTOCOMMIT OFF;

CREATE TABLE T (FECHA DATE);

INSERT INTO T VALUES ('01/01/2017');
INSERT INTO T VALUES ('01/02/2017');

SAVEPOINT febrero;

INSERT INTO T VALUES ('01/03/2017');
```

```
INSERT INTO T VALUES ('01/04/2017');

SAVEPOINT abril;

INSERT INTO T VALUES ('01/05/2017');

ROLLBACK TO febrero;
-- También puede escribirse ROLLBACK TO SAVEPOINT febrero;
-- En este ejemplo sólo se guardan en la tabla
-- los 2 primeros registros o filas.
```

5.3.4. Estado de los datos durante la transacción

Si se inicia una transacción usando comandos DML hay que tener en cuenta que:
Se puede volver a la instrucción anterior a la transacción cuando se desee.
Las instrucciones de consulta SELECT realizadas por el usuario que inició la transacción muestran los datos ya modificados por las instrucciones DML.
El resto de usuarios ven los datos tal cual estaban antes de la transacción, de hecho los registros afectados por la transacción aparecen bloqueados hasta que la transacción finalice. Esos usuarios no podrán modificar los valores de dichos registros.
Tras la transacción todos los usuarios ven los datos tal cual quedan tras el fin de transacción. Los bloqueos son liberados y los puntos de ruptura borrados.

5.3.5. Concurrencia de varias transacciones (Bloqueos)

Cuando se realizan varias transacciones de forma simultánea, pueden darse diversas situaciones en el acceso concurrente a los datos, es decir, cuando se accede a un mismo dato en dos transacciones distintas. Estas situaciones son:
Lectura sucia (Dirty Read). Una transacción lee datos que han sido escritos por otra transacción que aún no se ha confirmado.
Lectura no repetible (Non-repeateable Read). Una transacción vuelve a leer los datos que ha leído anteriormente y descubre que otra transacción confirmada ha modificado o eliminado los datos.
Lectura fantasma (Phantom Read). Una transacción vuelve a ejecutar una consulta que devuelve un conjunto de filas que satisface una condición de búsqueda y descubre que otra transacción confirmada ha insertado filas adicionales que satisfacen la condición.
Para una mejor gestión de estas situaciones debemos indicar el nivel de aislamiento que deseamos. De las cuatro propiedades de ACID de un SGBD, la **propiedad de aislamiento** es la más laxa. Un nivel de aislamiento bajo aumenta la capacidad de muchos usuarios para acceder a los mismos datos al mismo tiempo, pero también aumenta el número de efectos de concurrencia (como lecturas sucias). Un mayor nivel de aislamiento puede dar como resultado

una pérdida de concurrencia y el aumento de las posibilidades de que una transacción bloquee a otra.

Podemos solicitar al SGBD cuatro niveles de aislamiento. De menor a mayor nivel de aislamiento, tenemos:

READ UNCOMMITTED (Lectura no confirmada). Las sentencias SELECT son efectuadas sin realizar bloqueos, por tanto, todos los cambios hechos por una transacción pueden verlos las otras transacciones. Permite que sucedan las 3 situaciones indicadas previamente: lecturas fantasma, no repetibles y sucias.

READ COMMITTED (Lectura confirmada). Los datos leídos por una transacción pueden ser modificados por otras transacciones. Se pueden dar lectuas fantasma y lecturas no repetibles.

REPEATEABLE READ (Lectura repetible). Consiste en que ningún registro leído con un SELECT se puede cambiar en otra transacción. Solo pueden darse lecturas fantasma.

SERIALIZABLE. Las transacciones ocurren de forma totalmente aislada a otras transacciones. Se bloquean las transacciones de tal manera que ocurren unas detrás de otras, sin capacidad de concurrencia. El SGBD las ejecuta concurrentemente si puede asegurar que no hay conflicto con el acceso a los datos.

Nivel de aislamiento y Lecturas

Nivel de aislamiento	Lecturas sucias	Lecturas no repetibles	Lecturas fantasma
READ UNCOMMITTED	SÍ	SÍ	SÍ
READ COMMITTED	NO	SÍ	SÍ
REPEATEABLE READ	NO	NO	SÍ
SERIALIZABLE	NO	NO	NO

Internamente el SGBD proporciona dicho nivel de aislamiento mediante **bloqueos** en los datos.

En Oracle, el nivel por defecto es **READ COMMITED**. Además de éste, solo permite **SERIALIZABLE**. Se puede cambiar ejecutando el comando:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

6 Transacciones

6.1 Definición

Una transacción SQL es un conjunto de sentencias SQL que se ejecutan formando una unidad lógica de trabajo (*LUW* del inglés *Logic Unit of Work*), es decir, en forma indivisible o atómica.

Una transacción SQL finaliza con un `COMMIT`, para aceptar todos los cambios que la transacción ha realizado en la base de datos, o un `ROLLBACK` para deshacerlos.

MySQL nos permite realizar transacciones en sus tablas si hacemos uso del motor de almacenamiento InnoDB (*MyISAM* no permite el uso de transacciones).

El uso de transacciones nos permite realizar operaciones de forma segura y recuperar datos si se produce algún fallo en el servidor durante la transacción, pero por otro lado las transacciones pueden aumentar el tiempo de ejecución de las instrucciones.

Las transacciones deben cumplir las cuatro propiedades ACID.

6.2 `AUTOCOMMIT`

Algunos Sistemas Gestores de Bases de Datos, como MySQL (si trabajamos con el motor InnoDB) tienen activada por defecto la variable `AUTOCOMMIT`. Esto quiere decir que automáticamente se aceptan todos los cambios realizados después de la ejecución de una sentencia SQL y no es posible deshacerlos.

Aunque la variable `AUTOCOMMIT` está activada por defecto al inicio de una sesión SQL, podemos configurarlo para indicar si queremos trabajar con transacciones implícitas o explíticas.

Podemos consultar el valor actual de `AUTOCOMMIT` haciendo:

```
SELECT @@AUTOCOMMIT;
```

Para desactivar la variable `AUTOCOMMIT` hacemos:

```
SET AUTOCOMMIT = 0;
```

Si hacemos esto siempre tendríamos una transacción abierta y los cambios sólo se aplicarían en la base de datos ejecutando la sentencia `COMMIT` de forma explícita.

Para activar la variable `AUTOCOMMIT` hacemos:

```
SET AUTOCOMMIT = 1;
```

Para poder trabajar con transacciones en MySQL es necesario utilizar InnoDB.

Se recomienda la lectura del siguiente documento SQL Transactions.

6.3 START TRANSACTION, COMMIT y ROLLBACK

Los pasos para realizar una transacción en MySQL son los siguientes:

1. Indicar que vamos a realizar una transacción con la sentencia `START TRANSACTION`, `BEGIN` o `BEGIN WORK`.
2. Realizar las operaciones de manipulación de datos sobre la base datos (insertar, actualizar o borrar filas).
3. Si las operaciones se han realizado correctamente y queremos que los cambios se apliquen de forma permanente sobre la base de datos usaremos la sentencia `COMMIT`. Sin embargo, si durante las operaciones ocurre algún error y no queremos aplicar los cambios realizados podemos deshacerlos con la setencia `ROLLBACK`.

A continuación se muestra la sintaxis que aparece en la documentación oficial para realizar transacciones en MySQL.

```
START TRANSACTION
    [transaction_characteristic [, transaction_characteristic] ...]

transaction_characteristic: {
    WITH CONSISTENT SNAPSHOT
  | READ WRITE
  | READ ONLY
}

BEGIN [WORK]
COMMIT [WORK] [AND [NO] CHAIN] [[NO] RELEASE]
ROLLBACK [WORK] [AND [NO] CHAIN] [[NO] RELEASE]

SET autocommit = {0 | 1}
```

Ejemplo 1:

```
START TRANSACTION;
SELECT @A:=SUM(salary) FROM table1 WHERE type=1;
UPDATE table2 SET summary=@A WHERE type=1;

COMMIT;
```

1 Triggers, procedimientos y funciones en MySQL

En esta unidad vamos a estudiar los procedimientos, funciones y *triggers* de MySQL, que son objetos que contienen código SQL y se almacenan asociados a una base de datos.

- Procedimiento almacenado: Es un objeto que se crea con la sentencia `CREATE PROCEDURE` y se invoca con la sentencia `CALL`. Un procedimiento puede tener cero o muchos parámetros de entrada y cero o muchos parámetros de salida.
- Función almacenada: Es un objeto que se crea con la sentencia `CREATE FUNCTION` y se invoca con la sentencia `SELECT` o dentro de una expresión. Una función puede tener cero o muchos parámetros de entrada y siempre devuelve un valor, asociado al nombre de la función.
- Trigger: Es un objeto que se crea con la sentencia `CREATE TRIGGER` y tiene que estar asociado a una tabla. Un *trigger* se activa cuando ocurre un evento de inserción, actualización o borrado, sobre la tabla a la que está asociado.

1.1 Procedimientos

Un procedimiento almacenado es un conjunto de instrucciones SQL que se almacena asociado a una base de datos. Es un objeto que se crea con la sentencia `CREATE PROCEDURE` y se invoca con la sentencia `CALL`. Un procedimiento puede tener cero o muchos parámetros de entrada y cero o muchos parámetros de salida.

1.1.1 Sintaxis

