

Por definición, SQL es un lenguaje estructurado de búsqueda de datos. Este lenguaje nos provee una poderosa herramienta para consulta y procesamiento de datos en pequeños y grandes volúmenes.

SQL

SQL estándar, de menos a más

Federico Roberto Dávila

Contenido

Introducción a Base de Datos.....	5
Modelo Entidad-Relación	5
Interfaz de consultas	8
Estructura del lenguaje	15
Operadores.....	22
Modelado de elementos de datos	27
Tipos de datos	30
Consultas relacionadas.....	35
Funciones de Base de Datos.....	39
Agrupaciones.....	50
Normalización de Base de datos	53
Sub Query	57
Stored Procedure	60
Sentencias preparadas	75

Clase 1:

Introducción a base de datos:

¿Qué es una base de datos?

¿Qué es una base de datos relacional?

Modelo Entidad-Relación

¿Qué es una entidad?

¿Qué son los atributos?

¿Qué son las relaciones?

Cardinalidad

Modelo de datos

Interfaz de Consultas

Introducción a la interfaz

Ejercicios

Clase 2:

Estructura del lenguaje

¿Qué es DDL?

CREATE

ALTER

DROP

TRUNCATE

¿Qué es DML?

INSERT

UPDATE

DELETE

SELECT

Ejercicios

Clase 3:

Operadores

Lógicos

De ordenamiento

Ejercicios

Clase 4:

Modelado de elementos de datos

Tablas

Restricciones

PK y FK

Ejercicios

Tipos de datos

Clase 5:

Consultas relacionadas

Left Join

Right Join

Inner Join

Union

Ejercicios

Clase 6:

Funciones

De agregado

Escalaes

Del sistema – Parte I

Ejercicios

Clase 7:

Funciones del sistema – Parte II

Agrupaciones

Group By

Having

Ejercicios

Clase 8:

Normalización

Clase 9:

Sub Query

Clase 10:

Stored Procedure

Definición

Variables

Parámetros

Estructuras de control – Parte I

Case 11:

Stored Procedure

Estructuras de control – Parte II

Cursores

Transacciones

Clase 12:

Sentencias preparadas

Repaso

Clase 13:

Examen

Clase 14:

Recuperatorio

BORRADOR

Introducción a Base de Datos

¿Qué es una base de datos?

Llamamos base de datos (BD) a un conjunto de datos pertenecientes a un mismo contexto y almacenados sistemáticamente para su posterior uso.

Extraído de http://es.wikipedia.org/wiki/Base_de_datos

¿Qué es una base de datos relacional?

Llamamos así a toda Base de Datos que cumple con el modelo relacional. Dicho paradigma es el más popular hoy en día, permitiendo generar relaciones entre los datos guardados en diferentes tablas, y a través de estas relaciones, conectar dichas tablas.

Características:

- Cada tabla será un conjunto de registros, conformados por filas y columnas.
- Se compone de varias tablas y relaciones entre ellas.
- No pueden existir dos o más tablas con igual nombre.
- Las relaciones se llevarán a cabo a través de campos especiales, conocidos como claves principales y foráneas; que ya trataremos en detalle.

Entre los gestores o manejadores actuales más populares encontramos: MySQL, PostgreSQL, Oracle, DB2, INFORMIX, Microsoft SQL Server, etc..

Modelado

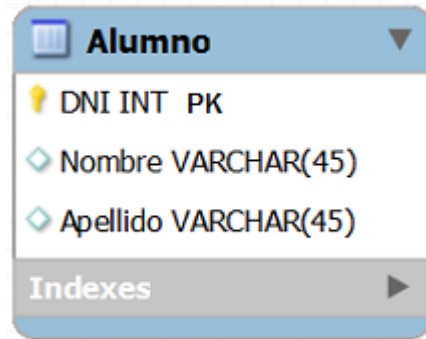
¿Qué es una entidad?

Entidad es una representación de un objeto (una casa), ser (una persona) o concepto (licencia de conducir) del mundo real que se describe en una base de datos.

Cada entidad estará construida por uno o más atributos. Por ejemplo, una Persona tendrá los atributos nombre, apellido, DNI, etc..

¿Qué son los atributos?

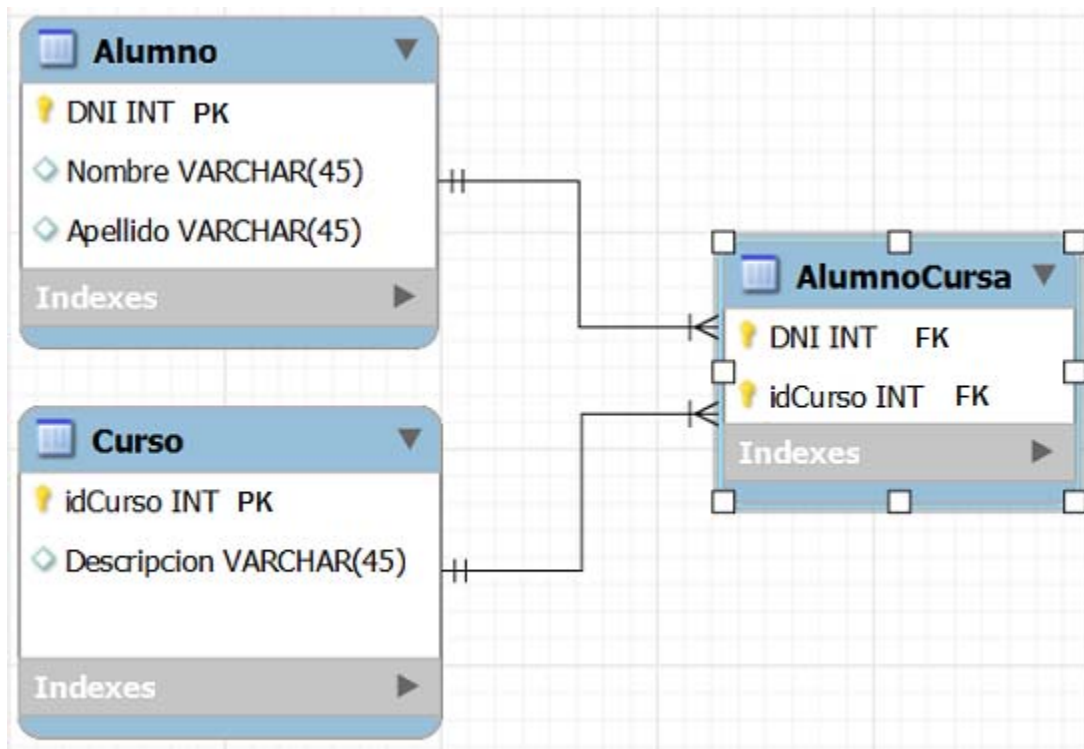
Las entidades contienen atributos para representar las propiedades que nos interesan almacenar. En la BD se almacenarán como columnas o campos de las tablas.



