Tema 6 – Lenguaje SQL: DML

1º DAW – Bases de Datos

J. Carlos Moreno

Curso 2021/2022

Contenido

6 Lenguaje SQL - DML

6.1 Introducción

Una vez creada la base de datos con sus tablas y restricciones, debemos poder insertar, modificar y borrar los valores de las filas o registros de las tablas. Para poder hacer esto, el SQL92 nos ofrece las siguientes sentencias:

- INSERT para insertar nuevos registros
- UPDATE para modificar o actualizar los valores de dichos registros
- DELETE para realizar una eliminación selectiva de registros.

Una vez hemos insertado valores en nuestras tablas, tenemos que poder consultarlos. La sentencia para hacer consultas a una base de datos con el SQL92 es:

- SELECT FROM.

Estas cuatro comandos SQL conforman el Lenguaje de Manipulación de Datos (DML - Data ManipulationLanguage) siendo el principal componente de Lenguaje SQL.

6.2 Insertar Registros: INSERT

Permite insertar nuevos registros en una tabla, para ello hay que indicar el nombre de la tabla, opcionalmente la lista de campos y los valores asignados a cada campo.

Existen varios métodos de uso de INSERT veamos la sintaxis de cada uno de ellos:

INSERT VALUES

```
INSERT [INTO]tbl_name [(col_name,...)] VALUES
 ({valor|expr | DEFAULT},...)
```

INSERT SET

```
INSERT [INTO] tbl_name SET
col_name={valor|expr | DEFAULT}, ...
```

INSERT SELECT

```
INSERT [INTO] tbl_name [(col_name,...)] SELECT ...
```

6.2.1 INSERT VALUES

Es el método más utilizado y permite insertar registros basados en valores explícitamente especificados.

```
INSERT [LOW_PRIORITY | DELAYED | HIGH_PRIORITY] [IGNORE]
```

```
[INTO] tbl_name [(col_name,...)]
VALUES ({valor | expr | DEFAULT},...),(...),...
[ ON DUPLICATE KEY UPDATE col_name=expr,... ]
```

Observaciones:

Lista de Columnas. [(col_name,...)]

Se refiere a la lista de columnas a las que se les van a asignar los valores establecidos con la cláusula VALUES.

Lista de Valores. ({valor|expr | DEFAULT},...)

Permite especificar la lista de valores de columna.

Observaciones:

- Si no se ha especificado la Lista de Columnas. La lista de valores se ha de mostrar en el orden de creación de las columnas establecidos en CREATE TABLE o ALTER TABLE.Si no se conoce dicho orden usar el comando *DESCRIBE NombreTabla*o buscarlo en el panel izquierdo de wordkbench.
- Si se ha especificado la Lista de Columnas el orden de la Lista de Valores debe coincidir con el de las columnas.
- Si una columna no se ha especificado en la Lista de Columnas, se le asigna el valor NULL o el establecido por defecto con DEFAULT.
- No es necesario especificar un valor para la columna AUTO_INCREMENT.
- Puede especificar una expresión *expr* para proporcionar un valor de columna. Esto puede involucrar conversión de tipos si el tipo de la expresión no coincide con el tipo de la columna, y la conversión de un valor dado puede resultar en distintos valores insertados dependiendo del tipo de columna. Una expresión expr puede referirse a cualquier columna que se haya asignado antes en una lista de valores. Por ejemplo, puede hacer esto porque el valor para col2 se refiere a col1, que se ha asignado previamente:

```
INSERT INTO tbl_name(col1,col2) VALUES(15,col1*2);
```

[LOW_PRIORITY | DELAYED | HIGH_PRIORITY]

- DELAYED, el servidor pone el registro o registros a ser insertados en un buffer, y cuando la tabla se libera, el servidor comienza a insertar registros, chequeando periódicamente para ver si hay alguna petición de lectura para la tabla. Si la hay, la cola de registros retardados se suspende hasta que la tabla se libera de nuevo. Se realiza la inserción poco a poco, cuando se puede.
- LOW_PRIORITY, la ejecución de INSERT se retrasa hasta que no hay otros clientes leyendo de la tabla y entonces se procesan las inserciones de un golpe.Es posible, por lo tanto, para un cliente que realice un comando INSERT con LOW_PRIORITY que tenga

que esperar durante mucho tiempo (o incluso para siempre) en un entorno de muchas lecturas.

- HIGH_PRIORITY, deshabilita el efecto de la opción - low-priority-

[IGNORE]

Si usa la palabra IGNORE en un comando INSERT , los errores que ocurren mientras se ejecuta el comando se tratan como advertencias. Por ejemplo, sin IGNORE, un registro que duplique un índice UNIQUE existente o valor PRIMARY KEY en la tabla hace que un error de clave duplicada en el comando se aborte. Con IGNORE, el registro todavía no se inserta, pero no se muestra error.

[ON DUPLICATE KEY UPDATE *col_name*=*expr*,...]

Si especifica ON DUPLICATE KEY UPDATE, y se inserta un registro que duplicaría un valor en un índice UNIQUE o PRIMARY KEY, se realiza un UPDATE del antiguo registro. Por ejemplo, si la columna a se declara como UNIQUE y contiene el valor 1, los siguientes dos comandos tienen efectos idénticos:

```
INSERT INTO table (a,b,c) VALUES (1,2,3)
ON DUPLICATE KEY UPDATE c=c+1;

UPDATE table SET c=c+1 WHERE a=1;
```

Veamos algunos ejemplos:

```
USE TEST;
DROP TABLE IF EXISTS temas;
CREATE TABLE IF NOT EXISTS temas(
        id int unsigned AUTO_INCREMENT PRIMARY KEY,
        tema VARCHAR(30)
);

-- Se especifica el valor de la columna AUTO_INCREMENT, aunque no hace
-- falta. En vez de poner el valor podría haber puesto directamente
-- NULL.

INSERT INTO `temas` VALUES
(1,'Informática'),
(2,'Matemáticas'),
(3,'Novela'),
(4,'Viajes'),
(5,'Belleza'),
(6,'Deportes'),
(7,'Astronomía');

DROP TABLE IF EXISTS ventas;
CREATE TABLE ventas(
        Id int unsigned AUTO_INCREMENT,
        cliente_id int unsigned,
        fechaVenta TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
        importeBruto decimal(10, 2 ) DEFAULT 0.0,
        importeIva decimal(10, 2 ) DEFAULT 0.0,
        importeTotal decimal(10, 2 ) DEFAULT 0.0,
        PRIMARY KEY (IdVenta),
```

```
        FOREIGN KEY (cliente_id)REFERENCES clientes (id)
       ON DELETE CASCADE ON UPDATE CASCADE
);

INSERT INTO ventas VALUES
        (1,3,NULL,DEFAULT,DEFAULT,DEFAULT),
        (2,6,NULL,364.00,76.44,440.44),
        (3,4,DEFAULT,256.00,53.71,309.76),
        (4,7,'2014-01-15',1311.00,275.31,1586.31),
        (5,5,'2014-03-15',1129.20,237.13,1366.33),
        (6,1,'2014-03-24',1617.5,339.68,1957.18),
        (7,7,'2014-03-26',2787.00,585.27,3372.27),
        (8,6,'2014-03-25',633.00,132.93,765.93),
        (9,7,'2014-03-25',430.00,90.30,520.30),
        (10,1,'2014-03-21',243.00,51.30,294.03);

DROP TABLE IF EXISTS Articulos;

CREATE TABLE Articulos (
        id int unsigned PRIMARY KEY AUTO_INCREMENT,
        Nombre varchar(30) NOT NULL,
        precio decimal(10,2) NOT NULL,
        codigo Char(7) NOT NULL,
        familia int unsigned
) ENGINE=MyISAM DEFAULT CHARSET=latin1;

-- Se especifica la lista de columnas
-- Las columnas no incluidas se les asigna el valor establecido
-- por defecto con DEFAULT.

INSERT INTO articulos(id, Nombre, Familia) VALUES
        (NULL,'Monitor 16',1),
        (NULL,'Monitor 20',1),
        (NULL,'Monitor 22',2),
        (NULL,'Motherboard FX',3),
        (DEFAULT,'Papel A4-500',2),
        (NULL,'Diskettes 10',2),
        (NULL,'Diskettes 20',2);
```

6.2.2 INSERT SET

Este método también permite insertar registros basados en los valores explícitamente especificados.

La sintaxis completa sería la siguiente

```
INSERT [LOW_PRIORITY | DELAYED | HIGH_PRIORITY] [IGNORE]
[INTO] tbl_name
SET col_name={expr | DEFAULT},...
[ ON DUPLICATE KEY UPDATE col_name=expr,... ]
```

En este método no existe la lista de columna sino que después del SET se especifica directamente el valor que se le asigna a cada una de las columnas.Las columnas a las que no se les ha asignado valor se le asigna NULL o el valor establecido con DEFAULT.

```
USE TEST;
DROP TABLE IF EXISTS equipos;
CREATE TABLE equipos (
        id int unisigned AUTO_INCREMENT,
        nombre varchar(50) NOT NULL,
```

```
        estadio varchar(50) NOT NULL,
        aforo int DEFAULT NULL,
        ano_fundacion year DEFAULT NULL,
        ciudad varchar(50) DEFAULT NULL,
        PRIMARY KEY (id)
  ) ENGINE=InnoDB DEFAULT CHARSET=latin1;

-- Añade 3 registros con INSERT SET

INSERT INTO equipos SET
        id = 200,
        nombre = 'Real Betis',
        estadio = 'Benito Villamarín',
        ciudad = 'Sevilla';

INSERT INTO equipos SET
        id = 201,
        nombre = 'Cádiz CF',
        estadio = 'Ramón de Carranza',
        ciudad = 'Cádiz',
        ano_fundacion = '1912';
```

6.2.3 INSERT SELECT

Inserta nuevos registros en una tabla existente a partir de los seleccionados con una sentencia SELECT.

```
INSERT [LOW_PRIORITY | HIGH_PRIORITY] [IGNORE]
[INTO] tbl_name [(col_name,...)]
SELECT...
[ ON DUPLICATE KEY UPDATE col_name=expr,... ]
```

Este método lo usaremos más adelante cuando se estudie la sentencia SELECT.

6.3 Actualizar Registros. UPDATE

El comando UPDATE actualiza columnas en registros de tabla existentes con nuevos valores. La cláusula SET indica qué columna modificar y los valores que puede recibir.

La sintaxis es la siguiente

```
UPDATE [LOW_PRIORITY] [IGNORE] tbl_name
SET col_name1=expr1 [, col_name2=expr2...]
[WHERE where_definition]
[ORDER BY...][LIMIT row_cont]
```

La cláusula WHERE, si se da, especifica qué registros deben actualizarse. De otro modo, se actualizan todos los registros. Si la cláusula ORDER BY se especifica, los registros se actualizan en el orden que se especifica. La cláusula LIMIT es el límite de registros a actualizar.

El comando UPDATE admite los siguientes modificadores:

- Si usa la palabra clave LOW_PRIORITY , la ejecución de UPDATE se retrasa hasta que no haya otros clientes leyendo de la tabla.
- Si usa la palabra clave IGNORE, el comando de actualización no aborta incluso si ocurren errores durante la actualización. Los registros que presenten conflictos de clave duplicada no se actualizan.

Los registros cuyas columnas se actualizan a valores que provocarían errores de conversión de datos se actualizan al valor válido más próximo.