```
CREATE
    [DEFINER = user]
    PROCEDURE sp_name ([proc_parameter[,...]])
    [characteristic ...] routine_body


proc_parameter:
    [ IN | OUT | INOUT ] param_name type


type:
    Any valid MySQL data type

characteristic: {
    COMMENT 'string'
  | LANGUAGE SQL
  | [NOT] DETERMINISTIC
  | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
  | SQL SECURITY { DEFINER | INVOKER }
}

routine_body:
    Valid SQL routine statement
```

`routine_body` consiste en una sentencia de rutina SQL válida. Puede ser una declaración simple como SELECT o INSERT, o una declaración compuesta escrita usando BEGIN y END. Las sentencias compuestas pueden contener declaraciones, bucles y otras sentencias de estructura de control. En la práctica, los procedimientos almacenados tienden a usar declaraciones compuestas

Puede encontrar más información en la documentación oficial de MySQL.

1.1.2 DELIMITER

Para definir un procedimiento almacenado es necesario modificar temporalmente el carácter separador que se utiliza para delimitar las sentencias SQL.

El carácter separador que se utiliza por defecto en SQL es el punto y coma (`;`). En los ejemplos que vamos a realizar en esta unidad vamos a utilizar los caracteres `//` para delimitar las instrucciones SQL, pero es posible utilizar cualquier otro carácter.

Ejemplo:

En este ejemplo estamos configurando los caracteres `//` como los separadores entre las sentencias SQL.

```
DELIMITER //
```

En este ejemplo volvemos a configurar que el carácter separador es el punto y coma.

```
DELIMITER ;
```

1.1.3 Parámetros de entrada, salida y entrada/salida

En los procedimientos almacenados podemos tener tres tipos de parámetros:

- Entrada: Se indican poniendo la palabra reservada `IN` delante del nombre del parámetro. Estos parámetros no pueden cambiar su valor dentro del procedimiento, es decir, cuando el procedimiento finalice estos parámetros tendrán el mismo valor que tenían cuando se hizo la llamada al procedimiento. En programación sería equivalente al paso por valor de un parámetro.
- Salida: Se indican poniendo la palabra reservada `OUT` delante del nombre del parametro. Estos parámetros cambian su valor dentro del procedimiento. Cuando se hace la llamada al procedimiento empiezan con un valor inicial y cuando finaliza la ejecución del procedimiento pueden terminar con otro valor diferente. En programación sería equivalente al paso por referencia de un parámetro.
- Entrada/Salida: Es una combinación de los tipos `IN` y `OUT`. Estos parámetros se indican poniendo la palabra reservada `IN/OUT` delante del nombre del parámetro.

Ejemplo 1:

En este ejemplo, se muestra la cabecera de un procedimiento llamado `listar_productos` que sólo tiene el parámetro `gama` que es de entrada (`IN`).

```
CREATE PROCEDURE listar_productos(IN gama VARCHAR(50))
```

Ejemplo 2:

Aquí se muestra la cabecera de un procedimiento llamado `contar_productos` que tiene el parámetro `gama` de entrada (`IN`) y el parámetro `total` de salida (`OUT`).

```
CREATE PROCEDURE contar_productos(IN gama VARCHAR(50), OUT total INT UNSIGNED)
```

1.1.4 Ejemplo de un procedimiento con parámetros de entrada

Escriba un procedimiento llamado listar_productos que reciba como entrada el nombre de la gama y muestre un listado de todos los productos que existen dentro de esa gama. Este procedimiento no devuelve ningún parámetro de salida, lo que hace es mostrar el listado de los productos.

```
DELIMITER //
DROP PROCEDURE IF EXISTS listar_productos//
CREATE PROCEDURE listar_productos(IN gama VARCHAR(50))
BEGIN
  SELECT *
  FROM producto
  WHERE producto.gama = gama;
END

//
```

1.1.5 Llamada de procedimientos con `CALL`

Para hacer la llamada a un procedimiento almacenado se utiliza la palabra reservada `CALL`.

Ejemplo:

```
DELIMITER ;
CALL listar_productos('Herramientas');

SELECT * FROM producto;
```

1.1.6 Ejemplos de procedimientos con parámetros de salida

Ejemplo 1:

Escriba un procedimiento llamado `contar_productos` que reciba como entrada el nombre de la gama y devuelva el número de productos que existen dentro de esa gama. Resuelva el ejercicio de dos formas distintas, utilizando `SET` y `SELECT ... INTO`.

```
-- Solución 1. Utilizando SET
DELIMITER //
DROP PROCEDURE IF EXISTS contar_productos//
CREATE PROCEDURE contar_productos(IN gama VARCHAR(50), OUT total INT UNSIGNED)
BEGIN
  SET total = (
    SELECT COUNT(*)
    FROM producto
    WHERE producto.gama = gama);
END
//

DELIMITER ;
CALL contar_productos('Herramientas', @total);
SELECT @total;

-- Solución 2. Utilizando SELECT ... INTO
DELIMITER //
DROP PROCEDURE IF EXISTS contar_productos//
CREATE PROCEDURE contar_productos(IN gama VARCHAR(50), OUT total INT UNSIGNED)
BEGIN
  SELECT COUNT(*)
```

```
  INTO total
  FROM producto
  WHERE producto.gama = gama;
END
//

DELIMITER ;
CALL contar_productos('Herramientas', @total);

SELECT @total;
```

Ejemplo 2:

Escribe un procedimiento que se llame `calcular_max_min_media`, que reciba como parámetro de entrada el nombre de la gama de un producto y devuelva como salida tres parámetros. El precio máximo, el precio mínimo y la media de los productos que existen en esa gama. Resuelva el ejercicio de dos formas distintas, utilizando `SET` y `SELECT ... INTO`.

```
-- Solucioń 1. Utilizando SET
DELIMITER //
DROP PROCEDURE IF EXISTS calcular_max_min_media//
CREATE PROCEDURE calcular_max_min_media(
  IN gama VARCHAR(50),
    OUT maximo DECIMAL(15, 2),
  OUT minimo DECIMAL(15, 2),
    OUT media DECIMAL(15, 2)
)
BEGIN
  SET maximo = (
    SELECT MAX(precio_venta)
    FROM producto
        WHERE producto.gama = gama);

  SET minimo = (
    SELECT MIN(precio_venta)
    FROM producto
        WHERE producto.gama = gama);

  SET media = (
    SELECT AVG(precio_venta)
    FROM producto
        WHERE producto.gama = gama);
END
//

DELIMITER ;
CALL calcular_max_min_media('Herramientas', @maximo, @minimo, @media);
SELECT @maximo, @minimo, @media;

-- Solucioń 2. Utilizando SELECT ... INTO
DELIMITER //
DROP PROCEDURE IF EXISTS calcular_max_min_media//
CREATE PROCEDURE calcular_max_min_media(
  IN gama VARCHAR(50),
  OUT maximo DECIMAL(15, 2),
  OUT minimo DECIMAL(15, 2),
  OUT media DECIMAL(15, 2)
)
BEGIN
  SELECT
    MAX(precio_venta),
    MIN(precio_venta),
    AVG(precio_venta)
  INTO maximo, minimo, media
  FROM producto
```

```
  WHERE producto.gama = gama;
END
//

DELIMITER ;
CALL calcular_max_min_media('Herramientas', @maximo, @minimo, @media);

SELECT @maximo, @minimo, @media;
```

1.2 Funciones

Una función almacenada es un conjunto de instrucciones SQL que se almacena asociado a una base de datos. Es un objeto que se crea con la sentencia `CREATE FUNCTION` y se invoca con la sentencia `SELECT` o dentro de una expresión. Una función puede tener o no parámetros de entrada y siempre devuelve un valor de un tipo de dato.

1.2.1 Sintaxis

```
CREATE
    [DEFINER = user]
    FUNCTION sp_name ([func_parameter[,...]])
    RETURNS type
    [characteristic ...] routine_body

func_parameter:
    param_name type

type:
    Any valid MySQL data type

characteristic: {
    COMMENT 'string'
  | LANGUAGE SQL
  | [NOT] DETERMINISTIC
  | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
  | SQL SECURITY { DEFINER | INVOKER }
}

routine_body:
    Valid SQL routine statement
```

`routine body` consiste en una sentencia de rutina SQL válida. Puede ser una declaración simple como SELECT o INSERT, o una declaración compuesta escrita usando BEGIN y END. Las sentencias compuestas pueden contener declaraciones, bucles y otras sentencias de estructura de control. En la práctica, las funciones almacenadas tienden a usar declaraciones compuestas, a menos que el cuerpo consista en una sola declaración de RETURN.

Puede encontrar más información en la documentación oficial de MySQL.

1.2.2 Parámetros de entrada

En una función todos los parámetros son de entrada, por lo tanto, no será necesario utilizar la palabra reservada `IN` delante del nombre de los parámetros.

Ejemplo:

A continuación se muestra la cabecera de la función `contar_productos` que tiene un parámetro de entrada llamado `gama`.