¿Qué son las relaciones?

Es la descripción de ciertas interdependencias entre una o más entidades. Para definir una relación entre dos entidades utilizaremos las claves primarias (PK) y foráneas (FK).

- Clave Primaria o Primary Key: llamaremos así a un atributo (o combinación de varios) que se encargará de identificar unívocamente a cada fila de dicha entidad. Por ejemplo, en una tabla de personas podríamos utilizar su DNI, ya que este es único e irreplicable para cada persona.
- Clave Foránea o Foreign Key: esta clave identificará a un atributo (o conjunto de ellos) en una tabla que se refieren a un atributo (o conjunto de ellos) en otra tabla. Las FK harán referencia en casi todos los casos a la PK de la otra tabla. Por ejemplo, si nuestra base de datos es de nuestro Laboratorio de Software Libre, queremos identificar que alumno está inscripto en que curso, sin repetir constantemente toda la información de dicha persona. Nuestro modelo quedaría así:



¿Qué es la cardinalidad de una relación?

La cardinalidad indicará el número de instancias de una entidad de la tabla A con las que estarán relacionadas en la tabla B. Para aclarar el tema, veremos cómo puede ser esta relación:

- Uno a uno: una instancia de la tabla A se puede relacionar con una y sólo una instancia de la tabla B. Por ejemplo: un alumno puede ocupar un solo banco en el aula.
- Uno a varios: una instancia de la tabla A se relacionará con varias instancias de la tabla B. Por ej.: una misma persona puede ser titular de muchas tarjetas de crédito, pero una tarjeta de crédito sólo puede tener un titular.
- Varios a varios: varias instancias de la tabla A se podrán relacionar con varias instancias de la tabla B. Por ej.: un veterinario puede atender a varios animales, y a su vez esos animales pueden ser atendidos por varios doctores.

¿Qué es un modelo de datos?

Una entidad se describe en la estructura de la base de datos empleando un modelo de datos.

Llamamos modelo de datos es una colección de conceptos (entidades, atributos y relaciones entre ellos) empleados para describir la estructura de la Base de Datos.

¿Para qué sirve el modelo Entidad-Relación?

Es una herramienta para el modelado de datos del mundo real, permitiéndonos representar entidades relevantes, así como sus interrelaciones y propiedades.

¿Cómo debe crearse este modelo?

Suele cometerse el error de ir creándose nuevas tablas a medida que las necesitamos, generando así el modelo de datos y la construcción física de las tablas simultáneamente. En realidad, el modelo debería ser el análisis previo de una situación real, que luego se plasmará en una Base de Datos relacional.

Interfaz de consultas

¿Qué es la interfaz de consultas?

Muy superficialmente, la interfaz de consultas será el medio para poder ejecutar consultas sobre nuestras tablas de la base de datos.

¿Qué interfaz de usuario utilizar?

La alternativa que ofrecemos para este curso es principalmente el suministrado por la comunidad de MySQL (puede ser tanto el viejo Query Browser, como su reciente reemplazo, el MySQL Workbench) disponible en <http://dev.mysql.com/downloads/gui-tools/5.0.html> (para usuarios de MAC, Linux o Windows).

Para el fin de curso, vamos a administrar bases de datos locales (o sea, en nuestra propia máquina). Para esto debemos instalar también el Motor de Base de Datos, el cual es el servicio principal para almacenar, procesar y proteger los datos. Recomendamos, para una mayor simplicidad, instalar el XAMPP, disponible en <http://www.apachefriends.org/es/xampp.html>, el cual es un servidor Apache que contiene el motor de MySQL y lo administra fácilmente.

Si quieren saber más sobre XAMPP, ingresen en <http://www.apachefriends.org/es/xampp.html>.

Como vamos a ver ANSI SQL, lo cual es estándar para todas las bases de datos SQL, también podrían utilizar el Microsoft SQL Server 2008 Management Studio Express disponible en su versión para desarrolladores en <http://www.microsoft.com/es-es/download/details.aspx?id=7593> (esta última opción es sólo válida para usuarios Windows).

¿Por qué recomendamos estas alternativas?

Sencillamente porque son gratuitos y cumplen con el estándar ANSI SQL, el estándar de consultas del lenguaje. Todo lo visto a lo largo de este curso cumplirá con el estándar, siendo sencillo migrar nuestras consultas de un motor a otro. Como dicho estándar es respetado por MySQL, y cuenta con la ventaja de ser multiplataforma, lo consideramos la mejor alternativa.