Si accede a una columna de tbl_name en una expresión, UPDATE usa el valora ctual de la columna.Por ejemplo, el siguiente comando pone la columna age a uno más que su valor actual:

```
UPDATE persondata SET age=age+1;
```

Las asignaciones UPDATE se avalúna de izquierda a derecha. Por ejemplo, el siguiente comando dobla la columna age y luego la incrementa:

```
UPDATE persondata SET age=age*2, age=age+1;
```

Si actualiza una columna declarada como NOT NULL con un valor NULL, la columna recibe el valor por defecto apropiado para el tipo de la columna y se incrementa el contador de advertencias. El valor por defecto es 0 para tipos numéricos, la cadena vacía ('') para tipos de cadena, y el valor “cero” para valores de fecha y hora.

Puede usar LIMIT row_count para restringir el alcance del UPDATE. Una cláusula LIMIT es una restricción de registros coincidentes. El comando para en cuanto encuentra row_countregistos que satisfagan la cláusula WHERE , tanto si han sido cambiados como si no.

Veamos los siguientes ejemplos:

```
CREATE TABLE IF NOT EXISTS Alumnos(
        id INT AUTO_INCREMENT PRIMARY KEY,
        nombre VARCHAR(20),
        apellidos VARCHAR(40),
        poblacion VARCHAR(20),
        fechaIngreso TIMESTAMP,
        repetidor BOOLEAN,
        beca BOOLEAN,
        importeBeca DECIMAL(8, 2 )
);

-- Incrementar en 1.000 € el importe de la Beca
-- de los 4 primeros alumnos ordenados alfabéticamente.
UPDATE Alumnos
SET importeBeca = importeBeca + 1000
ORDER BY apellidos, nombre Limit 4;

-- Incrementar en 10 € la Beca de los
-- 10 alumnos con menor importe de beca asignado.
UPDATE Alumnos
SET importeBeca=importeBeca+10
ORDER BY importeBeca ASC Limit 10;
```

```
-- Todos los alumnos repetidores se le aplicará un ImporteBeca = 0
-- y se les rechazará la beca
UPDATE Alumnos
SET importeBeca=0,
    Beca=FALSE
WHERE Repetidor;

-- Incrementar en 100 € el importe de la beca de
-- los alumnos que no sean de Ubrique.
-- Aplicar sólo a los que se le ha concedido la beca.
UPDATE Alumnos
SET importeBeca=importeBeca + 100
WHERE beca AND NOT TRIM(poblacion)='Ubrique';
```

6.4 Borrar Registros: DELETE

Permite un borrado selectivo de los registros de una tabla existente.

La sintaxis es la siguiente

```
DELETE [LOW_PRIORITY] [QUICK] [IGNORE] FROM tbl_name
[WHERE where_definition]
[ORDER BY...]
[LIMIT row_count]
```

DELETE borra los registros de tbl_name que satisfacen la condición dada por where_definition, y retorna el número de registros borrados.

Si realiza un comando DELETE sin cláusula WHERE se borran todos los registros. Una forma más rápida de haceresto último, cuando no quiere saber el número de registros borrados, es a través de TRUNCATE TABLE sentencia vista en el tema anterior.

El comando DELETE soporta los siguientes modificadores:

- Si especifica LOW_PRIORITY, la ejecución de DELETE se retarda hasta que no hay más clientes leyendo de la tabla.
- Si usa la palabra QUICK , el motor de almacenamiento acelerará algunos tipos de operaciones de borrado.
- La palabra clave IGNORE hace que MySQL ignore todos los errores durante el proceso de borrar registros.
- La opción de LIMIT row_count para DELETE le dice al servidor el máximo número de registros a borrar antes de retornar el control al cliente. Esto puede usarse para asegurar que un comando DELETE específico no tarde demasiado tiempo. Puede simplemente repetir el comando DELETE hasta que el número de registros afectados sea menor que el valor LIMIT.
- Si el comando DELETE incluye una cláusula ORDER BY , los registros se borran en el orden especificado por la cláusula. Esto es muy útil sólo en conjunción con LIMIT.
- Si borra el registro conteniendo el máximo valor para una columna AUTO_INCREMENT el valor no se reusa

Veamos algunos ejemplos

```
CREATE TABLE IF NOT EXISTS Alumnos(
        id INT AUTO_INCREMENT PRIMARY KEY,
        Nombre VARCHAR(20),
        Apellidos VARCHAR(40),
        PoblacionVARCHAR(20),
        FechaIngreso TIMESTAMP,
        Repetidor BOOLEAN,
        Beca BOOLEAN,
        ImporteBeca DECIMAL(10, 2 )
);

-- Borrar todos los registros de la tabal Alumnos
DELETE FROM Alumnos;

-- Otra opción más rápida sería
TRUNCATE TABLE Alumnos;

-- Eliminar todos los alumnos repetidores
DELETE FROM Alumnos WHERE Repetidor;

-- Eliminar el primer alumno repetidor ordenado alfabéticamente
DELETE FROM Alumnos WHERE Repetidor ORDER BY Apellidos, Nombre
LIMIT 1;

-- Eliminar los dos alumnos con mayor importe de beca
DELETE FROM Alumnos ORDER BY ImporteBeca DESC LIMIT 2;
```

6.5 Base de Datos Maratoon: maratoon.sql

En este apartado se va a mostar un script completo de creación de la base de datos **maratoon**. Sobre esta base de datos estarán basados la mayoría de los ejemplos contenidos en el comando **SELECT**.

El script de creación de la base de datos **maratonn** es el siguiente:

```
-- SCRIPT SQL PARA LA CREACIÓN DE LA
-- BASE DE DATOS MARATOON
-- Versión 27 de marzo 2020

DROP DATABASE IF EXISTS maratoon;
CREATE DATABASE maratoon;
USE maratoon;
--
-- tabla categorías
--
DROP TABLE IF EXISTS Categorias;
CREATE TABLE IF NOT EXISTS Categorias(
 id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
 Nombrecorto CHAR(3) UNIQUE,
 Nombre VARCHAR(20),
 Descripcion VARCHAR(30)
);
--
-- Tabla Carreras

DROP TABLE IF EXISTS Carreras;
CREATE TABLE IF NOT EXISTS Carreras(
 id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
 Nombre VARCHAR(30),
 Ciudad VARCHAR(20),
```

```
 Distancia INT UNSIGNED,
 MesCelebracion TINYINT UNSIGNED
);
--
-- Tabla de Club
--
DROP TABLE IF EXISTS Clubs;
CREATE TABLE IF NOT EXISTS Clubs(
 id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
 NombreCorto CHAR(3) UNIQUE,
 Nombre VARCHAR(30),
 Ciudad VARCHAR(20),
 FecFundacion DATE,
 NumSocios INT UNSIGNED
);
--
-- Tabla Corredores
--
DROP TABLE IF EXISTS Corredores;
CREATE TABLE IF NOT EXISTS Corredores(
 id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
 Nombre VARCHAR(20),
 Apellidos VARCHAR(45),
 Ciudad VARCHAR(30),
 FechaNacimiento DATE NOT NULL,
 Sexo ENUM(' ', 'H', 'M') DEFAULT ' ',
 Edad INT(2) UNSIGNED,
 Categoria_id INT UNSIGNED,
 Club_id INT UNSIGNED,
 FOREIGN KEY (categoria_id)REFERENCES Categorias (id)
 ON DELETE RESTRICT ON UPDATE CASCADE,
 FOREIGN KEY (club_id)REFERENCES Clubs (id)
ON DELETE RESTRICT ON UPDATE CASCADE
);
-- Continuación script anterior
--
-- Tabla Registros
--
-- Registro de los tiempos invertidos por cada corredor en cada carrera
-- en la que participa
DROP TABLE IF EXISTS Registros;
CREATE TABLE Registros(
        id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
        carrera_id INT UNSIGNED,
        corredor_id INT UNSIGNED,
        Salida TIMESTAMP(6),
        Llegada DATETIME(6),
        TiempoInvertido TIME(6),
         FOREIGN KEY (carrera_id)REFERENCES Carreras (id)
        ON DELETE RESTRICT ON UPDATE CASCADE,
         FOREIGN KEY (corredor_id)REFERENCES Corredores (id)
        ON DELETE RESTRICT ON UPDATE CASCADE
);

-- Tabla Categoria
INSERT INTO Categorias VALUES
(1, 'INF', 'Infantil', 'Menores de 12 años'),
(2, 'JUN', 'Junior', 'Menores de 15 años'),
(3, 'JUV', 'Juvenil', 'Menores de 18 años'),
(4, 'SNA', 'Senior A', 'Menores de 30 años'),
(5, 'SNB', 'Senior B', 'Menores de 40 años'),
(6, 'VTA', 'Veteranos A', 'Menores de 50 años'),
(7, 'VTB', 'Veteranos B', 'Menores de 60 años'),
(8, 'VTC', 'Veteranos C', 'A partir de 60 años');
--
-- Tabla Carrera
INSERT INTO Carreras VALUES
```

```
(1, 'Nutrias Pantaneras', 'Ubrique', 11000, 10),
(2, 'Media Martoon Sevilla', 'Sevilla', 22000, 6),
(3, 'Media Martoon Jerez', 'Jerez', 22000, 5),
(4, 'Salida Sanluqueña', 'Sanlucar de Bda.', 11500, 3);
--
-- Tabla Carrera
INSERT INTO Clubs VALUES
(1, 'NUT', 'Nutrias Pantaneras', 'Ubrique', '1980-4-11', 150),
(2, 'CAV', 'Club Atletismo Villamartín', 'Villamartín', '1985-4-11', 200),
(3, 'ACG', 'Atletismo Campo de Gibraltar', 'La Línea', '1987-4-11', 150),
(4, 'ADJ', 'Asociación Deportiva de Arcos', 'Arcos de la Fra.', '1970-4-11', 300),
(5, 'CAF', 'Club Atletismo Fronter', 'Jerez de la Fra.', '1975-4-11', 220);

-- Tabla Corredor
INSERT INTO Corredores VALUES
(1,'Juan','García Pérez','Ubrique','1965-7-31', 'H', NULL, NULL, 1),
(2,'Juan José','Pérez Morales','Ubrique','1945-8-30', 'H', NULL, NULL, 1),
(3,'Eva','Rubiales Alva','Ubrique','1980-8-25', 'M', NULL, NULL, 1),
(4,'Josefa','Rios Pérez','Villamartín','1990-10-15', 'M', NULL, NULL, 2),
(5,'Pedro','Ortega Ríos','Villamartín','1994-5-14', 'H', NULL, NULL, 2),
(6,'Francisco','Morales Almeida','Villamartín','1970-2-1', 'H', NULL, NULL, 2),
(7,'Macarena','Fernández Pérez','Villamartín','1980-5-3', 'M', NULL, NULL, 2),
(8,'Jesús','Romero Reguera','Villamartín','1970-6-5', 'H', NULL, NULL, 2),
(9,'Pedro','García Ramírez','Ubrique','1967-7-31', 'H', NULL, NULL, 1),
(10,'María','Pérez Moreno','Ubrique','1975-8-30', 'M', NULL, NULL, 1),
(11,'Almudena','Romero Alva','Arcos','1986-8-25', 'M', NULL, NULL, 4),
(12,'Francisco','Pérez Amor','Arcos','1992-10-15', 'H', NULL, NULL, 4),
(13,'Juan','Rodríguez Ríos','Ubrique','1978-5-14', 'H', NULL, NULL, 1),
(14,'Cristina','García Almeida','Villamartín','1978-2-1', 'M', NULL, NULL, 2),
(15,'Romira','Jiménez Pérez','Arcos','1984-5-3', 'M', NULL, NULL, 4),
(16,'José','Rincón Pérez','Villamartín','1960-6-5', 'H', NULL, NULL, 2);
--
-- Tabla Registro
INSERT INTO registros VALUES
(1, 1, 1,'2012-4-11 10:00:00.000000','2012-4-11 10:45:10.000012', NULL),
(2, 1, 2,'2012-4-11 10:00:00.000000','2012-4-11 10:35:10.00067', NULL),
(3, 1, 3,'2012-4-11 10:00:00.000000','2012-4-11 10:37:10.00148', NULL),
(4, 1, 4,'2012-4-11 10:00:00.000000','2012-4-11 10:36:20.001546', NULL),
(5, 1, 5,'2012-4-11 10:00:00.000000','2012-4-11 10:35:40.000333', NULL),
(6, 1, 6,'2012-4-11 10:00:00.000000','2012-4-11 10:40:01.000164', NULL),
(7, 1, 7,'2012-4-11 10:00:00.000000','2012-4-11 10:30:30.009412', NULL),
(8, 1, 8,'2012-4-11 10:00:00.00000','2012-4-11 10:38:10.000754', NULL),
(9, 1, 9,'2012-4-11 10:00:00.00000','2012-4-11 10:48:10.000002', NULL),
(10, 1, 10,'2012-4-11 10:00:00.00000','2012-4-11 10:39:10.000003', NULL),
(11, 1, 11,'2012-4-11 10:00:00.00000','2012-4-11 10:55:10.001483', NULL),
(12, 1, 12,'2012-4-11 10:00:00.00000','2012-4-11 10:50:20.000004', NULL),
(13, 1, 13,'2012-4-11 10:00:00.00000','2012-4-11 10:58:40.000005', NULL),
(14, 1, 14,'2012-4-11 10:00:00.00000','2012-4-11 11::01.000005', NULL),
(15, 1, 15,'2012-4-11 10:00:00.00000','2012-4-11 11:10:30.009068', NULL),
(16, 1, 16,'2012-4-11 10:00:00.00000','2012-4-11 11:09:10.000500', NULL);

-- Actualiza el campo Edad a partir de la Fecha de Nacimiento del corredor
--
UPDATE Corredores SET Edad = TIMESTAMPDIFF(YEAR,FechaNacimiento,NOW());
--
-- Actualizar la columna CodCategoría de la tabla corredor, teniendo -- -- en -
-- cuenta la Edad y el cuadrante de la tabla categorías
--
UPDATE Corredores SET categoria_id = 1 WHERE Edad<12;
UPDATE Corredores SET categoria_id = 2 WHERE Edad BETWEEN 12 AND 14;
UPDATE Corredores SET categoria_id = 3 WHERE Edad BETWEEN 15 AND 17;
UPDATE Corredores SET categoria_id = 4 WHERE Edad BETWEEN 18 AND 29;
UPDATE Corredores SET categoria_id = 5 WHERE Edad BETWEEN 30 AND 39;
UPDATE Corredores SET categoria_id = 6 WHERE Edad BETWEEN 40 AND 49;
UPDATE Corredores SET categoria_id = 7 WHERE Edad BETWEEN 50 AND 60;
UPDATE Corredores SET categoria_id = 8 WHERE Edad>=60;
--
```

```
-- Actualizar la columna Tiempoinvertido de la tabla Registros,
-- a partir de las columnas Salida y Llegada.
--
UPDATE registros SET TiempoInvertido = TIMEDIFF(Llegada,Salida);
```