```
CREATE FUNCTION contar_productos(gama VARCHAR(50))
```

1.2.3 Resultado de salida

Una función siempre devolverá un valor de salida de un tipo de dato. En la definición de la cabecera de la función hay que definir el tipo de dato que devuelve con la palabra reservada `RETURNS` y en el cuerpo de la función debemos incluir la palabra reservada `RETURN` para devolver el valor de la función.

Ejemplo:

En este ejemplo se muestra una definición de una función donde se puede ver el uso de las palabras reservadas `RETURNS` y `RETURN`.

```
DELIMITER //
CREATE FUNCTION hello (s CHAR(20))
RETURNS CHAR(50) DETERMINISTIC
       RETURN CONCAT('Hello, ',s,'!');
//
Query OK, 0 rows affected (0.00 sec)

DELIMITER ;
SELECT hello('world');
      +----------------+
      | hello('world') |
      +----------------+
      | Hello, world!  |
      +----------------+
      1 row in set (0.00 sec)

//
```

1.2.4 Características de la función

Después de la definición del tipo de dato que devolverá la función con la palabra reservada `RETURNS`, tenemos que indicar las características de la función. Las opciones disponibles son las siguientes:

- `DETERMINISTIC`: Indica que la función siempre devuelve el mismo resultado cuando se utilizan los mismos parámetros de entrada.
- `NOT DETERMINISTIC`: Indica que la función no siempre devuelve el mismo resultado, aunque se utilicen los mismos parámetros de entrada. Esta es la opción que se selecciona por defecto cuando no se indica una característica de forma explícita.
- `CONTAINS SQL`: Indica que la función contiene sentencias SQL, pero no contiene sentencias de manipulación de datos. Algunos ejemplos de sentencias SQL que pueden aparecer en este caso son operaciones con variables (Ej: `SET @x = 1`) o uso de funciones de MySQL (Ej: `SELECT NOW();`) entre otras. Pero en ningún caso aparecerán sentencias de escritura o lectura de datos.
- `NO SQL`: Indica que la función no contiene sentencias SQL.
- `READS SQL DATA`: Indica que la función no modifica los datos de la base de datos y que contiene sentencias de lectura de datos, como la sentencia `SELECT`.
- `MODIFIES SQL DATA`: Indica que la función sí modifica los datos de la base de datos y que contiene sentencias como `INSERT`, `UPDATE` o `DELETE`.

Para poder crear una función en MySQL es necesario indicar al menos una de estas tres características:

- `DETERMINISTIC`
- `NO SQL`
- `READS SQL DATA`

Si no se indica al menos una de estas características obtendremos el siguiente mensaje de error.

```
ERROR 1418 (HY000): This function has none of DETERMINISTIC, NO SQL,
or READS SQL DATA in its declaration and binary logging is enabled
(you *might* want to use the less safe log_bin_trust_function_creators
variable)
```

Es posible configurar el valor de la varible global `log_bin_trust_function_creators` a `1`, para indicar a MySQL que queremos eliminar la restricción de indicar alguna de las características anteriores cuando definimos una función almacenada. Esta variable está configurada con el valor `0` por defecto y para poder modificarla es necesario contar con el privilegio `SUPER`.

```
SET GLOBAL log_bin_trust_function_creators = 1;
```

En lugar de configurar la variable global en tiempo de ejecución, es posible modificarla en el archivo de configuración de MySQL. (Para Linux */etc/mysql/my.cnf. Para Windows %PROGRAMDATA%\MySQL\MySQL Server X.X\my.ini o my.cnf*)

1.2(1) Show, Alter y Drop Procedure/Function

1.2.1. SHOW CREATE PROCEDURE y SHOW CREATE FUNCTION

```
SHOW CREATE {PROCEDURE|FUNTION} proc_name
```

Este comando es una extensión de MySQL . Similar a SHOW CREATE TABLE, retorna la cadena exacta que puede usarse para recrear la rutina nombrada.

mysql> SHOW CREATE FUNCTION test.hello\G
*************************** 1. row ***************************
Function: hello
sql_mode:
Create Function: CREATE FUNCTION `test`.`hello`(s CHAR(20)) RETURNS CHAR(50)
RETURN CONCAT('Hello, ',s,'!')

1.2.2. ALTER PROCEDURE y ALTER FUNCTION

```
ALTER PROCEDURE proc_name [characteristic ...]

characteristic: {
    COMMENT 'string'
  | LANGUAGE SQL
  | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
  | SQL SECURITY { DEFINER | INVOKER }
```

```
}
```

Este comando puede usarse para cambiar las características de un procedimiento o función almacenada. Debe tener el permiso ALTER ROUTINE para la rutina desde MySQL 5.0.3. El permiso se otorga automáticamente al creador de la rutina. Si está activado el logueo binario, necesitará el permiso SUPER
Pueden especificarse varios cambios con ALTER PROCEDURE o ALTER FUNCTION.

1.2.3. DROP PROCEDURE y DROP FUNCTION

```
DROP {PROCEDURE | FUNCTION} [IF EXISTS] sp_name
```

Este comando se usa para borrar un procedimiento o función almacenado. Esto es, la rutina especificada se borra del servidor. Debe tener el permiso ALTER ROUTINE para las rutinas desde MySQL 5.0.3. Este permiso se otorga automáticamente al creador de la rutina.
La cláusula IF EXISTS es una extensión de MySQL . Evita que ocurra un error si la función o procedimiento no existe. Se genera una advertencia que puede verse con SHOW WARNINGS.

1.2(2) Sentencia compuesta BEGIN ... END

```
[begin_label:] BEGIN
    [statement_list]
END [end_label]
```

La sintaxis BEGIN ... END se utiliza para escribir sentencias compuestas que pueden aparecer en el interior de procedimientos almacenados y triggers. Una sentencia compuesta puede contener múltiples sentencias, encerradas por las palabras BEGIN y END.

`statement_list` es una lista de una o más sentencias. Cada sentencia dentro de lista_sentencias debe terminar con un punto y coma (;) delimitador de sentencias. `statement_list` es opcional, lo que significa que la sentencia compuesta vacía (BEGIN END es correcta.

El uso de múltiples sentencias requiere que el cliente pueda enviar cadenas de sentencias que contengan el delimitador ;. Esto se gestiona en el cliente de línea de comandos mysql con el comando delimiter. Cambiar el delimitador de fin de sentencia ; (por ejemplo con //) permite utilizar ; en el cuerpo de una rutina.

Un comando compuesto puede etiquetarse. No se puede poner end_label a no ser que también esté presente begin_label , y si ambos están, deben ser iguales.

1.2.1 Variables en procedimientos almacenados o funciones

Tanto en los procedimientos como en las funciones es posible declarar variables locales con la palabra reservada `DECLARE`.

1.2.1.1. Declarar variables locales con DECLARE

```
DECLARE var_name [, var_name] ... type [DEFAULT value]
```

Este comando se usa para declarar variables locales. Para proporcionar un valor por defecto para la variable, incluya una cláusula DEFAULT . El valor puede especificarse como expresión, no necesita ser una constante. Si la cláusula DEFAULT no está presente, el valor inicial es NULL.

El ámbito de una variable local será el bloque `BEGIN` y `END` del procedimiento o la función donde ha sido declarada.

Una restricción que hay que tener en cuenta a la hora de trabajar con variables locales, es que se deben declarar antes de los cursores y los *handlers*.

Ejemplo:

En este ejemplo estamos declarando una variable local con el nombre `total` que es de tipo `INT UNSIGNED`.

```
DECLARE total INT UNSIGNED;
```

1.2.1.2. Sentencia SET para variables

```
SET variable = expr [, variable = expr] ...
```

El comando SET en procedimientos almacenados es una versión extendida del comando general SET. Las variables referenciadas pueden ser las declaradas dentro de una rutina, o variables de servidor globales.

El comando SET en procedimientos almacenados se implementa como parte de la sintaxis SET pre-existente. Esto permite una sintaxis extendida de SET a=x, b=y, ... donde distintos tipos de variables (variables declaradas local y globalmente y variables de sesión del servidor) pueden mezclarse. Esto permite combinaciones de variables locales y algunas opciones que tienen sentido sólo para variables de sistema; en tal caso, las opciones se reconocen pero se ignoran.

Examples:

SET @name = 43;

SET @total_tax = (SELECT SUM(tax) FROM taxable_transactions);

1.2.1.3. La sentencia SELECT ... INTO

SELECT col_name[,...] INTO var_name[,...] table_expr

Esta sintaxis SELECT almacena columnas seleccionadas directamente en variables. Por lo tanto, sólo un registro puede retornarse.

SELECT id,data INTO x,y FROM test.t1 LIMIT 1;

Puede encontrar más información en la documentación oficial de MySQL.

1.2.1.7. Ejemplos

Escriba una función llamada `contar_productos` que reciba como entrada el nombre de la gama y devuelva el número de productos que existen dentro de esa gama.