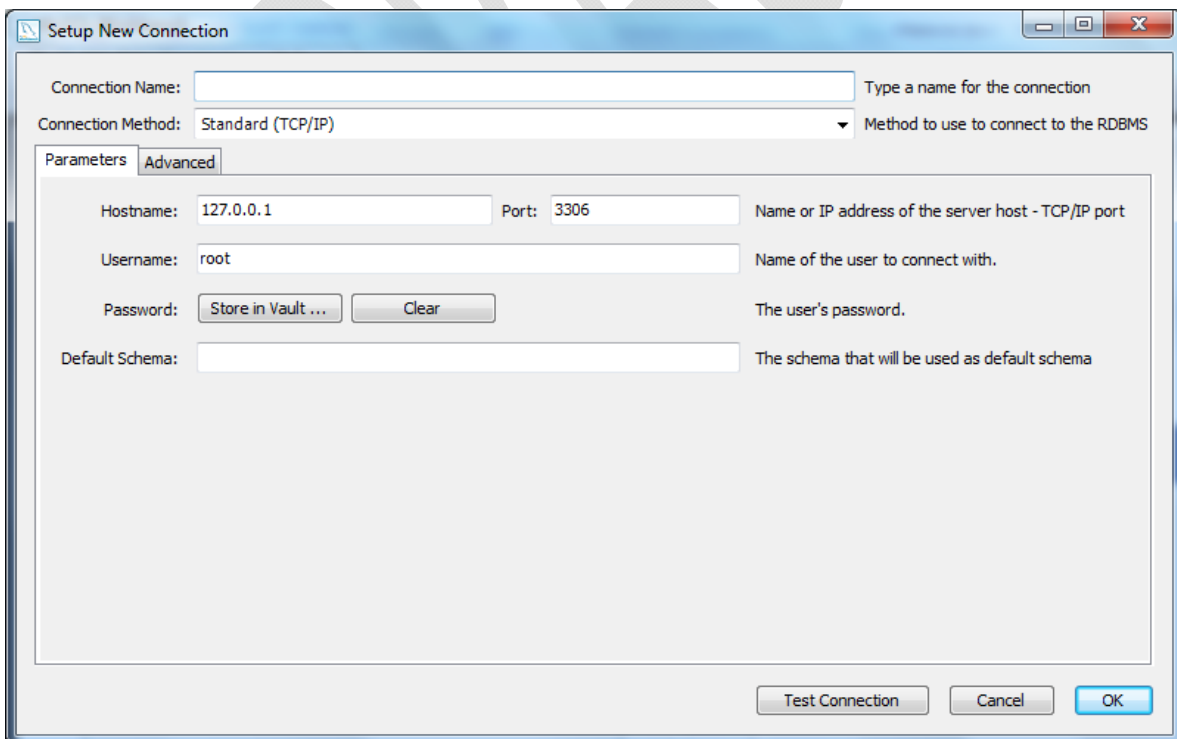
Conexión a la Base de Datos

Tras instalar nuestra interfaz de consulta y el XAMPP, debemos ejecutar en este último el motor de MySQL (simplemente ejecutando el programa y haciendo click en START). Luego ejecutaremos la interfaz elegida.

A la izquierda de la pantalla veremos las conexiones que ya realizamos o podemos elegir, por debajo de dicho cuadro, NEW CONNECTION. Ahí se nos pedirán los siguientes datos:

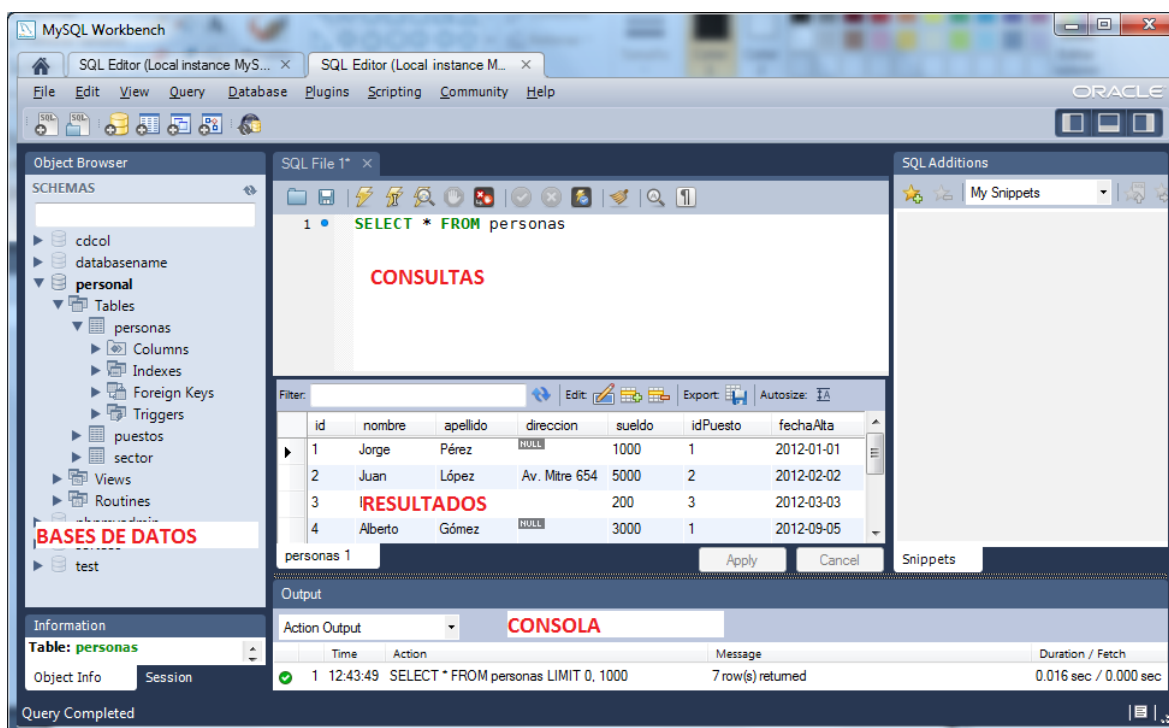
- Servidor (o Hostname): localhost o 127.0.0.1
- Usuario (Username): root
- Puerto (Port): dejar el por defecto (3306)
- Clave (Password):
 - Windows: vacío
 - Linux: el cargado al momento de la instalación

Nos pide el nombre del servidor y el puerto de conexión porque estableceremos una conexión TCP. Esto nos permitirá conectarnos tanto con una base de datos en nuestra propia máquina, como conectarnos con una base de datos en un servidor remoto, y así administrar, por ejemplo, una base de un sitio web desde cualquier máquina.