6.5.1 Tablas de Datos

Tabla 1. Tabla Carreras

TABLA CARRERAS				
id	Nombre	Ciudad	Distancia	MesCelebracion
1	Nutrias Pantaneras	Ubrique	11000	10
2	Media Martoon Sevilla	Sevilla	22000	6
3	Media Martoon Jerez	Jerez	22000	5
4	Salida Sanluqueña	Sanlucar de Bda.	11500	3

Tabla 2. Tabla Categorias

TABLA CATEGORIAS			
id	Nombrecorto	Nombre	Descripcion
1	INF	Infantil	Menores de 12 años
2	JUN	Junior	Menores de 15 años
3	JUV	Juvenil	Menores de 18 años
4	SNA	Senior A	Menores de 30 años
5	SNB	Senior B	Menores de 40 años
6	VTA	Veteranos A	Menores de 50 años
7	VTB	Veteranos B	Menores de 60 años
8	VTC	Veteranos C	A partir de 60 años

Tabla 3. Tabla Clubs

Tabla Clubs					
id	NombreCorto	Nombre	Poblacion	FecFundacion	Numsocios
1	NUT	Nutrias Pantaneras	Ubrique	1980-04-11	150
2	CAV	Club Atletismo Villamartín	Villamartín	1985-04-11	200
3	ACG	Atletismo Campo de Gibraltar	La Línea	1987-04-11	150
4	ADJ	Asociación Deportiva de Arcos	Arcos de la Fra.	1970-04-11	300
5	CAF	Club Atletismo Fronter	Jerez de la Fra.	1975-04-11	220

Tabla 4. Tabla Corredores

Tabla Corredores

id	Nombre	Apellidos	Poblacion	Fnacimiento	Sexo	Edad	Categoria_id	Club_id
1	Juan	García Pérez	Ubrique	1965-07-31	H	50	7	1
2	Juan José	Pérez Morales	Ubrique	1945-08-30	H	70	8	1
3	Eva	Rubiales Alva	Ubrique	1980-08-25	M	35	5	1
4	Josefa	Rios Pérez	Villamartín	1990-10-15	M	25	4	2
5	Pedro	Ortega Ríos	Villamartín	1994-05-14	H	21	4	2
6	Francisco	Morales Almeida	Villamartín	1970-02-01	H	46	6	2
7	Macarena	Fernández Pérez	Villamartín	1980-05-03	M	35	5	2
8	Jesús	Romero Reguera	Villamartín	1970-06-05	H	45	6	2
9	Pedro	García Ramírez	Ubrique	1967-07-31	H	48	6	1
10	María	Pérez Moreno	Ubrique	1975-08-30	M	40	6	1
11	Almudena	Romero Alva	Arcos	1986-08-25	M	29	4	4
12	Francisco	Pérez Amor	Arcos	1992-10-15	H	23	4	4
13	Juan	Rodríguez Ríos	Ubrique	1978-05-14	H	37	5	1
14	Cristina	García Almeida	Villamartín	1978-02-01	M	38	5	2
15	Romira	Jiménez Pérez	Arcos	1984-05-03	M	31	5	4
16	José	Rincón Pérez	Villamartín	1960-06-05	H	55	7	2

Tabla 5. Tabla Registros

Tabla Registros					
id	**Carrera_id**	**Corredor_id**	**Salida**	**Llegada**	**TiempoInvertido**
1	1	1	2012-04-11 10:00:00	2012-04-11 10:45:10	00:45:10
2	1	2	2012-04-11 10:00:00	2012-04-11 10:35:10	00:35:10
3	1	3	2012-04-11 10:00:00	2012-04-11 10:37:10	00:37:10
4	1	4	2012-04-11 10:00:00	2012-04-11 10:36:20	00:36:20
5	1	5	2012-04-11 10:00:00	2012-04-11 10:35:40	00:35:40
6	1	6	2012-04-11 10:00:00	2012-04-11 10:40:01	00:40:01
7	1	7	2012-04-11 10:00:00	2012-04-11 10:30:30	00:30:30
8	1	8	2012-04-11 10:00:00	2012-04-11 10:38:10	00:38:10
9	1	9	2012-04-11 10:00:00	2012-04-11 10:48:10	00:48:10
10	1	10	2012-04-11 10:00:00	2012-04-11 10:39:10	00:39:10
11	1	11	2012-04-11 10:00:00	2012-04-11 10:55:10	00:55:10
12	1	12	2012-04-11 10:00:00	2012-04-11 10:50:20	00:50:20
13	1	13	2012-04-11 10:00:00	2012-04-11 10:58:40	00:58:40
14	1	14	2012-04-11 10:00:00	2012-04-11 11:01:00	01:01:00
15	1	15	2012-04-11 10:00:00	2012-04-11 11:10:30	01:10:30
16	1	16	2012-04-11 10:00:00	2012-04-11 11:09:10	01:09:10

6.6 ConsultasBásicas: SELECT

Sentencia SQL que permite recuperar información de la Base de Datos mediante la realización de consultas sobre una o más tablas.

Mediante esta sentencia el usuario especifica que es lo que quiere obtener, no dónde ni cómo, característica propia de un lenguaje no procedimental como es SQL.

El resultado de una consulta SELECT siempre será un subconjunto de datos extraído de la Base de Datos.

La sintaxis reducida de la sentencia SELECT es la siguiente:

```
SELECT [ALL | DISTINCT]
expre_colum1, expre_colum2, ...
FROM [table_name1, table_name2, …]
[WHERE where_definition]

[GROUP BY {col_name | expr | position}
[ASC | DESC], ... [WITH ROLLUP]]
[HAVING where_definition]
[ORDER BY {col_name | expr | position}
[ASC | DESC] , ...]
[LIMIT {[offset,] row_count | row_count OFFSET offset}]
```

Todas las cláusulas deben usarse exactamente en el orden mostrado en la descripción de la sintaxis. Por ejemplo, una cláusula HAVING debe ir tras cualquier cláusula GROUP BY y antes de cualquier cláusula ORDER BY .

6.6.1 Lista de Columnas: expre_colum

Especifica la lista de columnas que van a intervenir en la consulta de selección

En la lista de columnas podemos usar las siguientes literales:

- Asterisco **(*)**. Cuando usamos el asterisco el resultado de la consulta mostrará todas las columnas de la tabla o tablas que participan en dicha consulta. Ver ejemplos.
- **Lista de Columnas**. Especificamos una lista de columnas de la tabla o tablas que participan en la consulta, el resultado sólo mostrará estas columnas. Ver ejemplos.
- Cláusula **AS**. Permite renombrar las columnas que queremos seleccionar o lo que es lo mismo asignarles un alias. Para ello especificaremos el nombre de la columna a continuación AS y seguido el alias o nuevo nombre. La palabra clave AS es opcional y es bastante habitual poner un espacio en blanco entre la expresión y el alias. En caso de establecer un alias para las columnas siempre se recomienda el uso de AS para dar mayor claridad al código. Ver ejemplos.
- **Prefijos**. Los prefijos se utilizan en caso de referencias ambiguas a los nombres de las columnas. Una referencia a una columna puede ser ambigua sólo cuando se realiza una consulta multitabla donde existen nombres de columnas comunes, en esos casos estoy obligado a poner el prefijo nombre de tabla al nombre de dicha columna. Pero no sólo puedo usar varias tablas de una misma base de datos en una consulta SELECT, además, puedo usar tablas de distintas bases de datos, por lo que estaré obligado a anteponer el prefijo del nombre de la base de datos el prefijo del nombre de la tabla al nombre de la columna seleccionada.
- **Expresiones**. Las expresiones podrán contener nombre de columnas, operadores y funciones. Hay que tener en cuenta que el orden de evaluación de una expresión es de izquierda a derecha. Normalmente cuando se usa una expresión en la lista de columnas se le asigna un alias para mejorar la claridad del resultado.