```
DELIMITER //
DROP FUNCTION IF EXISTS contar_productos//
CREATE FUNCTION contar_productos(gama VARCHAR(50))
  RETURNS INT UNSIGNED
BEGIN
  -- Paso 1. Declaramos una variable local
  DECLARE total INT UNSIGNED;

  -- Paso 2. Contamos los productos
  SET total = (
    SELECT COUNT(*)
    FROM producto
    WHERE producto.gama = gama);

  -- Paso 3. Devolvemos el resultado
  RETURN total;
END
//

DELIMITER ;

SELECT contar_productos('Herramientas');
```

1.3 Estructuras de control

1.3.1 Instrucciones condicionales

1.3.1.1 `IF-THEN-ELSE`

```
IF search_condition THEN statement_list
    [ELSEIF search_condition THEN statement_list] ...
    [ELSE statement_list]

END IF
```

Puede encontrar más información en la documentación oficial de MySQL.

1.3.1.2 `CASE`

Existen dos formas de utilzar `CASE`:

```
CASE case_value
    WHEN when_value THEN statement_list
    [WHEN when_value THEN statement_list] ...
    [ELSE statement_list]

END CASE
```

o

```
CASE
```

```
    WHEN search_condition THEN statement_list
    [WHEN search_condition THEN statement_list] ...
    [ELSE statement_list]

END CASE
```

Puede encontrar más información en la documentación oficial de MySQL.

1.3.2 Instrucciones repetitivas o bucles

1.3.2.1 LOOP

```
[begin_label:] LOOP
    statement_list

END LOOP [end_label]
```

Ejemplo:

```
CREATE PROCEDURE doiterate(p1 INT)
BEGIN
  label1: LOOP
    SET p1 = p1 + 1;
    IF p1 < 10 THEN
      ITERATE label1;
    END IF;
    LEAVE label1;
  END LOOP label1;
  SET @x = p1;

END;
```

Ejemplo:

```
DELIMITER //
DROP PROCEDURE IF EXISTS ejemplo_bucle_loop//
CREATE PROCEDURE ejemplo_bucle_loop(IN tope INT, OUT suma INT)
BEGIN
  DECLARE contador INT;

  SET contador = 1;
  SET suma = 0;

  bucle: LOOP
    IF contador > tope THEN
      LEAVE bucle;
    END IF;

    SET suma = suma + contador;
    SET contador = contador + 1;
  END LOOP;
END
//

DELIMITER ;
CALL ejemplo_bucle_loop(10, @resultado);

SELECT @resultado;
```

Puede encontrar más información en la documentación oficial de MySQL.

1.3.2.2 `REPEAT`

```
[begin_label:] REPEAT
    statement_list
UNTIL search_condition

END REPEAT [end_label]
```

Ejemplo:

```
DELIMITER //
DROP PROCEDURE IF EXISTS ejemplo_bucle_repeat//
CREATE PROCEDURE ejemplo_bucle_repeat(IN tope INT, OUT suma INT)
BEGIN
  DECLARE contador INT;

    SET contador = 1;
    SET suma = 0;

    REPEAT
    SET suma = suma + contador;
    SET contador = contador + 1;
  UNTIL contador > tope
  END REPEAT;
END
//

DELIMITER ;
CALL ejemplo_bucle_repeat(10, @resultado);

SELECT @resultado;
```

Puede encontrar más información en la documentación oficial de MySQL.

1.3.2.3 `WHILE`

```
[begin_label:] WHILE search_condition DO
    statement_list

END WHILE [end_label]
```

Ejemplo:

```
DELIMITER //
DROP PROCEDURE IF EXISTS ejemplo_bucle_while//
CREATE PROCEDURE ejemplo_bucle_while(IN tope INT, OUT suma INT)
BEGIN
  DECLARE contador INT;

  SET contador = 1;
  SET suma = 0;

  WHILE contador <= tope DO
    SET suma = suma + contador;
    SET contador = contador + 1;
  END WHILE;
END
//

DELIMITER ;
CALL ejemplo_bucle_while(10, @resultado);
```

```
SELECT @resultado;
```

Puede encontrar más información en la documentación oficial de MySQL.

1.4 Manejo de errores

1.4.1 `DECLARE ... HANDLER`

```
DECLARE handler_action HANDLER
    FOR condition_value [, condition_value] ...
    statement

handler_action:
    CONTINUE
  | EXIT
  | UNDO

condition_value:
    mysql_error_code
  | SQLSTATE [VALUE] sqlstate_value
  | condition_name
  | SQLWARNING
  | NOT FOUND

  | SQLEXCEPTION
```

Las acciones posibles que podemos seleccionar como *handler_action* son:

- `CONTINUE`: La ejecución del programa continúa.
- `EXIT`: Termina la ejecución del programa.
- `UNDO`: No está soportado en MySQL.

Puede encontrar más información en la documentación oficial de MySQL.

Ejemplo indicando el número de error de MySQL:

En este ejemplo estamos declarando un *handler* que se ejecutará cuando se produzca el error `1051` de MySQL, que ocurre cuando se intenta acceder a una tabla que no existe en la base de datos. En este caso la acción del *handler* es `CONTINUE` lo que quiere decir que después de ejecutar las instrucciones especificadas en el cuerpo del *handler* el procedimiento almacenado continuará su ejecución.

```
DECLARE CONTINUE HANDLER FOR 1051
BEGIN
  -- body of handler

END;
```

Ejemplo para `SQLSTATE`:

También podemos indicar el valor de la variable `SQLSTATE`. Por ejemplo, cuando se intenta acceder a una tabla que no existe en la base de datos, el valor de la variable `SQLSTATE` es `42S02`.

```
DECLARE CONTINUE HANDLER FOR SQLSTATE '42S02'
BEGIN
```

```
  -- body of handler

END;
```

Ejemplo para `SQLWARNING`:

Es equivalente a indicar todos los valores de `SQLSTATE` que empiezan con `01`.

```
DECLARE CONTINUE HANDLER FOR SQLWARNING
BEGIN
  -- body of handler

END;
```

Ejemplo para `NOT FOUND`:

Es equivalente a indicar todos los valores de `SQLSTATE` que empiezan con `02`. Lo usaremos cuando estemos trabajando con cursores para controlar qué ocurre cuando un cursor alcanza el final del *data set*. Si no hay más filas disponibles en el cursor, entonces ocurre una condición de `NO DATA` con un valor de `SQLSTATE` igual a `02000`. Para detectar esta condición podemos usar un *handler* para controlarlo.

```
DECLARE CONTINUE HANDLER FOR NOT FOUND
BEGIN
  -- body of handler

END;
```

Ejemplo para `SQLEXCEPTION:`:

Es equivalente a indicar todos los valores de `SQLSTATE` que empiezan por `00`, `01` y `02`.

```
DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
BEGIN
  -- body of handler

END;
```

1.4.2 Ejemplo 1 - `DECLARE CONTINUE HANDLER`

```
-- Paso 1
DROP DATABASE IF EXISTS test;
CREATE DATABASE test;
USE test;

-- Paso 2
CREATE TABLE test.t (s1 INT, PRIMARY KEY (s1));

-- Paso 3
DELIMITER //
CREATE PROCEDURE handlerdemo ()
BEGIN
  DECLARE CONTINUE HANDLER FOR SQLSTATE '23000' SET @x = 1;
  SET @x = 1;
  INSERT INTO test.t VALUES (1);
  SET @x = 2;
  INSERT INTO test.t VALUES (1);
  SET @x = 3;
END
```

```
//

DELIMITER ;
CALL handlerdemo();

SELECT @x;
```

¿Qué valor devolvería la sentencia `SELECT @x`?

1.4.3 Ejemplo 2 - `DECLARE EXIT HANDLER`

```
-- Paso 1
DROP DATABASE IF EXISTS test;
CREATE DATABASE test;
USE test;

-- Paso 2
CREATE TABLE test.t (s1 INT, PRIMARY KEY (s1));

-- Paso 3
DELIMITER //
CREATE PROCEDURE handlerdemo ()
BEGIN
  DECLARE EXIT HANDLER FOR SQLSTATE '23000' SET @x = 1;
  SET @x = 1;
  INSERT INTO test.t VALUES (1);
  SET @x = 2;
  INSERT INTO test.t VALUES (1);
  SET @x = 3;
END
//

DELIMITER ;
CALL handlerdemo();

SELECT @x;
```

¿Qué valor devolvería la sentencia `SELECT @x`?

1.5 Cómo realizar transacciones con procedimientos almacenados

Podemos utilizar el manejo de errores para decidir si hacemos `ROLLBACK` de una transacción. En el siguiente ejemplo vamos a capturar los errores que se produzcan de tipo `SQLEXCEPTION` y `SQLWARNING`.

Ejemplo:

```
DELIMITER //
CREATE PROCEDURE transaccion_en_mysql()
BEGIN
  DECLARE EXIT HANDLER FOR SQLEXCEPTION
    BEGIN
      -- ERROR
      ROLLBACK;
  END;

  DECLARE EXIT HANDLER FOR SQLWARNING
  BEGIN
      -- WARNING
```

```
  ROLLBACK;
  END;

  START TRANSACTION;
    -- Sentencias SQL
  COMMIT;
END

//
```

En lugar de tener un manejador para cada tipo de error, podemos tener uno común para todos los casos.