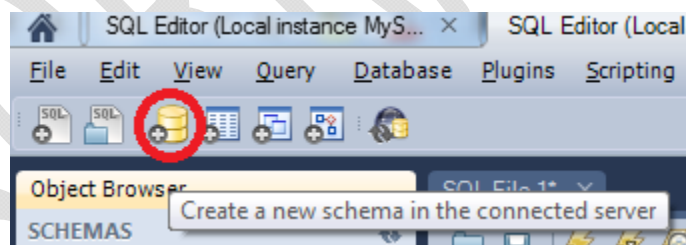
Esa será la configuración por defecto del motor. Dar OK y esperar unos segundos a que se establezca la conexión.

Aquí nos encontraremos una ventana similar a la siguiente:



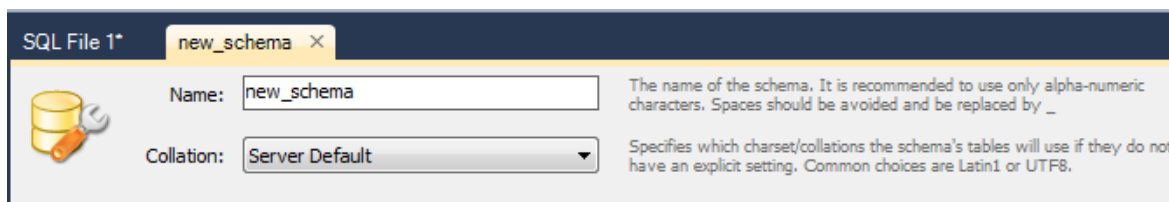
En rojo encontramos resaltado para que nos sirve cada una de las secciones de la pantalla.

Para crear nuestra primera base de datos, simplemente haremos click sobre el ícono ubicado por encima de nuestra sección **BASE DE DATOS**:

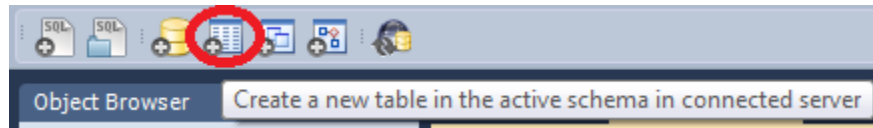


Ahí nos preguntará el nombre. Para esta descripción suele utilizarse el formato de separar las palabras con guión bajo “_” y todo en minúsculas (por ejemplo, nombre_base_datos).

Tras cargar el nombre de nuestra base, presionamos **APPLY**.



Luego, de igual forma, crearemos nuestra primer tabla. Primero seleccionaremos nuestra base de datos. Luego haremos lo propio en el ícono situado al lado del utilizado para crear nuestra base.



Aquí nos aparecerá una nueva ventana:

Column Name	Datatype	PK	NN	UQ	BIN	UN	ZF	AI	Defa
⚡ ATRIBUTOS	INT	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	

Below the table, there are input fields for 'Column Name', 'Data Type', 'Collation' (Table Default), and 'Default'. To the right of these fields are several checkboxes: Prim, Not, Uniq, Bina, Unsi, Zero, and Auto Increment. At the bottom of the dialog are 'Apply' and 'Revert' buttons.

En el primer campo colocaremos el nombre de nuestra tabla. Para esta descripción, y todas las próximas que veamos, suele utilizarse camelCase, lo cual se describe como todas las palabras juntas, la primera comenzada en minúscula y luego cada primer letra en mayúscula (por ej., nombreTabla).

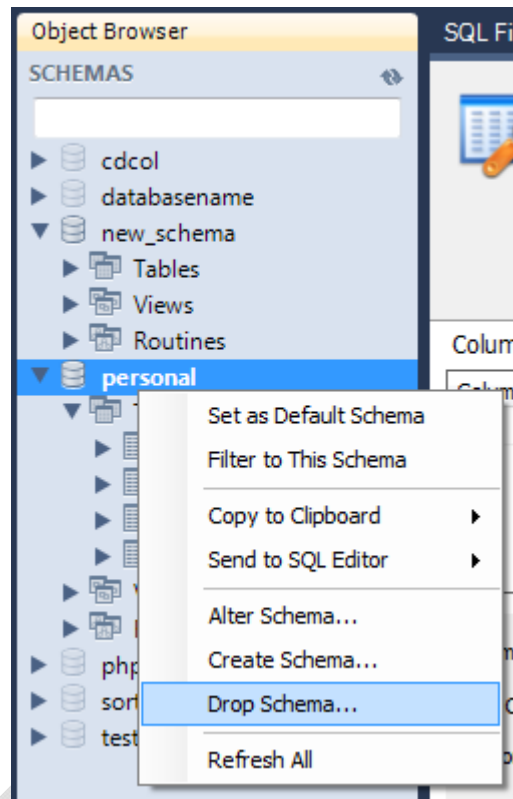
Luego nos centraremos en el cuadro inferior, con la cita COLUMNS. Aquí cargaremos los atributos de la tabla.

Al colocarnos sobre cada uno de los atributos, podremos modificarles los siguientes valores:

- **Primary Key:** el ID de la tabla debe ser marcado como Clave Primaria (PK). Siempre, al utilizar el MySQL Workbench, el primer atributo que carguemos se marcará como PK. En caso de querer cambiarlo manualmente, o querer agregar más de un índice para conformar dicha PK, hacemos click sobre el valor Primary Key.
- **Column Name:** nombre de la columna. Aquí colocaremos los atributos de la entidad (id, nombre, apellido, etc.).
- **Datatype:** el tipo de dato es avisarle a la BD que clase de valor será cargado en este atributo: numérico (INTEGER, SMALLINT, etc.), cadena de texto (VARCHAR, LONGTEXT, etc.), fecha (DATETIME, DATE, etc.); por dar algunos ejemplos.
- **NOT NULL (NN):** si el campo acepta o no valores nulos. Un valor nulo es la ausencia de valor, el vacío.
- **AUTO INCREMENT (AI):** si el campo es un valor numérico autoincrementable, o sea que seguirá una sucesión numérica automática (o sea, 1, 2, 3, 4, ...). Esto es ideal para los índices ID (identifier), o sea para las claves primarias de tablas sin un identificador único. Por ejemplo, tenemos una base de datos de un teatro con sus obras; la tabla Obras no tendrá ni legajo, ni DNI, ni nada que haga a cada obra única e irrepetible. En ese caso deberemos agregar un campo más, el cual podremos llamar idObra y será autoincrementable; identificando cada una de ellas de forma unívoca.
- **Default Value:** el valor por defecto que tomará este campo, si no es cargado con ningún valor.
- **Comment:** un comentario del usuario, sin uso práctico.