Veamos los siguientes ejemplos a partir de la base de datos maratoon

```
-- BASE DE DATOS MARATOON
-- Ejemplos Lista Columnas
USE maratoon;

-- Ejemplo 01 - Uso '*' en lista columnas
-- Muestra todas las columnas de la tabla corredores
SELECT * FROM corredores;

-- Ejemplo 02 - Espcificación de una lista de columnas
-- Muestra de la tabla corredor sólo las columnas especificadas en la lista
SELECT id, Apellidos, Nombre, Edad FROM corredores;

-- EJEMPLO 03
SELECT id, Apellidos, Nombre, categoría_id FROM corredores;

-- Ejemplo 04 - Cláusula AS
-- Especificación de columnas con asignación de alias a algunas columnas
SELECT
       Id  AS `Código`, Apellidos, Nombre, club_id AS Club
FROM
       corredores;

-- Ejemplo 05 - Renombrar columnas sin AS
-- Veamos a continuación que la cláusula AS no es obligatoria
SELECT
       id `Código`, Apellidos, Nombre, club_id Club
FROM
       corredores;

-- Ejemplo 06 - Prefijos
-- En este ejemplo vemos que el prefijo nombre de tabla es obligatorio sólo
-- para el campo Nombre, ya que dicha columna se repite en ambas tablas
SELECT
       corredores.id,
       corredores.Nombre,
       corredores.Apellidos,
       carreras.Nombre,
       carreras.Ciudad
FROM
       corredores,
       carreras;

-- Ejemplo 07 - Prefijos con alias en las tablas
SELECT
       co.id, co.Nombre, co.Apellidos, ca.Nombre, ca.Ciudad
FROM
       corredores AS co,
       carreras AS ca;

-- Ejemplo 08 - Expresiones en la lista de columnas
Select
       id,
       Concat(Apellidos, ', ', Nombre) AS Nombre,
       year(FechaNacimiento) As 'Año Nacimiento',
       Edad
from
       corredores;
```

6.6.2 Cláusula FROM

Permite especificar la tabla o conjunto de tablas a partir de la cual se extraerán los datos de la consulta.

- **Prefijos**. En caso que la tabla pertenezca a una base de datos distinta de la actual habrá que especificar el prefijo nombre de la base de datos en el nombre de la tabla.
- **Cláusula AS**. Se puede hacer referencia a una tabla mediante el uso de un alias, este alias se utiliza para hacer más breve el prefijo en las lista de campos multitablas. Al igual que en las columnas se puede omitir dicha cláusula dejándo simplemente un espacio en blanco entre el nombre de la tabla y el nombre del alias.

```
-- Ejemplos Cláusula FROM
--

-- Ejemplo 09 - Prefijos con alias en las tablas
--
SELECT
       co.id, co.Nombre, co.Apellidos, ca.Nombre, ca.Ciudad
FROM
       corredores AS co,
       carreras AS ca;

-- Ejemplo 10 - Prefijos con alias en las tablas omitiendo AS
--
SELECT
       co.id, co.Nombre, co.Apellidos, ca.Nombre, ca.Ciudad
FROM
       corredores co,
       carreracas ca;
```

6.6.3 Cláusula WHERE

La cláusula WHERE permite expresar un condición de forma que sólo quedarán seleccionados en la consulta aquellos registros que cumplan dicha condición. Se pueden crear condiciones de búsquedas complejas que proporcionan una gran potencia de selección de filas. A estas condiciones en MySQL se denominan predicados.

Un **predicado** expresa una condición que se cumple o no sobre un conjunto de valores o expresiones y el resultado de su evaluación puede ser Verdadero o Falso.

Los predicados se expresan normalmente junta a la cláusula WHERE.

Sólo se considera satisfecha la condición de búsqueda expresada en un predicado cuando toma el valor “Verdadero”. Esto quiere decir que el resultado de la evaluación de un predicado da lugar a la recuperación de las filas para las que toma el valor “Verdadero” y se rechazará para las que tome el valor “Falso”

Para definir el **predicado** de la cláusula WHERE podemos usar la siguiente sintáxis:

```
Expresión Operador Expresión
```

Como **expresión** podemos usar:

- Nombre de una columna
- Valor de una constante
- Expresión matemática

- Valor Nulo (NULL)
- Función MYSQL

Como **operador** podremos usar los operadores de comparación:

- =, <=>, >=, >, <=, <, <>, !=

Otros operadores de comparación:

- IN, BETWEEN, LIKE

Operadores lógicos son necesarios para crear condiciones compuestas en tal caso se podrán usar los paréntesis para forzar el orden de evaluación

- NOT – Para la negación de la condición
- AND - Para la conjunción de condiciones
- OR - Para la disyunción de condiciones

Veamos algunos ejemplos

```
-- Ejemplos cláusula WHERE
--
-- Ejemplo 11
-- Corredores con 35 años de edad
SELECT * FROM corredores WHERE edad=35;

-- EJEMPLO 12
-- Corredores Masculinos con más de 30 años
SELECT * FROM corredores WHERE edad>30 AND sexo='M';

-- Ejemplo 13
-- Corredores de Villamartín, nacidos en el 1980
SELECT * FROM corredores
WHERE YEAR(fechanacimiento)=1980 AND Ciudad = 'Villamartín';

-- Ejemplo 14
-- Corredores Masculinos de la Categoría 5
SELECT * FROM corredores WHERE Sexo='M' AND categoria_id='5';

-- Ejemplo 15
-- Corredores que no tengan indicado el club al que pertenecen
SELECT * FROM CORREDOR WHERE club_id IS NULL;
```

6.6.4 Otros operadores de comparación

6.6.4.1 IS NULL

Comprueba si un valor es o no NULL.

Ya sabemos que un valor NULL es un valor vacío, se recuerda:

- Valor cero para campo numérico no es NULL
- Espacio para campo alfanumérico no es NULL

Sintaxis

```
Valor IS NULL | IS NOT NULL
```

```
-- Ejemplo 16
-- Corredores que no tengan indicado el club al que pertenecen
SELECT * FROM corredores WHERE club_id IS NULL;

-- Ejemplo 17
-- Corredores que tengan asignada la ctegoría
SELECT * FROM corredores WHERE categoria_id IS NOT NULL;
```

6.6.4.2 IN

Nos permite comprobar si una expresión está incluida dentro de una lista de valores. Este operadorhace posible la realización de comparaciones múltiples.

Sintaxis

```
Expresión [NOT] IN (Lista_Valores)
```

Expresión normalmente es el nombre de una columna

La lista de valores va separadas por coma, si los valores son de tipo numérico no necesitan ir entre comillas.

```
-- Ejemplos Operador IN

-- Ejemplo 18
-- Corredores de la categoría 7, 8 y 5
SELECT * FROM corredores WHERE categoria_id = 7 OR categoria_id = 8 OR
categoria_id = 5;

-- Ejemplo 19
-- Mismo ejercicio anterior pero usando el operador IN
SELECT * FROM corredores WHERE categoria_id IN (7, 8, 5);
```

6.6.4.3 BETWEEN

Nos permite comprobar si una expresión numérica se encuentra dentro de un rango.

Sintaxis

```
Expr [NOT] BETWEEN ValorInicial ADN ValorFinal
```

Expresión normalmente es el nombre de una columna y Valor_inicial y valor_final fijan el rango

```
-- Ejemplos Operador BETWEEN

-- Ejemplo 20
-- Corredores entre 30 y 40 años ambos inclusives
SELECT * FROM corredores WHERE Edad>= 30 AND Edad<=40;

-- Ejemplo 21
-- Mismo ejemplo anterior pero con el opeador BETWEEN
SELECT * FROM corredores WHERE Edad BETWEEN 30 AND 40;

-- Ejemplo 22
-- Corredores que no tengan entre 30 y 40 años
SELECT * FROM corredores WHERE Edad NOT BETWEEN 30 AND 40;
```

6.6.4.4 LIKE

Permite comparar una expresión con un patrón de caracteres

```
Expr [NOT] LIKE 'PatrónCaracteres'
```

Para declarar el Patrón se pueden usar dos caracteres comodines:

- '%' – sustituye a un conjunto de caracteres
- '_' – sustituye a un solo carácter

```
-- Ejemplos Operador LIKE

-- Ejemplo 23
-- Seleccionar aquellos corredores cuyo nombre empiece por M
SELECT * FROM corredores WHERE nombre LIKE 'M%';

-- Ejemplo 24
-- seleccionar aquellos corredores que se apelliden Pérez
SELECT * FROM corredores WHERE apellidos LIKE '%Pérez%';

-- Ejemplo 25
-- Seleccionar aquellos corredores que sean de Ubrique y
-- su primer apellido empiece por R
SELECT * FROM corredores WHERE Ciudad='Ubrique' and apellidos LIKE 'R%';

-- Ejemplo 26
-- Seleccionar aquellos corredores cuya 3 letra del nombre es una a
SELECT * FROM corredores WHERE nombre LIKE '__a%';
```

6.6.4.5 REGEXP y Expresiones Regulares

El operador REGEXP (equivalente a RLIKE) tiene la misma funcionalidad que el LIKE aunque es mucho más potente ya que se base en el uso de expresiones regulares.

Una **expresión regular** es una forma muy potente de especificar un patrón de caracteres para una búsqueda compleja. Está formada por caracteres literales y otros con una función especial en el mismo sentido que "_" y "%" para el LIKE.

La expresión regular más sencilla es aquella que no contiene caracteres especiales. Por ejemplo la expresión regular “hola” coincide con “hola” y nada más.

Las expresiones regulares no triviales usan ciertas construcciones especiales de modo que pueden coincidir con más de una cadena. Por ejemplo, la expresión “Hola|mundo” coincide tanto con la cadena “Hola” como con la cadena “mundo”.

Como ejemplo algo más complejo la expresión regular “B[an]*s” coincide con cualquiera de las cadenas siguientes: “Bananas“, “Baaaaaas“, “Bs“, y cualquier otra que empiece con “B“, termina con “s“ y contenga cualquier número de caracteres “a“ o “n“ entre la “B“ y la “s“.

En la siguiente tabla se muestra los caracteres especiales más básicos que se pueden usar con el operador REGEXP.