```
DELIMITER //
CREATE PROCEDURE transaccion_en_mysql()
BEGIN
  DECLARE EXIT HANDLER FOR SQLEXCEPTION, SQLWARNING
    BEGIN
      -- ERROR, WARNING
      ROLLBACK;
  END;

  START TRANSACTION;
    -- Sentencias SQL
  COMMIT;
END

//
```

1.6 Cursores

Los cursores nos permiten almacenar una conjunto de filas de una tabla en una estructura de datos que podemos ir recorriendo de forma secuencial.

Los cursores tienen las siguientes propiedades:

- *Asensitive*: The server may or may not make a copy of its result table.
- *Read only*: son de sólo lectura. No permiten actualizar los datos.
- *Nonscrollable*: sólo pueden ser recorridos en una dirección y no podemos saltarnos filas.

Cuando declaramos un cursor dentro de un procedimiento almacenado debe aparecer antes de las declaraciones de los manejadores de errores (`HANDLER`) y después de la declaración de variables locales.

1.6.1 Operaciones con cursores

Las operaciones que podemos hacer con los cursores son las siguientes:

1.6.1.1 `DECLARE`

El primer paso que tenemos que hacer para trabajar con cursores es declararlo. La sintaxis para declarar un cursor es:

```
DECLARE cursor_name CURSOR FOR select_statement
```

1.6.1.2 OPEN

Una vez que hemos declarado un cursor tenemos que abrirlo con `OPEN`.

```
OPEN cursor_name
```

1.6.1.3 FETCH

Una vez que el cursor está abierto podemos ir obteniendo cada una de las filas con `FETCH`. La sintaxis es la siguiente:

```
FETCH [[NEXT] FROM] cursor_name INTO var_name [, var_name] ...
```

Cuando se está recorriendo un cursor y no quedan filas por recorrer se lanza el error `NOT FOUND`, que se corresponde con el valor `SQLSTATE` '02000'. Por eso cuando estemos trabajando con cursores será necesario declarar un *handler* para manejar este error.

```
DECLARE CONTINUE HANDLER FOR NOT FOUND ...
```

1.6.1.4 CLOSE

Cuando hemos terminado de trabajar con un cursor tenemos que cerrarlo.

```
CLOSE cursor_name
```

Ejemplo:

```
-- Paso 1
DROP DATABASE IF EXISTS test;
CREATE DATABASE test;
USE test;

-- Paso 2
CREATE TABLE t1 (
  id INT UNSIGNED PRIMARY KEY,
    data VARCHAR(16)
);

CREATE TABLE t2 (
  i INT UNSIGNED
);

CREATE TABLE t3 (
  data VARCHAR(16),
    i INT UNSIGNED
);

INSERT INTO t1 VALUES (1, 'A');
INSERT INTO t1 VALUES (2, 'B');

INSERT INTO t2 VALUES (10);
INSERT INTO t2 VALUES (20);

-- Paso 3
DELIMITER //
DROP PROCEDURE IF EXISTS curdemo//
CREATE PROCEDURE curdemo()
BEGIN
  DECLARE done INT DEFAULT FALSE;
```

```
  DECLARE a CHAR(16);
  DECLARE b, c INT;
  DECLARE cur1 CURSOR FOR SELECT id,data FROM test.t1;
  DECLARE cur2 CURSOR FOR SELECT i FROM test.t2;
  DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

  OPEN cur1;
  OPEN cur2;

  read_loop: LOOP
    FETCH cur1 INTO b, a;
    FETCH cur2 INTO c;
    IF done THEN
      LEAVE read_loop;
    END IF;
    IF b < c THEN
      INSERT INTO test.t3 VALUES (a,b);
    ELSE
      INSERT INTO test.t3 VALUES (a,c);
    END IF;
  END LOOP;

  CLOSE cur1;
  CLOSE cur2;
END

-- Paso 4
DELIMITER ;
CALL curdemo();


SELECT * FROM t3;
```

Solución utilizando un bucle `WHILE`:

```
DELIMITER //
DROP PROCEDURE IF EXISTS curdemo//
CREATE PROCEDURE curdemo()
BEGIN
  DECLARE done INT DEFAULT FALSE;
  DECLARE a CHAR(16);
  DECLARE b, c INT;
  DECLARE cur1 CURSOR FOR SELECT id,data FROM test.t1;
  DECLARE cur2 CURSOR FOR SELECT i FROM test.t2;
  DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

  OPEN cur1;
  OPEN cur2;

  WHILE done = FALSE DO
    FETCH cur1 INTO b, a;
    FETCH cur2 INTO c;

    IF done = FALSE THEN
    IF b < c THEN
      INSERT INTO test.t3 VALUES (a,b);
    ELSE
      INSERT INTO test.t3 VALUES (a,c);
    END IF;
  END IF;
  END WHILE;

  CLOSE cur1;
  CLOSE cur2;

END;
```

1.7 *Triggers*

```
CREATE
    [DEFINER = { user | CURRENT_USER }]
    TRIGGER trigger_name
    trigger_time trigger_event
    ON tbl_name FOR EACH ROW
    [trigger_order]
    trigger_body

trigger_time: { BEFORE | AFTER }

trigger_event: { INSERT | UPDATE | DELETE }


trigger_order: { FOLLOWS | PRECEDES } other_trigger_name
```

Un *trigger* es un objeto de la base de datos que está asociado con una tabla y que se activa cuando ocurre un evento sobre la tabla.

Los eventos que pueden ocurrir sobre la tabla son:

- `INSERT`: El *trigger* se activa cuando se inserta una nueva fila sobre la tabla asociada.
- `UPDATE`: El *trigger* se activa cuando se actualiza una fila sobre la tabla asociada.
- `DELETE`: El *trigger* se activa cuando se elimina una fila sobre la tabla asociada.

Ejemplo:

Crea una base de datos llamada `test` que contenga una tabla llamada `alumnos` con las siguientes columnas.

Tabla `alumnos`:

- `id` (entero sin signo)
- `nombre` (cadena de caracteres)
- `apellido1` (cadena de caracteres)
- `apellido2` (cadena de caracteres)
- `nota` (número real)

Una vez creada la tabla escriba dos triggers con las siguientes características:

- Trigger 1: `trigger_check_nota_before_insert`
    - Se ejecuta sobre la tabla `alumnos`.
    - Se ejecuta *antes* de una operación de *inserción*.
    - Si el nuevo valor de la nota que se quiere insertar es negativo, se guarda como 0.
    - Si el nuevo valor de la nota que se quiere insertar es mayor que 10, se guarda como 10.
- Trigger2 : `trigger_check_nota_before_update`
    - Se ejecuta sobre la tabla `alumnos`.
    - Se ejecuta *antes* de una operación de *actualización*.
    - Si el nuevo valor de la nota que se quiere actualizar es negativo, se guarda como 0.

- Si el nuevo valor de la nota que se quiere actualizar es mayor que 10, se guarda como 10.

Una vez creados los triggers escriba varias sentencias de inserción y actualización sobre la tabla `alumnos` y verifica que los *triggers* se están ejecutando correctamente.

```
-- Paso 1
DROP DATABASE IF EXISTS test;
CREATE DATABASE test;
USE test;

-- Paso 2
CREATE TABLE alumnos (
    id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    nombre VARCHAR(50) NOT NULL,
    apellido1 VARCHAR(50) NOT NULL,
    apellido2 VARCHAR(50),
    nota FLOAT
);

-- Paso 3
DELIMITER //
DROP TRIGGER IF EXISTS trigger_check_nota_before_insert//
CREATE TRIGGER trigger_check_nota_before_insert
BEFORE INSERT
ON alumnos FOR EACH ROW
BEGIN
  IF NEW.nota < 0 THEN
    set NEW.nota = 0;
  ELSEIF NEW.nota > 10 THEN
    set NEW.nota = 10;
  END IF;
END

DELIMITER //
DROP TRIGGER IF EXISTS trigger_check_nota_before_update//
CREATE TRIGGER trigger_check_nota_before_update
BEFORE UPDATE
ON alumnos FOR EACH ROW
BEGIN
  IF NEW.nota < 0 THEN
    set NEW.nota = 0;
  ELSEIF NEW.nota > 10 THEN
    set NEW.nota = 10;
  END IF;
END

-- Paso 4
DELIMITER ;
INSERT INTO alumnos VALUES (1, 'Pepe', 'López', 'López', -1);
INSERT INTO alumnos VALUES (2, 'María', 'Sánchez', 'Sánchez', 11);
INSERT INTO alumnos VALUES (3, 'Juan', 'Pérez', 'Pérez', 8.5);

-- Paso 5
SELECT * FROM alumnos;

-- Paso 6
UPDATE alumnos SET nota = -4 WHERE id = 3;
UPDATE alumnos SET nota = 14 WHERE id = 3;
UPDATE alumnos SET nota = 9.5 WHERE id = 3;