Cargamos los datos, y luego hacemos click “Apply”.

Para borrar tanto tablas como base de datos completas, hacemos click derecho y seleccionamos “Drop Schema”.



Ejercicios

1. Crear una base de datos llamada Empresa.
2. Crear una tabla Empleados con los siguientes atributos:
 - a. ID - INTEGER
 - b. Nombre - VARCHAR(255)
 - c. Apellido - VARCHAR(255)
 - d. Direccion - VARCHAR(50)
 - e. ID puesto - SMALLINT
3. Crear una segunda tabla llamada Puestos:
 - a. ID - SMALLINT
 - b. puesto - VARCHAR(45)

personas
id INT(11)
nombre VARCHAR(255)
apellido VARCHAR(255)
direccion VARCHAR(50)
idPuesto SMALLINT(5)
Indexes

puestos
id SMALLINT(5)
puesto VARCHAR(45)
Indexes

Estructura del lenguaje

¿Qué es DDL?

Llamamos DDL (Data Definition Language o Lenguaje de Definición de Datos) a aquellas instrucciones que se encargarán de la modificación de la estructura de objetos una base de datos. Las operaciones básicas son:

- CREATE
- ALTER
- DROP
- TRUNCATE

Volvamos a nuestra interfaz de consultas y probémoslo.

Primero, crearemos nuestra primera tabla mediante el comando CREATE:

```
CREATE TABLA nombre_tabla (campoUno tipoDato [modificadores], [campoDos, ...])
```

Ejemplo:

```
CREATE TABLE personas  
(  
    id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,  
    nombre VARCHAR(50),  
    apellido VARCHAR(50)  
)
```

Si luego quisiéramos agregar un nuevo campo a esta tabla, deberíamos utilizar ALTER:

```
ALTER TABLA nombre_tabla ADD (campoNuevo1 tipoDato [modificadores],  
[campoNuevo2, ...]);
```

Ejemplo:

```
ALTER TABLE personas ADD direccion VARCHAR(50);
```

En el caso de querer borrar todos los datos contenidos en una tabla, y resetear su ID autoincrementable (hacer que al ingresar un nuevo valor este arranque la cuenta desde 1 nuevamente), debemos utilizar TRUNCATE. Recordar que siempre vuelve la tabla a foja cero, borra absolutamente todo:

```
TRUNCATE TABLA nombre_tabla;
```

Ejemplo:

TRUNCATE TABLE personas;

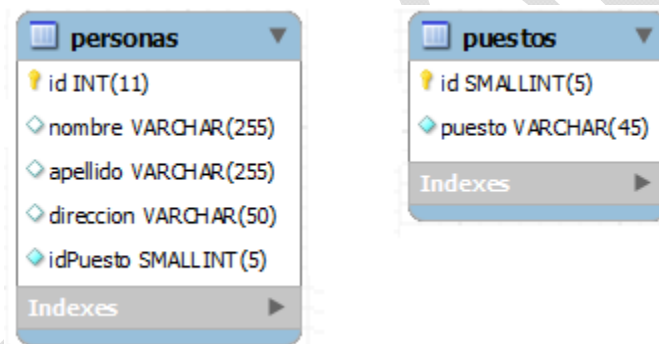
Finalmente, para eliminar toda la estructura de una tabla y su contenido, utilizaremos DROP de la siguiente forma:

DROP TABLA nombre_tabla;

Ejemplo:

DROP TABLE personas;

Volvamos a dejar nuestra tabla como al finalizar el ejercicio anterior:



¿Qué es DML?

Llamamos DML (Data Manipulation Language o Lenguaje de Manipulación de Datos) a los lenguajes que nos proporcionan por un sistema de gestión de base de datos, los cuales nos permiten llevar a cabo consultas o manipulación de datos. SQL es un lenguaje de DML, el más utilizado hoy en día.

Las sentencias que utilizaremos son:

- INSERT
- UPDATE
- DELETE
- SELECT

Las primeras 3 son sentencias de alteración de datos, y la última es una sentencia de consulta.

La sentencia **INSERT** es utilizada para cargar datos dentro de nuestras tablas, de a uno por vez. Su nomenclatura es:

INSERT INTO tabla_nombre (columna1, [columna2, ...]) VALUES (valor1, [valor2, ...]);

Ejemplo:

```
INSERT INTO personas (nombre,apellido) VALUES ('José','Pérez');
```

Podremos obviar los nombres de las columnas, lo cual nos agregará tres complicaciones extras:

1. Debemos insertar la totalidad de las columnas.
2. Debemos respetar el orden en que las columnas fueron creadas en la tabla.
3. Debemos cargar el ID a mano, a pesar de que este pueda ser autoincremental, lo cual puede provocar errores en la carga.

```
INSERT INTO personas VALUES (2,'Juan','López', 'Av. Mitre 654');
```

Podremos hacer cargas múltiples en una sola sentencia, utilizando la nomenclatura:

```
INSERT INTO tabla_nombre (columna1, [columna2, ...]) VALUES (valor1a, [valor2a, ...]),  
(valor1b, [valor2b, ...]);
```

Ejemplo:

```
INSERT INTO personas (nombre,apellido) VALUES ('Ernesto','Giménez'),('Alberto','Gómez');
```

La sentencia **UPDATE** es utilizada para modificar datos ya cargados dentro de nuestras tablas. Su nomenclatura es:

```
UPDATE tabla_nombre  
SET columna1=valor1, [columna2=valor2, ...]  
WHERE columna3=valor3;
```

Ejemplo:

```
UPDATE personas  
SET nombre='Jorge'  
WHERE id = 1;
```

WHERE es la palabra clave que nos indicará que condición deben tener las filas para que la sentencia las utilice, en este caso, para modificar sus datos.