Tabla 6. Tabla operadores REGEXP

Operador	Función	Ejemplo	Descripción
^	Coincidencia con el principio de una cadena	REGEXP ‘^A’;	Cadenas que comiencen por A
$	Coincidencia del final de una cadena	REGEXP ‘a$’;	Cadenas que finalicen por a.
|	Coincidencia de una entre varias coincidencias posibles	REGEXP '^P|J'	Cadenas que comiencen por P o J.
.	Coincidencia con el contenido de la cadena que le precede	REGEXP 'an.'	Cadenas que contengan an
*	Coincidencia de cero o más caracteres iguales que el precedente	REGEXP 'ua*'	Cadenas que contenga una u y a continuación una a
+	Coincidencia de una o más veces	REGEXP 'jo+';	Cadenas que contengan la secuencia jo
(secuencia)	Para indicar una secuencia de varios caracteres	REGEXP '(ua|fa)'	Cadenas que contengan la secuenciaua o fa
{n}, {n,m}	Permite indicar un número o rango de ocurrencias	REGEXP 'j{2,3}'	Cadena que contenga la secuencia jj o jjj
[a-z]	Coincidencia con cualquier carácter del conjunto	REGEXP '[dv]'	Cadenas que contengan una d o v

Veamos algunos ejemplos

```
-- Ejemplos REGEXP

-- Ejemplo 27
-- Selecciona aquellos corredores cuyo nombre comience por A.
SELECT * FROM Corredores WHERE Nombre REGEXP '^A';

-- Ejemplo 28
-- Selecciona aquellos corredores cuyo nombre finalice por a.
SELECT * FROM Corredores WHERE Nombre REGEXP 'a$';

-- Ejemplo 29
-- Selecciona aquellos corredores cuyo nombre comience por P o J.
SELECT * FROM Corredores WHERE Nombre REGEXP '^P|J';

-- Ejemplo 30
-- Selecciona aquellos corredores cuyo nombre contenga la cadena an.
SELECT * FROM Corredores WHERE Nombre REGEXP 'an.';


-- Ejemplo 31
-- Selecciona aquellos corredores cuyo nombre contenga una u y una a
continuación o no
SELECT * FROM Corredores WHERE Nombre REGEXP 'ua*';

-- Ejemplo 32
-- Selecciona aquellos corredores cuyo nombre contenga la cadena jo
SELECT * FROM Corredores WHERE Nombre REGEXP 'jo+';

-- Ejemplo 33
-- Selecciona aquellos corredores cuyo nombre contenga la cadena ua o fa
SELECT * FROM Corredores WHERE Nombre REGEXP '(ua|fa)';

-- Ejemplo 34
-- Selecciona aquellos corredores cuyo nombre comience por Ju o Ev
SELECT * FROM Corredores WHERE Nombre REGEXP '^(Ju|Ev)';

-- Ejemplo 35
-- Selecciona aquellos corredores cuyo nombre contenga la secuencia j
-- de dos a tres veces seguidas
SELECT * FROM Corredores WHERE Nombre REGEXP 'j{2,3}';

-- Ejemplo 36
-- Selecciona aquellos corredores cuyo nombre contenga d o una v
-- de dos a tres veces
SELECT * FROM Corredor WHERE Nombre  REGEXP '[dv]';

-- Ejemplo 38
-- Selecciona aquellos corredores cuyo nombre no contega una a o una p
SELECT * FROM Corredores WHERE Nombre  NOT REGEXP '[ap]';
```

Cláusula ORDER BY

Permite especificar un criterio de clasificación para el resultado de la consulta

Sintaxis

```
ORDER BY {Columna| Expresión | Posición} [ASC|DESC]
```

- Columna. Especificaremos la columna a partir de la cual se establezca el criterio de ordenación. Podemos establecer criterios de ordenación anidados, es decir usando varias columnas.
- Expresión. Podemos usar una expresión como criterio de clasificación.
- Posición. Especificaremos la posición de la columna a la derecha del SELECT a partir de la cual se establecerá la ordenación
- ASC|DESC. Criterio ordenación ascendente o descente. Si se omite usa ASC.

```
-- Ejemplos cláusula ORDER BY

-- Ejemplo 39
-- Corredores ordenados por apellidos
SELECT * FROM corredores ORDER BY apellidos;

-- Ejemplo 40
-- Corredores ordenados por apellidos y luego por nombre
-- Mostrar id, Apellidos, Nombre, Ciudad y Edad
SELECT
       id, Apellidos, Nombre, Ciudad, Edad
FROM
       corredores
ORDER BY apellidos,nombre;

-- Ejemplo 41
-- Mostrar corredores desde los más veteranos a los más jóvenes
SELECT
       Id, Apellidos, Nombre, Ciudad, Edad
FROM
       corredores
ORDER BY Edad DESC;
```

6.6.5 Cláusula LIMIT

Se usa para restringir el número de registros devueltos por el SELECT

Sintaxis

```
[LIMIT {[offset,] row_count | row_count OFFSET offset}]
```

- Offset. Indica el desplazamiento del primer registro a retornar
- Row_count. El número de registros a retornar.Si no se ha especificado Offset devuelve los Row_count primeros.
- row_count OFFSET offset. Se suele usar para mantener la compatibilidad con PostgreeSQL

```
-- Ejemplos cláusula LIMIT

-- Ejemplo 42
-- Corredores ordenados por apellidos 3 primeros
SELECT * FROM corredores ORDER BY apellidos LIMIT 3;

-- Ejemplo 43
-- Corredores ordenados por apellidos y luego por nombre
-- Mostrar id, Apellidos, Nombre, Ciudad y Edad
-- A partir de los 2 primeros mostrar 5
SELECT
       id, Apellidos, Nombre, Ciudad, Edad
FROM
       corredores
ORDER BY apellidos,nombre
LIMIT 2,5;

-- EJEMPLO 44
-- Mismo resultado anterior con OFFSET
SELECT
       id, Apellidos, Nombre, Ciudad, Edad
FROM
       corredores
ORDER BY apellidos,nombre
LIMIT 5 OFFSET 2;

-- Ejemplo 45
-- Mostrar los 5 corredores más veteranos
SELECT
       id, Apellidos, Nombre, Ciudad, Edad
FROM
       corredores
ORDER BY Edad DESC
LIMIT 5;
```

6.6.6 Modificadores ALL y DISTINCT

- ALL - Recuperamos todas las filas aunque algunas estén repetidas. Es la opción por omisión.
- DISTINCT –No recupera las filas repetidas.

```
-- Ejemplos con ALL o DISTINCT

-- Ejemplo 46
-- Muestra las poblaciones de los distintos corredores
SELECT ALL ciudad FROM corredores;

-- Ejemplo 47
-- Mismo ejemplo anterior omitiendo ALL
SELECT ciudad FROM corredores;

-- Ejemplo 48
-- Muestra las poblaciones de los corredores, no mostrar las repetidas
SELECT DISTINCT ciudad FROM corredores;
```

6.6.7 Expresiones

En una sentencia para formular una consulta se pueden realizar operaciones con los datos. Por ejemplo, se puede solicitar el resultado del producto de los valores de dos columnas, o el valor de una columna dividido por valor numérico literal. Para ello se utilizan expresiones en las que se combinan valores literales o de campos con operadores de distinto tipo.

También como hemos visto en la cláusula WHERE pueden utilizarse expresiones en las condiciones de búsqueda.

Una **expresión** es una **combinación de operadores, operandos y paréntesis**. El resultado de la ejecución de una expresión es un único valor. En el caso de que la expresión sea de tipo condición, sólo devolverá dos valores posibles cero o uno.

En el formato de la sentencia SELECT descrito anteriormente, las expresiones pueden utilizarse en la cláusula **SELECT** en lugar de nombres de columnas y en la clausula **WHERE** en la formulación de una condición.

Los operandos pueden ser nombres de columnas, constantes u otras expresiones. Más adelante veremos otros tipos de operandos, como las funciones de columna.

Los operadores actúan sobre datos homogéneos, es decir, bien numéricos o alfanuméricos.

Los tipos de operadores son:

- Aritméticos
- De comparación
- Lógicos
- De asignación
- De bits
- De cadenas
- Y de control de flujo

A continuación presentamos una tabla resumen de los mismos:

Tabla 7. Operadores MySQL

Aritméticos	+	Suma
	-	Resta
	*	Multiplicación
	/	División
	** ^	Exponenciación
Relacionales o de Comparación	<	Menor
	<=	Menor o igual
	>	Mayor

	>=	Mayor o igual
	<> !=	Distinto
	!<	No menor que
	!>	No mayor que
Lógicos	AND	Los operadores lógicos permiten comparar expresiones lógicas devolviendo siempre un valor verdadero o falso. Los operadores lógicos se evalúan de izquierda a derecha
	NOT	
	OR	
	XOR	

Veamos algunos ejemplos:

```
-- Ejemplo 49
-- Devuelve la fecha actual del sistema
SELECT CURRENT_DATE AS 'FECHA ACTUAL';

-- Ejemplo 50
-- Devuelve la fecha y la hora actual del sistema
SELECT CURRENT_TIMESTAMP AS 'FECHA HORA ACTUAL';

-- Ejemplo 51
-- Devuelve la hora actual
SELECT CURRENT_TIME AS 'HORA ACTUAL';
```

6.6.8 Funciones de Agregado

Estas funciones (también llamadas de columnas o colectivas) permiten obtener un solo valor como resultado de aplicar una determinada operación sobre los valores contenidos en una columna.

Son aquellas funciones cuyo argumento es una colección de valores tomados de los pertenecientes a una o más columnas de una tabla. Se llaman también por ello funciones de columna.

Las funciones de agregación son las siguientes:

- **AVG**: devuelve la media de los valores de una colección
- **MAX**: devuelve el valor máximo
- **MIN**: devuelve el valor mínimo
- **SUM**: devuelve la suma
- **COUNT**: devuelve el número de elementos que tiene la colección

Sintaxis

```
nom_Funcion ([DISTINCT] nom_columna)
```

En general, las funciones de agregación se aplican a una columna, excepto la función de agregación COUNT, que normalmente se aplica a todas las columnas de la tabla o tablas seleccionadas. Por lo tanto, COUNT (*) contará todas las filas de la tabla o las tablas que cumplan las condiciones. Si se utilizase COUNT(DISTINCT columna), sólo contaría los valores que no fuesen nulos ni repetidos, y si se utilizase COUNT (columna), sólo contaría los valores que no fuesen nulos.

En una sentencia SELECT no puede especificarse DISTINCT más de una vez, ya sea dentro de una función o detrás del la cláusula SELECT.

```
-- Ejemplos Funciones de Agregado

-- Ejemplo 52
-- Número de corredores de Ubrique
SELECT COUNT(*) FROM Corredores WHERE Ciudad='Ubrique';

-- Ejemplo 53
-- Edad media de todos los corredores
SELECT AVG(edad) FROM Corredores;

-- Ejemplo 53
-- Edad máxima de los corredores de Villamartín
SELECT MAX(edad) FROM Corredores WHERE Ciudad='Villamartín';

-- Ejemplo 54
-- Edad mínima de los corredores de Arcos
SELECT MIN(edad) FROM Corredores WHERE Ciudad='Arcos';

-- Ejemplo 55
-- Suma de todos los socios de los club de Ubrique
SELECT SUM(NumSocios) FROM Club WHERE Ciudad='Ubrique';

-- Ejemplo 56
-- Edad Máxima, Mínima y Media de los Corredores de Ubrique
SELECT
       'Edades Club Ubrique',
       MAX(Edad) AS EdadMAX,
       MIN(Edad) AS EdadMin,
       AVG(Edad) AS EdadMedia
FROM
       Corredor
WHERE
    Ciudad = 'Ubrique';
```

Veamos los siguientes ejemplos

6.7 Subconsultas. SELECT anidados

Una subconsulta es una consulta incluida dentro de una cláusula WHEREo HAVING de otra consulta.

En ocasiones, para expresar ciertas condiciones no hay más remedio que obtener el valor que buscamos como resultado de una consulta. Por ejemplo, si queremos saber que Corredores son mayores que Eva Rubiales, previamente debiéramos de saber cuál es la edad de Eva

Rubiales, lo que requiere llevar a cabo una consulta adicional. Para evitar hacer dos consultas en línea se usan las consultas subordinadas o subconsultas.

Una sentencia subordinada de otra puede tener a su vez otras sentencias subordinadas a ella. Llamamos sentencia externa a la primera de todas, la que no es subordinada de ninguna. Una sentencia es antecedente de otra cuando esta es su subordinada directa o subordinada de sus subordinadas a cualquier nivel.

A la sentencias subordinadas suele llamárseles anidadas. Puede haber varios niveles de anidamiento según el SGBD.