-- Paso 7

SELECT * FROM alumnos;
```

1.8 Ejercicios

1.8.1 Procedimientos sin sentencias SQL

1. Escribe un procedimiento que no tenga ningún parámetro de entrada ni de salida y que muestre el texto `¡Hola mundo!`.
2. Escribe un procedimiento que reciba un número real de entrada y muestre un mensaje indicando si el número es positivo, negativo o cero.
3. Modifique el procedimiento diseñado en el ejercicio anterior para que tenga un parámetro de entrada, con el valor un número real, y un parámetro de salida, con una cadena de caracteres indicando si el número es positivo, negativo o cero.
4. Escribe un procedimiento que reciba un número real de entrada, que representa el valor de la nota de un alumno, y muestre un mensaje indicando qué nota ha obtenido teniendo en cuenta las siguientes condiciones:

- [0,5) = Insuficiente
- [5,6) = Aprobado
- [6, 7) = Bien
- [7, 9) = Notable
- [9, 10] = Sobresaliente
- En cualquier otro caso la nota no será válida.

5. Modifique el procedimiento diseñado en el ejercicio anterior para que tenga un parámetro de entrada, con el valor de la nota en formato numérico y un parámetro de salida, con una cadena de texto indicando la nota correspondiente.
6. Resuelva el procedimiento diseñado en el ejercicio anterior haciendo uso de la estructura de control `CASE`.
7. Escribe un procedimiento que reciba como parámetro de entrada un valor numérico que represente un día de la semana y que devuelva una cadena de caracteres con el nombre del día de la semana correspondiente. Por ejemplo, para el valor de entrada `1` debería devolver la cadena `lunes`.

1.8.2 Procedimientos con sentencias SQL

1. Escribe un procedimiento que reciba el nombre de un país como parámetro de entrada y realice una consulta sobre la tabla `cliente` para obtener todos los clientes que existen en la tabla de ese país.
2. Escribe un procedimiento que reciba como parámetro de entrada una forma de pago, que será una cadena de caracteres (Ejemplo: `PayPal`, `Transferencia`, etc). Y devuelva como salida el pago de máximo valor realizado para esa forma de pago. Deberá hacer uso de la tabla `pago` de la base de datos `jardineria`.
3. Escribe un procedimiento que reciba como parámetro de entrada una forma de pago, que será una cadena de caracteres (Ejemplo: `PayPal`, `Transferencia`, etc). Y devuelva como salida los siguientes valores teniendo en cuenta la forma de pago seleccionada como parámetro de entrada:

- el pago de máximo valor,
- el pago de mínimo valor,
- el valor medio de los pagos realizados,
- la suma de todos los pagos,

- el número de pagos realizados para esa forma de pago.

Deberá hacer uso de la tabla `pago` de la base de datos `jardineria`.

4. Crea una base de datos llamada `procedimientos` que contenga una tabla llamada `cuadrados`. La tabla `cuadrados` debe tener dos columnas de tipo `INT UNSIGNED`, una columna llamada `número` y otra columna llamada `cuadrado`.

Una vez creada la base de datos y la tabla deberá crear un procedimiento llamado `calcular_cuadrados` con las siguientes características. El procedimiento recibe un parámetro de entrada llamado `tope` de tipo `INT UNSIGNED` y calculará el valor de los cuadrados de los primeros números naturales hasta el valor introducido como parámetro. El valor del números y de sus cuadrados deberán ser almacenados en la tabla `cuadrados` que hemos creado previamente.

Tenga en cuenta que el procedimiento deberá eliminar el contenido actual de la tabla antes de insertar los nuevos valores de los cuadrados que va a calcular.

Utilice un bucle `WHILE` para resolver el procedimiento.

5. Utilice un bucle `REPEAT` para resolver el procedimiento del ejercicio anterior.
6. Utilice un bucle `LOOP` para resolver el procedimiento del ejercicio anterior.
7. Crea una base de datos llamada `procedimientos` que contenga una tabla llamada `ejercicio`. La tabla debe tener una única columna llamada `número` y el tipo de dato de esta columna debe ser `INT UNSIGNED`.

Una vez creada la base de datos y la tabla deberá crear un procedimiento llamado `calcular_números` con las siguientes características. El procedimiento recibe un parámetro de entrada llamado `valor_inicial` de tipo `INT UNSIGNED` y deberá almacenar en la tabla `ejercicio` toda la secuencia de números desde el valor inicial pasado como entrada hasta el 1.

Tenga en cuenta que el procedimiento deberá eliminar el contenido actual de las tablas antes de insertar los nuevos valores.

Utilice un bucle `WHILE` para resolver el procedimiento.

8. Utilice un bucle `REPEAT` para resolver el procedimiento del ejercicio anterior.
9. Utilice un bucle `LOOP` para resolver el procedimiento del ejercicio anterior.
10. Crea una base de datos llamada `procedimientos` que contenga una tabla llamada `pares` y otra tabla llamada `impares`. Las dos tablas deben tener única columna llamada `número` y el tipo de dato de esta columna debe ser `INT UNSIGNED`.

Una vez creada la base de datos y las tablas deberá crear un procedimiento llamado `calcular_pares_impares` con las siguientes características. El procedimiento recibe un parámetro de entrada llamado `tope` de tipo `INT UNSIGNED` y deberá almacenar en la tabla `pares` aquellos números pares que existan entre el número 1 el valor introducido como parámetro. Habrá que realizar la misma operación para almacenar los números impares en la tabla `impares`.

Tenga en cuenta que el procedimiento deberá eliminar el contenido actual de las tablas antes de insertar los nuevos valores.

Utilice un bucle `WHILE` para resolver el procedimiento.

11. Utilice un bucle `REPEAT` para resolver el procedimiento del ejercicio anterior.
12. Utilice un bucle `LOOP` para resolver el procedimiento del ejercicio anterior.

1.8.3 Funciones sin sentencias SQL

1. Escribe una función que reciba un número entero de entrada y devuelva `TRUE` si el número es par o `FALSE` en caso contrario.
2. Escribe una función que devuelva el valor de la hipotenusa de un triángulo a partir de los valores de sus lados.
3. Escribe una función que reciba como parámetro de entrada un valor numérico que represente un día de la semana y que devuelva una cadena de caracteres con el nombre del día de la semana correspondiente. Por ejemplo, para el valor de entrada `1` debería devolver la cadena `lunes`.
4. Escribe una función que reciba tres números reales como parámetros de entrada y devuelva el mayor de los tres.
5. Escribe una función que devuelva el valor del área de un círculo a partir del valor del radio que se recibirá como parámetro de entrada.
6. Escribe una función que devuelva como salida el número de años que han transcurrido entre dos fechas que se reciben como parámetros de entrada. Por ejemplo, si pasamos como parámetros de entrada las fechas `2018-01-01` y `2008-01-01` la función tiene que devolver que han pasado 10 años.

Para realizar esta función puede hacer uso de las siguientes funciones que nos proporciona MySQL:

- `DATEDIFF`
- `TRUNCATE`

7. Escribe una función que reciba una cadena de entrada y devuelva la misma cadena pero sin acentos. La función tendrá que reemplazar todas las vocales que tengan acento por la misma vocal pero sin acento. Por ejemplo, si la función recibe como parámetro de entrada la cadena `María` la función debe devolver la cadena `Maria`.

1.8.4 Funciones con sentencias SQL

1. Escribe una función para la base de datos `tienda` que devuelva el número total de productos que hay en la tabla `productos`.
2. Escribe una función para la base de datos `tienda` que devuelva el valor medio del precio de los productos de un determinado fabricante que se recibirá como parámetro de entrada. El parámetro de entrada será el nombre del fabricante.
3. Escribe una función para la base de datos `tienda` que devuelva el valor máximo del precio de los productos de un determinado fabricante que se recibirá como parámetro de entrada. El parámetro de entrada será el nombre del fabricante.
4. Escribe una función para la base de datos `tienda` que devuelva el valor mínimo del precio de los productos de un determinado fabricante que se recibirá como parámetro de entrada. El parámetro de entrada será el nombre del fabricante.

1.8.5 Manejo de errores en MySQL

1. Crea una base de datos llamada `test` que contenga una tabla llamada `alumno`. La tabla debe tener cuatro columnas:

- `id`: entero sin signo (clave primaria).
- `nombre`: cadena de 50 caracteres.
- `apellido1`: cadena de 50 caracteres.
- `apellido2`: cadena de 50 caracteres.

Una vez creada la base de datos y la tabla deberá crear un procedimiento llamado `insertar_alumno` con las siguientes características. El procedimiento recibe cuatro parámetros de entrada (`id`, `nombre`, `apellido1`, `apellido2`) y los insertará en la tabla `alumno`. El procedimiento devolverá como salida un parámetro llamado `error` que tendrá un valor igual a 0 si la operación se ha podido realizar con éxito y un valor igual a 1 en caso contrario.

Deberá manejar los errores que puedan ocurrir cuando se intenta insertar una fila que contiene una clave primaria repetida.