Es importante aclarar que el WHERE no es obligatorio. De no colocarlo, se actualizarán TODOS los registros de la tabla. En el caso del ejemplo anterior, TODAS las personas pasarían a llamarse “Jorge” haciendo:

```
UPDATE personas SET nombre='Jorge';
```

La sentencia **DELETE** es utilizada para borrar datos dentro de nuestras tablas. Su nomenclatura es:

```
DELETE FROM tabla_nombre [WHERE columna1=valor1] [LIMIT cantidad_registros];
```

Ejemplo:

```
DELETE FROM personas WHERE nombre = 'José';
```

En el caso de que sólo quisiésemos borrar una cantidad X de ocurrencias, podríamos utilizar LIMIT. En el siguiente ejemplo, eliminaremos sólo el primer registro:

```
DELETE FROM personas LIMIT 1;
```

Es importante aclarar que el WHERE no es obligatorio. De no colocarlo, se borarán TODOS los registros de la tabla.

La sentencia **SELECT** es utilizada para consultar registros dentro de nuestras tablas. Su nomenclatura básica es:

```
SELECT columna1,[columna2, ...] FROM tabla_nombre;
```

Ejemplo:

```
SELECT nombre FROM personas;
```

Para este caso, también podremos utilizar el comodín * en lugar de los nombres de nuestras columnas, seleccionando de este modo todos los datos.

```
SELECT * FROM personas;
```

Las consultas podrán ser filtradas por medio de la sentencia WHERE, como vemos a continuación:

```
SELECT nombre  
FROM Personas  
WHERE id = 1;
```

Conociendo el uso básico del SELECT, también podremos aplicarlo al INSERT:

```
INSERT INTO tabla_nombre (columna1, [columna2, ...])  
SELECT columna1, [columna2, ...] FROM tabla_nombre2;
```

Ejemplo:

```
INSERT INTO personas2 (nombre, apellido)  
SELECT nombre, apellido  
FROM personas;
```

Operadores matemáticos

Otros operadores que podemos utilizar son:

- = (igual)
- > (mayor)
- >= (mayor o igual)
- < (menor)
- <= (menor o igual)
- != (distinto)
- IS NULL (el valor es nulo)
- IS NOT NULL (el valor no es nulo)

Dentro del SELECT podemos usar operadores matemáticos. Estos son:

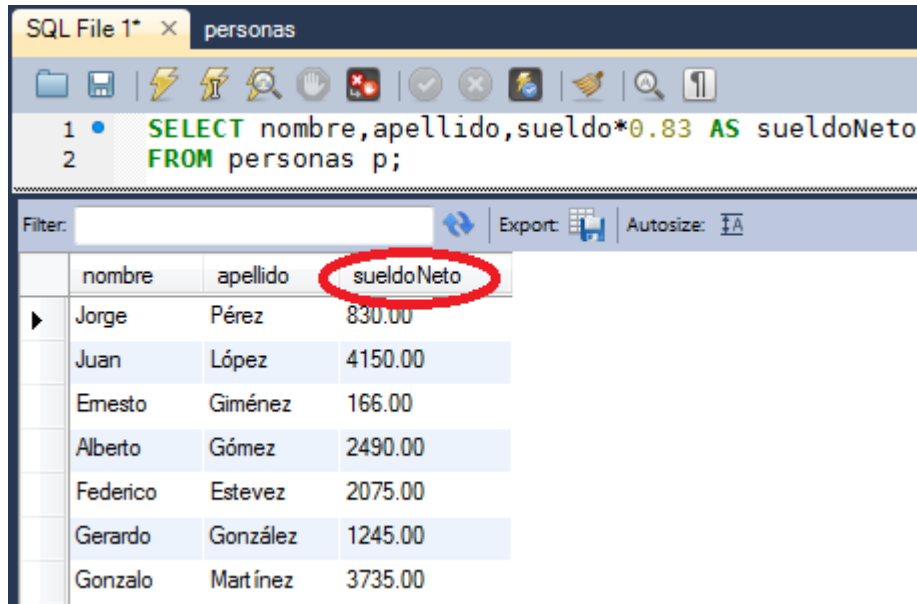
- + (suma)
- - (resta)
- * (multiplicación)
- / (división)
- % (módulo)

Con estos operadores podremos actuar tanto dentro de los campos que son resultante de la consulta del SELECT (se ve en el ejemplo) como en los de condiciones (WHERE). Por ejemplo, podríamos calcular sobre el sueldo bruto cuanto es el sueldo neto que la persona iría a percibir:

```
SELECT nombre,apellido,sueldo*0.83  
FROM personas p;
```

Por último, por ahora, tenemos algo llamado Alias. Los alias nos sirven para renombrar una columna, dándole un nombre más claro, útil o descriptivo. Por ahora, esto sólo será a modo ilustrativo, pero nos servirá mucho cuando veamos la cláusula de agrupamiento:

```
SELECT nombre,apellido,sueldo*0.83 AS sueldoNeto
FROM personas;
```



	nombre	apellido	sueldoNeto
▶	Jorge	Pérez	830.00
	Juan	López	4150.00
	Ernesto	Giménez	166.00
	Alberto	Gómez	2490.00
	Federico	Estevez	2075.00
	Gerardo	González	1245.00
	Gonzalo	Martínez	3735.00

Los alias también pueden ser aplicados a las tablas, en este caso sin colocar la palabra clave AS. Por ejemplo:

```
SELECT p.nombre, p.apellido, p.sueldo*0.83 AS sueldoNeto
FROM personas p;
```

En el FROM renombramos la entidad Personas como p. Para hacer referencia a alguno de los campos de Personas, podremos colocar p.nombre, por ejemplo.

Resumiendo

Básicamente cada sentencia se escribe como se lee. ¿Cómo es esto? Ejemplifiquémoslo.