Qué aportan las subconsultas:

- Permiten consultas estructuradas de forma que es posible aislar cada parte de un comando.
- Proporcionan un modo alternativo de realizar operaciones que de otro modo necesitarían joins y uniones complejos.
- La innovación de las subconsultas lo que dio la idea original de llamar a SQL "Structured Query Language."

Sintaxis

```
SELECT Lista_Columnas FROM Lista_Tablas WHERE Columna Operador_Comparacion
(SELECT … FROM … WHERE ..)
```

Una subconsulta debe ir siempre entre paréntesis.

Ejemplos

```
-- Ejemplos Subconsultas

-- Ejemplo 57
-- Determinar los corredores que son mayores que Eva Rubiales
SELECT
    *
FROM
       Corredores
WHERE
       Edad> (SELECT Edad From Corredores
              WHERE
              Nombre = 'Eva' And Apellidos Like 'Rubiales%');
```

En las subconsultas también podemos usar las funciones de agregado, veamos el siguiente

```
-- Ejemplos Subconsultas

-- Ejemplo 58
-- Determinar el corredor más veterano
SELECT
       *
FROM
       Corredores
WHERE
       Edad = (SELECT max(Edad)From Corredor);
```

ejemplo

Las subconsultas pueden ser parte de los siguientes predicados:

- Predicados básicos de comparación
- Predicados cuantificados (ANY, SOME, ALL)
- Predicado EXISTS
- Predicado IN

6.7.1 Predicados Básicos de Comparación

Solo se podrán usar cuando la subconsulta devuelve un valor único, es decir, una tabla con una sola fila y una sola columna.

El conjunto de operadores que lo conforman son los ya conocidos operadores de comparación:

- >, <, <>, <=, >=, =

De este tipo son los ejemplos vistos en el apartado anterior, donde la consulta sólo devuelve un valor que puede ser comparado con el valor de una columna o literal.

6.7.2 Predicados Cuantificadores (ANY, SOME, ALL)

A veces se permite que el resultado de la sentencia SELECT subordinada tenga más de un valor si ésta viene precedida de una de las palabras ALL, SOME, ANY (palabras cuantificadoras). Cuando se usan estas palabras o cláusulas los predicados en los que participan se denomina predicados cuantificados.

El resultado de la sentencia SELECT subordinada debe ser una tabla con una sola columna y cero o más filas.

6.7.2.1 Cuantificador ALL

El predicado cualificado es verdadero si la comparación es verdadera para todos y cada uno de los valores devueltos por el SELECT subordinado, también si devuelve una tabla vacía.

Así los valores que devuelve:

- FALSO. Si para alguno de los valores no nulos la comparación toma el valor FALSO.
- DESCONOCIDO. Cuando la comparación es verdadera para todos los valores no nulos.
- VERDADERO. Si la comparación es verdadera para todos los valores devueltos por el SELECT subordinado.

```
-- Ejemplos Subconsultas ALL
-- Ejemplo 59
-- Determinar los corredores que son mayores que todos los de Villamartín
SELECT
    *
FROM
       Corredores
WHERE
       Edad> ALL (SELECT
                              Edad
                      From
                              Corredores
                      WHERE
                              Ciudad = 'Villamartín');
```

6.7.2.2 Cuantificador ANY o SOME

El predicado cuantificado es verdadero si la comparación es verdadera para uno cualquiera de los valores devueltos por la ejecución de la sentencia SELECT subordinado.

Si la sentencia subordinada devuelve una tabla vacía, el predicado cuantificado toma el valor FALSO.

Si devuelve una o más filas y alguna de ellas es nula, el predicado cuantificado puede ser:

- VERDADERO. Si para alguno de los valores no nulos el resultado de la comparación es VERDADERO.
- DESCONOCIDO. Si para todos los valores no nulos de la tabla el resultado de la comparación es FALSO.

Si devuelve una o más filas y ninguna es nula, el predicado cuantificado es verdadero si la comparación es verdadera para alguno de los valores. En cualquier otro caso es FALSO.

```
-- Ejemplos Subconsultas ANY o SOME
-- Ejemplo 60
-- Determinar los corredores cuya edad sea inferior a
-- la edad de algún corredor de Arcos
SELECT
    *
FROM
    Corredores
WHERE Edad< ANY
   (SELECT Edad From Corredores WHERE Ciudad = 'Arcos');
```

6.7.3 Predicado IN.

Cuando la subconsulta devuelve un conjunto de valores podemos usar la cláusula IN para comprobar que el valor de la comparación se encuentra entre la lista de valores devueltos

Los posibles valores resultantes son:

- VERDADERO. Cuando hay coincidencia con uno o más valores devueltos por la subconsulta.
- FALSO. No hay coincidencia con ninguno de los valores.
- DESCONOCIDO. Cuando la subconsulta no devuelve ningún valor.

```
-- Ejemplos Subconsultas IN

-- Ejemplo 61
-- Determinar los corredores cuya edad coincida
-- con la edad de cualquiera de los corredores de Arcos
SELECT
    *
FROM
       Corredores
WHERE
       Edad IN (SELECT
                           Edad
                    FROM
                           Corredores
                    WHERE
                           Ciudad = 'Arcos');
```

6.7.4 Predicado [NOT] EXISTS

Devuelve VERDADERO si la subconsulta es o no vacía y FALSO en caso contrario.

```
-- Ejemplos Subconsultas EXISTS

-- Ejemplo 62
-- Determinar los clubs que no tengan corredores
SELECT
    *
FROM
       clubs
WHERE
    NOT EXISTS (SELECT
                           *
                    FROM
                           Corredores
                    WHERE
                           clubs.id=Corredor.club_id);
```

6.8 Consultas Multitablas. SELECT … JOIN

En las consultas vistas hasta ahora y con la excepción de las subconsultas sólo hemos necesitado una sola tabla ya que era todo lo necesario, tanto los campos a mostrar como las condiciones impuestas estaban en la tabla. Sin embargo, en la mayoría de casos necesitaríamos campos de otras tablas bien para obtener más información, como el nombre del club al que pertenece un corredor, o bien porque las condiciones o filtros afecten a campos de otras tablas, por ejemplo si necesitásemos los corredores del equipo Nutrias Pantaneras.

Para estos casos necesitaremos incluir en la cláusula FROM las tablas requeridas por la consulta en un proceso conocido como producto cartesiano o combinación de tabla.

El **Producto Cartesiano** entre dos tablas, tiene como resultado una nueva tabla resultado de la combinación de cada fila de la primera tabla con cada fila de la segunda, de modo que la cardinalidad de nueva tabla se corresponde con multiplicar las filas de las dos tablas iniciales.

Por ejemplo, la siguiente consulta:

```
SELECT * FROM Corredor, Club
```

Produce una nueva tabla con 16*5 registros.

Si lo que pretendemos es que se muestren los datos del corredor junto con los datos de su club, no todas las filas son válidas, ya que estamos combinando cada jugador con todos los equipos. A las filas sobrantes del producto cartesiano se les llama **filas espurias**.

Para conseguir nuestro objetivo debe incluirse un filtro en la cláusula WHERE que deje solamente las filas válidas. En nuestro ejemplo son aquellas en que el campo común (o clave ajena) CodClub son iguales.

```
SELECT * FROM Corredores Co, Clubs Cl WHERE Co.club_id = Cl.id
```

Así junto con los datos de cada jugador se mostrarían también los de su equipo.

Para eliminar posible ambigüedad derivada del hecho de que puede haber campos con el mismo nombre es conveniente usar siempre prefijos para las tablas.

El proceso de diseño de una consulta de varias tablas se resume en:

1. Analizar la consulta para ver las tablas necesarias para resolverla.
2. Incluir dichas tablas en el FROM
3. Filtrar las filas espurias usando campos comunes o claves ajenas.
4. Añadir los filtros o cláusulas necesarios como si trabajásemos con una única tabla.

6.8.1 Operaciones de Reunión (JOIN)

6.8.1.1 INNER JOIN. Composición Interna.

Se denomina composición interna puesto que el resultado es un subconjunto del producto cartesiano entre las tablas combinadas.

La clave está en que se establece lo que se denomina la condición del JOIN de forma que se elimina del producto cartesiano todas aquellas filas que no cumplen dicha condición.

La composición interna se puede hacer con la cláusula WHERE como vimos en el apartado anterior, pero normalmente se usa la cláusula específica INNER JOIN.

Sintaxis

```
SELECT Lista_Columnas FROM Tabla1 [INNER] JOIN Tabla2 {ON Tabla1.Columna =
Tabla2.Columna | USING(Columna)}
```

- La cláusula ON permite especificar la condición del JOIN o de la composición.
- La cláusula USIGN(Columna) se utiliza cuando el nombre de la columnas que se incluyen en la condición son iguales. Normalmente ocurre así con respecto a las claves ajenas y primarias relacionadas.

6.8.1.2 Ejemplo Ilustrado INNER JOIN.

El **INNER JOIN** es uno de los comandos más importante de SQL ya que permite mostrar en una misma consulta columnas o datos procedente de más de 1 tabla. Para poder realizar este tipo de consultas es preciso que entre las tablas participantes exista una **relación** que puede ser bien de tipo 1:1, de tipo 1:N (la más habitual) o de tipo 1:N. Eso quiere decir que una tabla participante **A** tendrá una clave ajena que estará relacinada con la clave principal de la otra tabla participante **B**.

Ilustramos el INNER JOIN con un ejemplo, partiendo de dos tablas. La tabla "Empleados" contiene todos los empleados de una empresa, junto a clave principal **id** y la clave ajena **departamento_id** que indica el departamento al que pertenece el empleado.

Tabla: empleados

id	Apellidos	Nombre	Departamento_id
1	García Hurtado	Macarena	3
2	Ocaña Martínez	Francisco	1
3	Gutiérrez Doblado	Elena	1
4	Hernández Soria	Manuela	2
5	Oliva Cansino	Andrea	NULL

Esta tabla muestra dos particularidades: los empleados Ocaña Martínez y Gutiérrez Doblado trabajan en el mismo departamento. A la empleada Oliva Cansino todavía no se le ha asignado un departamento (NULL).

La tabla "**Departamentos**" enumera todos los departamentos de la empresa, incluyendo el número identificador de cada departamento y su ubicación.

id	Denominación	Localización
1	Ventas	Sevilla
2	IT	Málaga
3	Recursos Humanos	Marbella
4	Investigación	Málaga

Ambas tablas están enlazadas por una relación de tipo 1:N. La tabla **empleados** tiene la columna **departamento_id** a la cual se le ha aplicado la restricción **FOREIG KEY** restricción que relaciona **departamento_id** de **empleados** con el **id** de la tabla **departamentos**.

Esta conexión es la que permite generar un INNER JOIN con ambas tablas, que puede ayudar a determinar, por ejemplo, la localización del puesto de trabajo de un empleado, o cuál es el nombre del departamento al que pertenece un determinado empleado.

Así cuando se realiza una consulta sobre dos o más tablas estás deben estar relacionadas, se suele definir como condición de unión la correspondencia entre una clave ajena y otra principal. La condición se considera cumplida si la clave ajena seleccionada de una tabla coincide con la clave principal de la otra tabla (=), es decir, solo se emiten aquellos registros de datos que contienen valores comunes en ambas columnas.