1.8.6 Transacciones con procedimientos almacenados

1. Crea una base de datos llamada `cine` que contenga dos tablas con las siguientes columnas.

Tabla `cuentas`:

- `id_cuenta`: entero sin signo (clave primaria).
- `saldo`: real sin signo.

Tabla `entradas`:

- `id_butaca`: entero sin signo (clave primaria).
- `nif`: cadena de 9 caracteres.

Una vez creada la base de datos y las tablas deberá crear un procedimiento llamado `comprar_entrada` con las siguientes características. El procedimiento recibe 3 parámetros de entrada (`nif`, `id_cuenta`, `id_butaca`) y devolverá como salida un parámetro llamado `error` que tendrá un valor igual a 0 si la compra de la entrada se ha podido realizar con éxito y un valor igual a 1 en caso contrario.

El procedimiento de compra realiza los siguientes pasos:

- Inicia una transacción.
- Actualiza la columna `saldo` de la tabla `cuentas` cobrando 5 euros a la cuenta con el `id_cuenta` adecuado.
- Inserta una una fila en la tabla `entradas` indicando la butaca (`id_butaca`) que acaba de comprar el usuario (`nif`).
- Comprueba si ha ocurrido algún error en las operaciones anteriores. Si no ocurre ningún error entonces aplica un `COMMIT` a la transacción y si ha ocurrido algún error aplica un `ROLLBACK`.

Deberá manejar los siguientes errores que puedan ocurrir durante el proceso.

- ERROR 1264 (Out of range value)
- ERROR 1062 (Duplicate entry for PRIMARY KEY)

2. ¿Qué ocurre cuando intentamos comprar una entrada y le pasamos como parámetro un número de cuenta que no existe en la tabla `cuentas`? ¿Ocurre algún error o podemos comprar la entrada?

En caso de que exista algún error, ¿cómo podríamos resolverlo?.

1.8.7 Cursores

1. Escribe las sentencias SQL necesarias para crear una base de datos llamada `test`, una tabla llamada `alumnos` y 4 sentencias de inserción para inicializar la tabla. La tabla alumnos está formada por las siguientes columnas:

- `id` (entero sin signo y clave primaria)
- `nombre` (cadena de caracteres)
- `apellido1` (cadena de caracteres)
- `apellido2` (cadena de caracteres
- `fecha_nacimiento` (fecha)

Una vez creada la tabla se decide añadir una nueva columna a la tabla llamada `edad` que será un valor calculado a partir de la columna `fecha_nacimiento`. Escriba la sentencia SQL necesaria para modificar la tabla y añadir la nueva columna.

Escriba una función llamada `calcular_edad` que reciba una fecha y devuelva el número de años que han pasado desde la fecha actual hasta la fecha pasada como parámetro:

- Función: `calcular_edad`
- Entrada: Fecha
- Salida: Número de años (entero)

Ahora escriba un procedimiento que permita calcular la edad de todos los alumnmos que ya existen en la tabla. Para esto será necesario crear un procedimiento llamado `actualizar_columna_edad` que calcule la edad de cada alumno y actualice la tabla. Este procedimiento hará uso de la función `calcular_edad` que hemos creado en el paso anterior.

2. Modifica la tabla `alumnos` del ejercicio anterior para añadir una nueva columna `email`. Una vez que hemos modificado la tabla necesitamos asignarle una dirección de correo electrónico de forma automática.

Escriba un procedimiento llamado crear_email que dados los parámetros de entrada: `nombre`, `apellido1`, `apellido2` y `dominio`, cree una dirección de email y la devuelva como salida.

- Procedimiento: `crear_email`
- Entrada:
    - `nombre` (cadena de caracteres)
    - `apellido1` (cadena de caracteres)
    - `apellido2` (cadena de caracteres)
    - `dominio` (cadena de caracteres)
- Salida:
    - `email` (cadena de caracteres)

devuelva una dirección de correo electrónico con el siguiente formato:

- El primer carácter del parámetro `nombre`.
- Los tres primeros caracteres del parámetro `apellido1`.
- Los tres primeros caracteres del parámetro `apellido2`.
- El carácter `@`.
- El dominio pasado como parámetro.

Ahora escriba un procedimiento que permita crear un email para todos los alumnmos que ya existen en la tabla. Para esto será necesario crear un procedimiento llamado `actualizar_columna_email` que actualice la columna `email` de la tabla `alumnos`. Este procedimiento hará uso del procedimiento `crear_email` que hemos creado en el paso anterior.

3. Escribe un procedimiento llamado `crear_lista_emails_alumnos` que devuelva la lista de emails de la tabla `alumnos` separados por un punto y coma. Ejemplo: `juan@iescelia.org;maria@iescelia.org;pepe@iescelia.org;lucia@iescelia.org`.

1.8.8 *Triggers*

1. Crea una base de datos llamada `test` que contenga una tabla llamada `alumnos` con las siguientes columnas.

Tabla `alumnos`:

- `id` (entero sin signo)
- `nombre` (cadena de caracteres)
- `apellido1` (cadena de caracteres)
- `apellido2` (cadena de caracteres)
- `nota` (número real)

Una vez creada la tabla escriba dos triggers con las siguientes características:

- Trigger 1: `trigger_check_nota_before_insert`
  - Se ejecuta sobre la tabla `alumnos`.
  - Se ejecuta *antes* de una operación de *inserción*.
  - Si el nuevo valor de la nota que se quiere insertar es negativo, se guarda como 0.
  - Si el nuevo valor de la nota que se quiere insertar es mayor que 10, se guarda como 10.
- Trigger2 : `trigger_check_nota_before_update`
  - Se ejecuta sobre la tabla `alumnos`.
  - Se ejecuta *antes* de una operación de *actualización*.
  - Si el nuevo valor de la nota que se quiere actualizar es negativo, se guarda como 0.
  - Si el nuevo valor de la nota que se quiere actualizar es mayor que 10, se guarda como 10.

Una vez creados los triggers escriba varias sentencias de inserción y actualización sobre la tabla `alumnos` y verifica que los *triggers* se están ejecutando correctamente.

2. Crea una base de datos llamada `test` que contenga una tabla llamada `alumnos` con las siguientes columnas.

Tabla `alumnos`:

- `id` (entero sin signo)
- `nombre` (cadena de caracteres)
- `apellido1` (cadena de caracteres)
- `apellido2` (cadena de caracteres)
- `email` (cadena de caracteres)

Escriba un procedimiento llamado crear_email que dados los parámetros de entrada: `nombre`, `apellido1`, `apellido2` y `dominio`, cree una dirección de email y la devuelva como salida.

- Procedimiento: `crear_email`
- Entrada:
  - `nombre` (cadena de caracteres)
  - `apellido1` (cadena de caracteres)
  - `apellido2` (cadena de caracteres)
  - `dominio` (cadena de caracteres)
- Salida:
  - `email` (cadena de caracteres)

devuelva una dirección de correo electrónico con el siguiente formato:

- El primer carácter del parámetro `nombre`.
- Los tres primeros caracteres del parámetro `apellido1`.
- Los tres primeros caracteres del parámetro `apellido2`.
- El carácter `@`.
- El dominio pasado como parámetro.

Una vez creada la tabla escriba un trigger con las siguientes características:

- Trigger: `trigger_crear_email_before_insert`
  - Se ejecuta sobre la tabla `alumnos`.
  - Se ejecuta *antes* de una operación de *inserción*.
  - Si el nuevo valor del email que se quiere insertar es `NULL`, entonces se le creará automáticamente una dirección de email y se insertará en la tabla.
  - Si el nuevo valor del email no es `NULL` se guardará en la tabla el valor del email.

Nota: Para crear la nueva dirección de email se deberá hacer uso del procedimiento `crear_email`.

3. Modifica el ejercicio anterior y añade un nuevo *trigger* que las siguientes características:

Trigger: `trigger_guardar_email_after_update`:

- Se ejecuta sobre la tabla `alumnos`.
- Se ejecuta *después* de una operación de *actualización*.

- Cada vez que un alumno modifique su dirección de email se deberá insertar un nuevo registro en una tabla llamada `log_cambios_email`.

La tabla `log_cambios_email` contiene los siguientes campos:

- `id`: clave primaria (entero autonumérico)
- `id_alumno`: id del alumno (entero)
- `fecha_hora`: marca de tiempo con el instante del cambio (fecha y hora)
- `old_email`: valor anterior del email (cadena de caracteres)
- `new_email`: nuevo valor con el que se ha actualizado

4. Modifica el ejercicio anterior y añade un nuevo *trigger* que tenga las siguientes características:

Trigger: `trigger_guardar_alumnos_eliminados`:

- Se ejecuta sobre la tabla `alumnos`.
- Se ejecuta *después* de una operación de *borrado*.
- Cada vez que se elimine un alumno de la tabla `alumnos` se deberá insertar un nuevo registro en una tabla llamada `log_alumnos_eliminados`.

La tabla `log_alumnos_eliminados` contiene los siguientes campos:

- `id`: clave primaria (entero autonumérico)
- `id_alumno`: id del alumno (entero)
- `fecha_hora`: marca de tiempo con el instante del cambio (fecha y hora)
- `nombre`: nombre del alumno eliminado (cadena de caracteres)
- `apellido1`: primer apellido del alumno eliminado (cadena de caracteres)
- `apellido2`: segundo apellido del alumno eliminado (cadena de caracteres)
- `email`: email del alumno eliminado (cadena de caracteres)

1.9 Ejercicios de repaso

1. ¿Qué beneficios nos puede aportar utilizar procedimientos y funciones almacenadas?
2. Según la siguiente sentencia, ¿estamos haciendo una llamada a un procedimiento o a una función?