- INSERT: Insertar en TABLA (los campos) los valores (valores)
- UPDATE: Actualizar la TABLA configurando los valores como CONDICIÓN X, donde los campos tengan las siguientes condiciones.
- DELETE: Borrar de la TABLA donde los campos tengan las siguientes condiciones.
- SELECT: Seleccionar los campos desde la tabla, donde los campos tengan las siguientes condiciones.

Ejercicio:

1. Crear una nueva columna en personas llamada 'sueldo' del tipo INTEGER.

2. Insertar datos, cargando todos los valores de cada registro.
3. Actualizar nuestros registros cargados previamente con distintos valores de sueldo.
4. Seleccionar nombre, apellido y retenciones del sueldo en Argentina (17%).

Resolución:

1. ALTER TABLE personas ADD sueldo INTEGER;
2. INSERT INTO personas (nombre,apellido,direccion,sueldo) VALUES ('Federico','Estevez','NN',2500),('Gerardo','González','NA',1500);
3. UPDATE personas SET sueldo = 1000 WHERE id=1;
4. SELECT nombre,apellido,sueldo*0.17 FROM personas p;

Operadores

Operadores lógicos

Otros operadores útiles de la cláusula *WHERE* son el *AND* y *OR*. Estos se utilizan para concatenar filtros. Por ejemplo:

```
SELECT nombre, apellido, sueldo
FROM personas
WHERE sueldo > 1000 AND direccion IS NOT NULL;
```

También podemos utilizar el *BETWEEN*, un operador que nos dejará consultar por un dato entre dos valores de la siguiente forma:

```
SELECT nombre, apellido, sueldo
FROM personas
WHERE sueldo BETWEEN 100 AND 2000;
```

Esta cláusula, como la gran mayoría, también puede ser negada con el modificador *NOT*:

```
SELECT nombre, apellido, sueldo
FROM personas
WHERE sueldo NOT BETWEEN 1000 AND 2000;
```

Por otro lado, existe la sentencia *IN*, la cual sirve para determinar si el valor de una expresión está dentro de una serie de valores dados:

```
SELECT nombre, apellido, sueldo
FROM personas
WHERE sueldo IN (1000,2000);
```

También puede implementarse el *NOT IN*.

Los valores que se le pasan a esta cláusula pueden ser los devueltos por otra consulta.

La cláusula *LIKE* sirve para hacer comparaciones entre cadenas de caracteres y patrones. Nos da la ventaja de poder buscar información dentro de una cadena utilizando dos comodines:

- % marca que pueden aparecer 0, 1 o más caracteres, sin importar cuales
- _ marca que debe aparecer al menos un, y sólo un, carácter, cualquiera sea.

Su utilización es simple:

```
SELECT nombre, apellido, sueldo
FROM personas
WHERE nombre LIKE '_na%';
```

La expresión concordará con nombres que comiencen con una letra cualquiera, continúen con 'na' y luego sigan de cualquier forma (por ej.: Ana, Analía, Anacleto, Anastasio).

También se puede combinar, o ir en cualquier parte del patrón:

```
SELECT nombre, apellido, sueldo
FROM personas
WHERE nombre LIKE 'J__n C%s';
```

Esta expresión coincidirá con el nombre "Juan Carlos", por ejemplo.

Algunas aclaraciones sobre el LIKE:

- Este comando no distingue entre mayúsculas y minúsculas, al menos que se le indique por medio de Casting (más adelante en este apunte)
- Para poder buscar dentro del patrón con los caracteres reservados (entiéndase % y _), se los debe anteceder con el carácter de escape contra barra \. Al colocar \% o _ o \\, se buscará que dicha cadena contenga esos caracteres.

```
SELECT nombre, apellido, sueldo
FROM personas
WHERE nombre LIKE '\_tabla';
```

Operador de ordenamiento

Para ordenar los resultados de una consulta tenemos el operador ORDER BY. Con este indicamos la/s columna/s por las que queremos ordenar, teniendo mayor peso la que se encuentre más a la izquierda. Su nomenclatura es:

```
SELECT columnas FROM tabla ORDER BY columna1, [columna2, ...];
```

Ejemplo:

```
SELECT nombre, apellido
FROM personas
ORDER BY apellido;
```


SQL File 1* x personas

```

1 SELECT nombre, apellido
2 FROM personas
3 ORDER BY apellido;

```

Filter: Export:

nombre	apellido
Federico	Estevez
Ernesto	Giménez
Alberto	Gómez
Gerardo	González
Juan	López
Gonzalo	Martínez
Jorge	Pérez

El ordenamiento, por defecto, será ascendente. Si quisiéramos ordenar en forma descendente, deberíamos colocar el modificador DESC al final:

```

SELECT nombre, apellido
FROM personas
ORDER BY apellido DESC;

```

SQL File 1* x personas

```

1 SELECT nombre, apellido
2 FROM personas
3 ORDER BY apellido DESC;

```

Filter: Export:

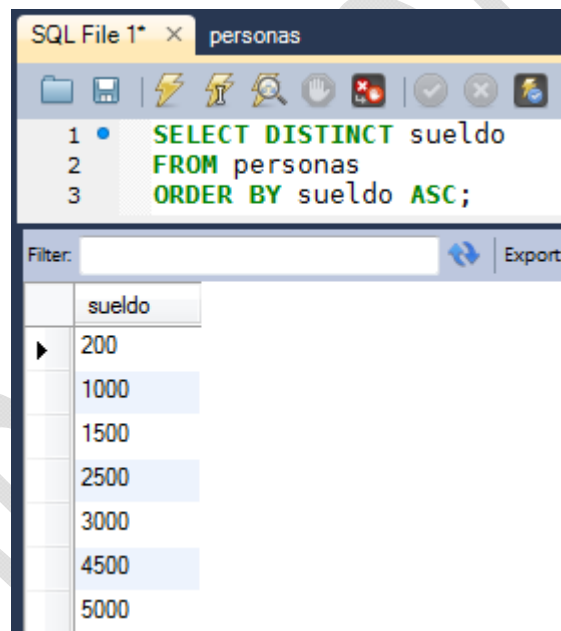
nombre	apellido
Jorge	Pérez
Gonzalo	Martínez
Juan	López
Gerardo	González
Alberto	Gómez
Ernesto	Giménez
Federico	Estevez

También podríamos aclarar que el ordenamiento es ascendente por medio del modificador ASC.