Así la sentencia SQL que permitiría hacer una consulta multitabla entre empleados y departamentos sería la siguiente:

```
SELECT * FROM empleados
INNER JOIN departamentos ON empleados.departamento_id = departamentos.id;
```

En caso de querer obtener el siguiente resultado, es decir, los empleados junto con los datos del departamento al que pertenecen:

id	Apellidos	Nombre	Departamento_id	Denominación	Localización
1	García Hurtado	Macarena	3	Recursos Humanos	Marbella
2	Ocaña Martínez	Francisco	1	Ventas	Sevilla
3	Gutiérrez Doblado	Elena	1	Ventas	Sevilla
4	Hernández Soria	Manuela	2	IT	Málaga

La instrucción SELECT necesaria sería la siguiente:

```
SELECT empleados.id,
       empleados.apellidos,
       empleados.departamento_id,
       departamentos.denominacion,
       departamentos.localizacion
FROM   empleados
       INNER JOIN departamentos ON empleados.departamento_id = departamentos.id;
```

6.8.1.3 Ejemplos INNER JOIN sobre maratoon.

```
-- Ejemplos COMPOSICIONES INTERNAS INNER JOIN
```

```
-- Ejemplo 63
-- Mostrar junto con los datos de los corredores los datos del club
-- alquepertenece
SELECT
    *
FROM
    Corredores Co
        INNER JOIN
    clubs cl ON Co.club_id = Cl.id;

-- Ejemplo 64
-- Mismo ejemplo anterior sin la cláusula INNER
SELECT
       *
FROM
       Corredor Co
        JOIN
       club cl ON Co.club_id = Cl.id;

-- Ejemplo 66
-- Mostrar IdCorredor, Apellidos, Nombre, Edad, Nombre del Club y Nombre
-- de la categoría a la que pertenece cada corredor.
SELECT
       co.id,
       co.Apellidos,
       co.Nombre,
       co.Edad,
       cl.nombre AS Club,
       ca.nombre AS Categoría
FROM
       Corredores co
        INNER JOIN
       Clubs cl ON Co.club_id = Cl.id
         INNER JOIN
      Categorias ca ON co.categoria_id = ca.id
ORDER BY co.apellidos ,co.nombre;
-- Como vemos se trata de un DOBLE JOIN donde primero conecto Corredor con
-- Club y a su vez el resultado lo conecto con Categoría. Como condición de los
-- JOINS uso las correspondientes claves ajenas.
```

6.8.1.4 *NATURAL JOIN. Composición Interna Natural.*

La cláusula NATURAL JOIN requiere que entre las tablas sobre las que se aplica el JOIN exista una sola columna con mismo con el mismo nombre y dominio. Normalmente esta columna coincidente se corresponde con la clave ajena de una y la clave principal de otra. Si es así, ella misma establece la condición del JOIN igualando el valor de ambas columnas.

El NATURAL JOIN facilita la realización de una composición interna, ya que el código del script SQL lo reduce a la mínima expresión.

Sintaxis

```
SELECT Lista_Columnas FROM Tabla1 NATURAL JOIN Tabla2;
```

- No necesita condición de JOIN, ella misma lo establece a partir de las columnas comunes
- Por defecto, una de las columnas comunes no la muestra

El NATURAL JOIN no es compatible con algunas versiones de SQL, por lo que no se suele utilizar.

En el ejemplo de la tabla empleado y departamentos sólo se podría usar si la clave principal de departamento en vez de ser **id** se llamase **departamento_id**.

Tabla empleados

id	Apellidos	Nombre	Departamento_id
1	García Hurtado	Macarena	3
2	Ocaña Martínez	Francisco	1
3	Gutiérrez Doblado	Elena	1
4	Hernández Soria	Manuela	2
5	Oliva Cansino	Andrea	NULL

Tabla departamentos

Departamento_id	Denominación	Localización
1	Ventas	Sevilla
2	IT	Málaga
3	Recursos Humanos	Marbella
4	Investigación	Málaga

Ahora la consulta SELECT podríamos realizarla con un NATURAL JOIN, ya que la clave ajena y la clave principal tienen el mismo nombre.

```
SELECT empleados.id,
       empleados.apellidos,
       empleados.departamento_id,
       departamentos.denominacion,
       departamentos.localizacion
FROM
       empleados NATURAL JOIN departamentos;
```

6.8.1.5 LEFT JOIN. Composiciones Externas.

El resultado no es un subconjunto del producto cartesiano entre las dos tablas, por ello se denominan composiciones externas.

Se combinan todas las filas de la primera tabla en la cláusula FROM con cada una de la segunda tabla que cumpla la condición del JOIN expresada con ON. Este tipo de consultas no permite el uso del WHERE para comparar los campos comunes.

```
SELECT Lista_columnas FROM Tabla1 LEFT JOIN Tabla2 {ON CondicionJoin |
USING(Columna)}
```

En caso de que los campos comunes tengan el mismo nombre y no existan más columnas coincidentes podemos usar el **NATURAL LEFT JOIN**

```
SELECT Lista_columnas FROM Tabla1 NATURAL LEFT JOIN Tabla2
```

Veamos el siguiente ejemplo sobre las tablas **Empleados** y **Departamentos**.

Si ejecuto este comando

```
SELECT empleados.id,
       empleados.apellidos,
       empleados.departamento_id,
       departamentos.denominacion,
       departamentos.localizacion
FROM   empleados
       INNER JOIN departamentos ON empleados.departamento_id = departamentos.id;
```

El resultado obtenido com ya sabemos el siguiente

id	Apellidos	Nombre	Departamento_id	Denominación	Localización
1	García Hurtado	Macarena	3	Recursos Humanos	Marbella
2	Ocaña Martínez	Francisco	1	Ventas	Sevilla
3	Gutiérrez Doblado	Elena	1	Ventas	Sevilla
4	Hernández Soria	Manuela	2	IT	Málaga

Si observo bien la tabla resultante el empleado **5** no se muestra. Eso es debido a que este empleado no tiene asignado aún ningún departamento y por lo tanto no cumple la condición del INNER JOIN. En caso de querer mostrar en el informe también aquellos empleados que no tienen asignado ningún departamento, estamos obligados a usar la cláusula **LEFT JOIN**.

```
SELECT empleados.id,
       empleados.apellidos,
```

```
        empleados.departamento_id,
        departamentos.denominacion,
        departamentos.localizacion
FROM    empleados
        LEFT JOIN departamentos ON empleados.departamento_id = departamentos.id;
```

El resultado entonces sería

id	Apellidos	Nombre	Departamento_id	Denominación	Localización
1	García Hurtado	Macarena	3	Recursos Humanos	Marbella
2	Ocaña Martínez	Francisco	1	Ventas	Sevilla
3	Gutiérrez Doblado	Elena	1	Ventas	Sevilla
4	Hernández Soria	Manuela	2	IT	Málaga
5	Oliva Cansino	Andrea	NULL	NULL	NULL

Si observamos las columnas correspondientes al departamento se muestra el valor NULL, lo cual es lógico puesto que este empleado aún no tiene asignado ningún departamento.

6.8.1.6 RIGHT JOIN. Composición Externa

Mismo caso anterior salvo que considera primero la tabla de la derecha.

Se combinan todas las filas de la segunda tabla en la cláusula FROM con cada una de la primera tabla que cumpla la condición del JOIN expresada con ON. Este tipo de consultas no permite el uso del WHERE para comparar los campos comunes.

Sintaxis

```
SELECT Lista_columnas FROM Tabla1 RIGHT JOIN Tabla2 {ON CondicionJoin |
USING(Columna)
```

En caso de que los campos comunes tengan el mismo nombre y no existan más columnas coincidentes podemos usar el **NATURAL RIGHT JOIN**

```
SELECT Lista_columnas FROM Tabla1 NATURAL LEFT JOIN Tabla2
```

Veamos el siguiente ejemplo basado en las tablas **empleados** y **departamentos**.

Si ejecuto el comando

```
SELECT empleados.id,
       empleados.apellidos,
       empleados.departamento_id,
       departamentos.denominacion,
       departamentos.localizacion
FROM   empleados
       INNER JOIN departamentos ON empleados.departamento_id = departamentos.id;
```

Entonces el resultado es el que ya sabemos

id	Apellidos	Nombre	Departamento_id	Denominación	Localización
1	García Hurtado	Macarena	3	Recursos Humanos	Marbella
2	Ocaña Martínez	Francisco	1	Ventas	Sevilla
3	Gutiérrez Doblado	Elena	1	Ventas	Sevilla
4	Hernández Soria	Manuela	2	IT	Málaga

Si nos fijamos el departamento **4** no se muestra en el informe, la razón es porque no existe ningún empleado en el departamento **4** por lo tanto no cumple la condición del INNER JOIN. Si en el informe se desean mostrar los datos de todos los departaemtos aún sin tener empleados asignados tendríamos que usar la cláusula **RIGHT JOIN**.

Así con el siguiente comando

```
SELECT empleados.id,
       empleados.apellidos,
       empleados.departamento_id,
       departamentos.denominacion,
       departamentos.localizacion
FROM   empleados
       RIGHT JOIN departamentos ON empleados.departamento_id = departamentos.id;
```

Obtendríamos el siguiente resultado

id	Apellidos	Nombre	Departamento_id	Denominación	Localización
1	García Hurtado	Macarena	3	Recursos Humanos	Marbella
2	Ocaña Martínez	Francisco	1	Ventas	Sevilla
3	Gutiérrez Doblado	Elena	1	Ventas	Sevilla
4	Hernández Soria	Manuela	2	IT	Málaga
null	null	null	null	Investigación	Málaga

6.8.2 Operaciones Básicas Álgebra Relacional

SQL soporta las tres operaciones del álgebra relacional sobre dos conjuntos de resultados o relaciones. Para ello ambas relaciones deben tener mismo número de campos y dominios compatibles.

Sin embargo, aunque gestores como ORACLE y SQL Server soportan todos ellos MySQL sólo soporta la unión. No obstante se puede lograr la intersección y diferencia indirectamente aplicando métodos de consultas avanzadas.

6.8.2.1 Unión

En ocasiones podemos requerir que nuestros datos estén divididos en varias tablas. Por ejemplo, una empresa que vende productos deportivos podría tener en su base de datos una sola tabla para cada categoría de sus productos (atletismo, baloncesto, etc...). Aunque cada categoría tiene sus propios atributos, es normal que compartan algunos de ellos como el Identificador, nombre, precio, etc.. De este modo si queremos mostrar los datos comunes de todos ellos una unión sería el tipo de consulta más apropiada.

Para efectuar una unión todas las consultas involucradas deben tener el mismo número y tipo de campos.

Los nombres de columna usados por el primer SELECT se usan como nombres de columna para los resultados retornados.

Sintaxis UNION

```
SELECT ...
UNION [ALL | DISTINCT] SELECT ...
[UNION [ALL | DISTINCT] SELECT ...]
```

Salvo que usemos la cláusula ALL todos los registros devueltos son únicos, como si hubiésemos hecho un DISTINCT para el conjunto de resultados total. Si especificamos ALL obtenemos todos los registros coincidentes de todos los SELECT usados.

```
-- Ejemplo 70
-- UNION
-- Ejemplo 71
-- Mostrar todos los corredores de Ubrique y Villamartín ordenados
alfabéticamente
-- usando el operador UNION.
(SELECT
       id, Apellidos, Nombre, Ciudad
FROM
       Corredores
WHERE
    Ciudad = 'Ubrique')UNION (SELECT
       id, Apellidos, Nombre, Ciudad
FROM
    Corredor
WHERE
    Ciudad = 'Villamartín')ORDER BY Apellidos,Nombre;
```

6.8.2.2 Intersección

Es una operación entre dos conjuntos cuyo resultado es el conjunto de elementos comunes a ambos.