```
CALL resolver_ejercicio2()
```

3. ¿Cuáles de los siguientes bloques son correctos?

```
1.
LOOP bucle:
  statements
END bucle:

2.
bucle: LOOP
  statements
END bucle;

3.
```

```
bucle:
LOOP bucle;
  statements;

END bucle;
```

4. ¿Pueden aparecer las siguientes sentencias en el mismo bloque de código?

```
DECLARE a INT;

DECLARE a INT;
```

5. ¿Pueden aparecer las siguientes sentencias en el mismo bloque de código?

```
DECLARE a INT;

DECLARE a FLOAT;
```

6. ¿Pueden aparecer las siguientes sentencias en el mismo bloque de código?

```
DECLARE b VARCHAR(20);

DECLARE b HANDLER FOR SQLSTATE '02000';
```

7. ¿Para qué podemos utilizar un cursor en MySQL?
8. ¿Puedo actualizar los datos de un cursor en MySQL? Si fuese posible actualizar los datos de un cursor, ¿se actualizarían automáticamente los datos de la tabla?
9. Cuál o cuáles de los siguientes bucles no está soportado en MySQL: `FOR`, `LOOP`, `REPEAT` y `WHILE`.
10. Si el cuerpo del bucle se debe ejecutar al menos una vez, ¿qué bucle sería más apropiado?
11. ¿Qué valor devuelve la sentencia `SELECT value`?

- 0
- 9
- 10
- `NULL`
- El código entra en un bucle infinito y nunca alcanza la sentencia `SELECT value`

```
DELIMITER //
CREATE PROCEDURE incrementor (OUT i INT)
BEGIN
  REPEAT
    SET i = i + 1;
  UNTIL i > 9
  END REPEAT;
END;

DELIMITER //
CREATE PROCEDURE test ()
BEGIN
  DECLARE value INT default 0;
  CALL incrementor(value);

  -- ¿Qué valor se muestra en esta sentencia?
  SELECT value;
END;

DELIMITER ;
```

```
CALL test();
```

11. ¿Qué valor devuelve la sentencia `SELECT value`?

- 0
- 9
- 10
- `NULL`
- El código entra en un bucle infinito y nunca alcanza la sentencia `SELECT value`

```
DELIMITER //
CREATE PROCEDURE incrementor (IN i INT)
BEGIN
  REPEAT
    SET i = i + 1;
  UNTIL i > 9
  END REPEAT;
END;

DELIMITER //
CREATE PROCEDURE test ()
BEGIN
  DECLARE value INT default 0;
  CALL incrementor(value);

  -- ¿Qué valor se muestra en esta sentencia?
  SELECT value;
END;

DELIMITER ;

CALL test();
```

12. Realice los siguientes procedimientos y funciones sobre la base de datos `jardineria`.

1.

- Función: `calcular_precio_total_pedido`
- Descripción: Dado un código de pedido la función debe calcular la suma total del pedido. Tenga en cuenta que un pedido puede contener varios productos diferentes y varias cantidades de cada producto.
- Parámetros de entrada: `codigo_pedido` (`INT`)
- Parámetros de salida: El precio total del pedido (`FLOAT`)

2.

- Función: `calcular_suma_pedidos_cliente`
- Descripción: Dado un código de cliente la función debe calcular la suma total de todos los pedidos realizados por el cliente. Deberá hacer uso de la función `calcular_precio_total_pedido` que ha desarrollado en el apartado anterior.
- Parámetros de entrada: `codigo_cliente` (`INT`)
- Parámetros de salida: La suma total de todos los pedidos del cliente (`FLOAT`)

3.

- Función: `calcular_suma_pagos_cliente`
- Descripción: Dado un código de cliente la función debe calcular la suma total de los pagos realizados por ese cliente.
- Parámetros de entrada: `codigo_cliente` (`INT`)
- Parámetros de salida: La suma total de todos los pagos del cliente (`FLOAT`)

4.

- Procedimiento: `calcular_pagos_pendientes`
- Descripción: Deberá calcular los pagos pendientes de todos los clientes. Para saber si un cliente tiene algún pago pendiente deberemos calcular cuál es la cantidad de todos los pedidos y los pagos que ha realizado. Si la cantidad de los pedidos es mayor que la de los pagos entonces ese cliente tiene pagos pendientes.

Deberá insertar en una tabla llamada `clientes_con_pagos_pendientes` los siguientes datos:

- `id_cliente`
- `suma_total_pedidos`
- `suma_total_pagos`
- `pendiente_de_pago`

13. Teniendo en cuenta el significado de los siguientes códigos de error:

- Error: 1036 (ER_OPEN_AS_READONLY). Table '%s' is read only
- Error: 1062 (ER_DUP_ENTRY). Duplicate entry '%s' for key %d

```
-- Paso 1
CREATE TABLE t (s1 INT, PRIMARY KEY (s1));

-- Paso 2
DELIMITER //
CREATE PROCEDURE handlerexam(IN a INT, IN b INT, IN c INT, OUT x INT)
BEGIN
  DECLARE EXIT HANDLER FOR 1036 SET x = 10;
  DECLARE EXIT HANDLER FOR 1062 SET x = 30;

  SET x = 1;
  INSERT INTO t VALUES (a);
  SET x = 2;
  INSERT INTO t VALUES (b);
  SET x = 3;
  INSERT INTO t VALUES (c);
  SET x = 4;
END

//
```

¿Qué devolvería la última sentencia `SELECT @x` en cada caso (a y b)? Justifique su respuesta. Sin una justificación válida la respuesta será considerada incorrecta.

```
-- a)
CALL handlerexam(1, 2, 3, @x);
SELECT @x;

-- b)
CALL handlerexam(1, 2, 1, @x);

SELECT @x;
```

14. Dado el siguiente procedimiento:

```
-- Paso 1
CREATE TABLE t (s1 INT, PRIMARY KEY (s1));

-- Paso 2
DELIMITER //
CREATE PROCEDURE test(IN a INT, OUT b INT)
BEGIN
    SET b = 0;
```

```
    WHILE a > b DO
        SET b = b + 1;
        IF b != 2 THEN
            INSERT INTO t VALUES (b);
        END IF
    END WHILE;

END;
```

¿Qué valores tendría la tabla `t` y qué valor devuelve la sentencia `SELECT value` en cada caso (a y b)? Justifique la respuesta. Sin una justificación válida la respuesta será considerada incorrecta.

```
-- a)
CALL test(-10, @value);
SELECT @value;

-- b)
CALL test(10, @value);

SELECT @value;
```

15. Escriba un procedimiento llamado `obtener_numero_empleados` que reciba como parámetro de entrada el código de una oficina y devuelva el número de empleados que tiene.

Escriba una sentencia SQL que realice una llamada al procedimiento realizado para comprobar que se ejecuta correctamente.

16. Escriba una función llamada `cantidad_total_de_productos_vendidos` que reciba como parámetro de entrada el código de un producto y devuelva la cantidad total de productos que se han vendido con ese código.

Escriba una sentencia SQL que realice una llamada a la función realizada para comprobar que se ejecuta correctamente.

17. Crea una tabla que se llame `productos_vendidos` que tenga las siguientes columnas:
- `id` (entero sin signo, auto incremento y clave primaria)
- `codigo_producto` (cadena de caracteres)
- `cantidad_total` (entero)

Escriba un procedimiento llamado `estadísticas_productos_vendidos` que para cada uno de los productos de la tabla `producto` calcule la cantidad total de unidades que se han vendido y almacene esta información en la tabla `productos_vendidos`.

El procedimiento tendrá que realizar las siguientes acciones:

- Borrar el contenido de la tabla `productos_vendidos`.
- Recorrer cada uno de los productos de la tabla `producto`. Será necesario usar un cursor.
- Calcular la cantidad total de productos vendidos. En este paso será necesario utilizar la función `cantidad_total_de_productos_vendidos` desarrollada en el ejercicio 2.
- Insertar en la tabla `productos_vendidos` los valores del código de producto y la cantidad total de unidades que se han vendido para ese producto en concreto.

18. Crea una tabla que se llame `notificaciones` que tenga las siguientes columnas:
- `id` (entero sin signo, autoincremento y clave primaria)
- `fecha_hora`: marca de tiempo con el instante del pago (fecha y hora)

- `total`: el valor del pago (real)
- `codigo_cliente`: código del cliente que realiza el pago (entero)

Escriba un *trigger* que nos permita llevar un control de los pagos que van realizando los clientes. Los detalles de implementación son los siguientes:

- Nombre: `trigger_notificar_pago`
- Se ejecuta sobre la tabla `pago`.
- Se ejecuta *después* de hacer la inserción de un pago.
- Cada vez que un cliente realice un pago (es decir, se hace una inserción en la tabla `pago`), el *trigger* deberá insertar un nuevo registro en una tabla llamada `notificaciones`.

Escriba algunas sentencias SQL para comprobar que el *trigger* funciona correctamente.