MySQL no soporta este comando que en otros gestores sería INSTERSECT.

6.8.2.3 Diferencia

Es la operación entre dos conjuntos cuyo resultado es el conjunto de elementos que son distintos.

En otros sistemas gestores la cláusula SQL es EXCEPT(PostrgreSQL) o MINUS(SQL Server)

6.9 Consultas con Agrupación de Registros. GROUP BY

Hasta ahora hemos usado la cláusula SELECT para recuperar filas de una tabla y la cláusula WHERE para seleccionar el número de filas que se recuperan.

También hemos empleado las Funciones de Agregación para trabajar con conjuntos de filas, así podemos calcular el salario medio de los empleados de un determinado departamento o la edad media de los corredores de un determinado club.

A veces interesa consultar los datos según grupos determinados. Por ejemplo: saber la edad media de los corredores de cadaclub.Las cláusulas que conocemos nos permiten obtener la edad media de los corredores de un solo club, pero no de todos.Para ello necesitamos hacer un agrupamiento de registros por club, mediante GROUP BY y luego mostrar la columna Club y la función de agregación Edad Media.

La sintaxis de una consulta SELECT con la cláusula GROUP BY quedaría de la siguiente forma:

```
SELECT nombre_columnas_a seleccionar
FROM tabla_a_consultar
[WHERE condiciones]
GROUP BY columnas_según_las_cuales_se_quiere_agrupar
[HAVING condiciones_por_grupos]
[ORDER BY columna_ordenación [DESC] [, columna [DESC]...]];
```

6.9.1 Cláusula GROUP BY

La cláusula GROUP BY indica que se han de agrupar las filas de la tabla de modo que todas las que tengan iguales valores para las columnas de agrupamiento formen un grupo. Pueden existir grupos de una sola fila. Los valores nulos en la columna de agrupamiento se consideran valores iguales, por lo que se incluyen en el mismo grupo.

Una vez formados los grupos, para cada uno de ellos se evalúan las expresiones de la cláusula SELECT, por lo tanto cada uno de ellos producirá una única fila en la tabla resultante.

Las columnas que participan en las expresiones de la cláusula SELECT y no sean de agrupamiento sólo pueden especificarse en los argumentos de funciones de agregado

colectivas. Dicho de otro modo, si aparece una columna en un SELECT y no está incluida en una función de agregado debe ser una columna de agrupamiento, si no la consulta será errónea.

Si se especifica la cláusula WHERE primero selecciona los registros que cumplan la condición especificada, y a partir de dicho resultado se establece la agrupación de registros.

Dada la tabla corredor

Tabla Corredor

IdCorredor	Nombre	Apellidos	Poblacion	Fnacimiento	Sexo	Edad	Cat	Club
1	Juan	García Pérez	Ubrique	1965-07-31	H	50	7	1
2	Juan José	Pérez Morales	Ubrique	1945-08-30	H	70	8	1
3	Eva	Rubiales Alva	Ubrique	1980-08-25	M	35	5	1
4	Josefa	Rios Pérez	Villamartín	1990-10-15	M	25	4	2
5	Pedro	Ortega Ríos	Villamartín	1994-05-14	H	21	4	2
6	Francisco	Morales Almeida	Villamartín	1970-02-01	H	46	6	2
7	Macarena	Fernández Pérez	Villamartín	1980-05-03	M	35	5	2
8	Jesús	Romero Reguera	Villamartín	1970-06-05	H	45	6	2
9	Pedro	García Ramírez	Ubrique	1967-07-31	H	48	6	1
10	María	Pérez Moreno	Ubrique	1975-08-30	M	40	6	1
11	Almudena	Romero Alva	Arcos	1986-08-25	M	29	4	4
12	Francisco	Pérez Amor	Arcos	1992-10-15	H	23	4	4
13	Juan	Rodríguez Ríos	Ubrique	1978-05-14	H	37	5	1
14	Cristina	García Almeida	Villamartín	1978-02-01	M	38	5	2
15	Romira	Jiménez Pérez	Arcos	1984-05-03	M	31	5	4
16	José	Rincón Pérez	Villamartín	1960-06-05	H	55	7	2

Se pide un script SQL que muestre el código de club, Nombre del Club y la Edad Mínima, Máxima, Media y Nº de Miembros de Cada Club.

Para ello aplicamos al SELECT la cláusula GROUP BY siendo la columna de agrupación club_id de la tabla Corredor, además como necesito el nombre del club tengo que hacer un INNER JOIN entre Corredor y Club.

```
-- Mostrar el Código de Club, Nombre del Club, la Edad Media, Máxima, Mínima y -- N° de
Miembros de cada club
SELECT
    co.club_id,
    cl.Nombre,
    AVG(co.Edad) AS 'Edad Media',
    MAX(Edad) AS 'Edad Máxima',
    MIN(Edad) AS 'Edad Mínima',
    COUNT(*) AS 'Miembros'
FROM
    Corredores co
        INNER JOIN
    Clubs cl ON co.club_id = cl.id
GROUP BY club_id;
```

Así aplicando el ejemplo anterior el resultado obtenido sería

RESULTADO GROUP BY EEJEMPLO					
CodClub	**Nombre**	**Edad Media**	**Edad Máxima**	**Edad Mínima**	**Miembros**
1	Nutrias Pantaneras	53.4444	74	39	9
2	Club Atletismo Villamartín	41.8571	59	25	7
4	Asociación Deportiva de Arcos	31.6667	35	27	3
5	Club Atletismo Fronter	29.0000	40	18	3

Vemos otro ejemplo

```
-- Mostrar el Código de la Categoría, Nombre de la Categoría y el Número de --Corredores
-- de cada categoría
SELECT
    co.categoria_id AS #, ca.Nombre, COUNT(*) 'Miembros'
FROM
    Corredores co
        JOIN
    Categorias ca ON co.categoria_id = ca.id
GROUP BY co.categoria_id
ORDER BY co.categoria_id;
```

Siendo el resultado

#	Nombre	Miembros
4	Senior A	5
5	Senior B	4
6	Veteranos A	5
7	Veteranos B	7
8	Veteranos C	1

6.9.2 Cláusula HAVING

La cláusula **HAVING** sólo se puede usar junto con la cláusula **GROUP BY**, y permite establecer una condición sobre los grupos, de forma que sólo los grupos que dicha función se mostrarán en el resultado de la consulta. Para establecer la condición del HAVING puedo usar las columnas que participan en la lista de columnas o las funciones de agregado correspondiente.

```
-- EJEMPLO 73
-- Mostrar el Código de Club, Nombre del Club, la Edad Media
-- Sólo los club con edad media superior a 40 años
SELECT
       co.CodClub,
       cl.Nombre,
       AVG(co.Edad) AS EdadMedia
FROM
       Corredor co
        JOIN
       Club cl USING (CodClub)
GROUP BY CodClub
HAVING EdadMedia>40;
```

La cláusula HAVING puede utilizarse también sin un GROUP BY previo. En este caso se aplica la condición al único grupo de filas formado por todas las filas de la tabla resultante.

La cláusula DISTINCT solo puede especificarse una vez, bien en la cláusula SELECT o bien dentro de funciones colectivas en la condición de la cláusula HAVING. Este límite no rige para las sentencias subordinadas que pueda haber en los predicados de la cláusula WHERE o HAVING.

6.10 Definición y uso de las vistas.

Como hemos observado, la arquitectura ANSI/SPARC distingue tres niveles, quese describen en el esquema conceptual, el esquema interno y los esquemas externos.Hasta ahora, mientras creábamos las tablas de la base de datos, íbamosdescribiendo el esquema conceptual. Para describir los diferentes esquemas externosutilizamos el concepto de vista del SQL.

Las vistas no existen realmente como un conjunto de valores almacenadosen la base de datos, sino que son tablas ficticias, denominadas derivadas (nomaterializadas). Se construyen a partir de tablas reales (materializadas) almacenadasen la base de datos, y conocidas con el nombre de tablas básicas (otablas de base). La no-existencia real de las vistas hace que puedan ser actualizableso no.

Las vistas tienen la misma estructura que una tabla y se tratan de forma semejante, así se pueden usar incluso en una consulta SELECT. Sentencias SELECT complejas suelen almacenarse en forma de vistas, para que a partir de ella se puedan realizar nuevas consultas.

Comparten el mismo espacio de nombres en la base de datos, por eso, una base de datos no puede contener una tabla y una vista con el mismo nombre.

En caso de ser suprimida una tabla se consideran inválidas las VISTAS asociadas.

6.10.1 Crear vista. CREATE VIEW

Para crear una vista es necesario utilizar la sentencia CREATE VIEW y se crea a partir del resultado de una consulta SELECT.

Sintaxis

```
CREATE [OR REPLACE] VIEW nombre_vista [(lista_columnas)] AS sentencia_select
[WITH CHECK OPTION];
```

Lo primero que tenemos que hacer para crear una vista es decidir qué nombrele queremos poner (nombre_vista). Si queremos cambiar el nombre de lascolumnas, o bien poner nombre a alguna que en principio no tenía, lo podemoshacer en lista_columnas. Y ya sólo nos quedará definir la consulta queformará nuestra vista.

Observaciones:

- [OR REPLACE]. Crea una vista nueva o reemplaza a la existente.

- [(lista_columnas)]. Si no se especifica asume el nombre de las columnas devueltas por el SELECT.
- Sentencia_select. Sentencia SELECT que proporciona la definición de la vista. Se pueden usar tablas u otras vistas de la base de datos.

Ejemplo **creación de vista** en la que se crea la vista **Clasificacion** que genere la clasificación general de la **carrera_id** igual a **1** en la que se muestra el código del corredor, apellidos, nombre, edad, club, categoría y tiempo invertido.

```
-- Creación de vistas
CREATE OR REPLACE VIEW Clasificacion AS
    SELECT
        co.id,
        co.Apellidos,
        co.Nombre,
        co.Edad,
        cl.nombre AS Club,
        ca.nombre AS Categoria,
        r.tiempoinvertido
    FROM
        Corredores co
            INNER JOIN
        Clubs cl ON co.club_id = cl.id
            INNER JOIN
        categorias ca ON co.categoria_id = ca.id
            INNER JOIN
        registros r ON r.corredor_id = co.id
    WHERE
        r.carrera_id = 1
    ORDER BY tiempoinvertido;
```

Una vez definida una vista ésta tiene el “mismo tratamiento” que si fuese una tabla más de la base de datos. Veamos el siguiente ejemplo en el que a partir de la vista anterior Clasificacion obtengo sólo la clasificación de la categoría “Senior A”.

```
SELECT
    id,
    Apellidos,
    Nombre,
    Edad,
    Club,
    Categoria,
    tiempoinvertido
FROM
    clasificacion
WHERE
    categoria LIKE 'Senior A'
ORDER BY tiempoinvertido;
```

El siguiente comando SQL permite mostrar la sentencia SELECT a partir de la cual está creada una vista:

```
SHOW CREATE VIEW nombre_vista;
```

6.10.2 Modificar Vista. ALTER VIEW

Permite la modificación de una vista existente.

Sintaxis

```
ALTER VIEW nombre_vista [(lista_columnas)] AS sentencia_select;
```

Normalmente este comando no se suele utilizar, a cambio se usa el CREATE VIEW con la cláusula OR REPLACE.

6.10.3 Eliminar Vista. DROP VIEW

Permite la eliminación de una o más vistas existente de la base de datos.

```
DROP VIEW [IF EXISTS ]nombre_vista [, nombre_vista];
```