Por último, aunque sólo por ahora, veremos el operador DISTINCT. Tras la palabra SELECT se pueden poner 3 operadores: ALL (no hace falta escribirlo, es el valor por defecto), DISTINCT y DISTINCTROW (estos últimos dos son sinónimos).

DISTINCT nos ayuda a filtrar resultados repetidos. Por ejemplo, si quisiéramos conseguir los sueldos de los empleados, sin repeticiones, sólo los distintos sueldos que tenemos cargados, y ordenados de mayor a menor, deberíamos escribir:

```
SELECT DISTINCT sueldo
FROM personas
ORDER BY sueldo ASC;
```



Ejercicio:

1. Recoger todos los sueldos mayores a 1000 y menores a 5000, ordenados de mayor a menor de forma explícita.
2. Consultar los sueldos mayores o iguales a 2000, sin repetir ocurrencias, ordenados de menor a mayor de forma explícita.
3. En sólo una consulta, traer todos los nombres que comiencen con J, y los apellidos finalizados en Z.
4. Informar los datos de todas las personas apellidadas Gómez, González y Pérez.

Resolución:

1. `SELECT nombre,apellido,sueldo FROM personas WHERE sueldo BETWEEN 1000 AND 5000 ORDER BY sueldo DESC;`

2. `SELECT nombre,apellido,sueldo FROM personas WHERE sueldo > 2000 ORDER BY sueldo ASC;`
3. `SELECT nombre,apellido FROM personas WHERE nombre LIKE 'J%' or apellido LIKE '%z';`
4. `SELECT nombre,apellido FROM personas WHERE apellido IN ('Gómez','González','Pérez');`

BORRADOR

Modelado de elementos de datos

Tablas:

Las tablas son utilizadas por SQL para almacenar datos, dejándolos a disposición de los usuarios para que estos lo manipulen a través de sentencias, como las que vimos antes.

Se define a las tablas como objetos compuestos por una estructura (conjunto de columnas), que almacenarán información interrelacionada (filas) acerca de un objeto en particular. Algunas de sus características más importantes serán:

- Un nombre único y unívoco.
- Compuestas por registros y columnas.
- Dichos registros y columnas podrán estar en diferentes órdenes.
- Una base de datos tendrá muchas tablas, donde cada una almacenará información de forma particular.
- Las columnas tendrán un nombre único e irrepetible dentro de una misma tabla.
- No podrá haber dos registros con el mismo valor de PK.

Restricciones de las columnas:

Una columna tiene dos restricciones importantes para seleccionar al momento de crearla:

- Not NULL: definiremos si la columna podrá, o no, alojar un valor nulo, inexistente.
- No Duplicates: se indicará si el valor de un registro de esta columna se podrá repetir en otro. Por ejemplo, si quisiéramos controlar que no haya dos personas con exactamente el mismo sueldo, tildaríamos esta opción.

Clave Primaria (PK)

Definiremos a la PK como una columna, o conjunto de ellas, que forzarán la integridad de los datos, asegurándonos que cada registro de la tabla será único, al menos diferenciándose en este valor. Los campos que conformen la PK no aceptarán valores nulos ni duplicados. Por ejemplo, en nuestra tabla “personas” podríamos haber usado el número de documento de la persona.

Clave Foránea (FK)

La FK nos permitirá llegar a un dato presente en otra entidad, a través de una relación entre ellos. Generalmente, una FK se ata con la PK de la otra tabla.

Por ejemplo, nosotros tenemos nuestra tabla Personas con los empleados de la empresa y nuestra tabla Puestos con los puestos que estos empleados ocupan. Nuestra tabla Puestos tiene como PK el atributo ID, un entero chico y autoincrementable. En nuestra tabla personas tendremos un campo del MISMO tipo de dato (SMALLINT, en este ejemplo), al que llamaremos idPuesto. idPuesto deberá contener valores que estén representados en la tabla Puestos. idPuesto será nuestra FK.



Algunos motores de base de datos, como SQL Server, comprueban cosas como por ejemplo que no borremos un Puesto asignado a algún Empleado, para no generar un problema de consistencia de los datos. Esto se llama Integridad Referencial.

Ejercicio:

Con el editor de consultas:

1. Crear la tabla Puestos con un SMALLINT para el ID y un campo VARCHAR llamado puesto.
2. Crear el campo ID Puesto dentro de nuestra tabla personas.
3. Cargar datos dentro de la tabla puestos.
4. Asignar valores a la columna ID Puesto en la tabla personas.
5. Atar la referencia PK ID Puesto con FK ID Puesto.

Resolución:

1.

```
CREATE TABLE `personal`.`puestos` (
  `id` SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT,
  `puesto` VARCHAR(45) NOT NULL DEFAULT "",
  PRIMARY KEY(`id`)
) ENGINE = InnoDB;
```
2.

```
ALTER TABLE `personal`.`personas` ADD COLUMN `idPuesto` SMALLINT UNSIGNED NOT
NULL DEFAULT 0 AFTER `sueldo`;
```
- 3.

- 4.
5.

```
ADD CONSTRAINT `FK_puesto_personas` FOREIGN KEY `FK_puesto_personas` (`idPuesto`)
REFERENCES `puestos` (`id`)
ON DELETE RESTRICT
ON UPDATE RESTRICT;
```

BORRADOR

Tipos de datosⁱ

Los tipos de datos no son respetados de igual manera por todos los motores. A partir de la versión 4.xx.xx los tipos de datos para MySQL son los siguientes:

Tipos Numéricos

Existen tipos de datos numéricos, que se pueden dividir en dos grandes grupos, los que están en coma flotante (con decimales) y los que no.

TinyInt: es un número entero con o sin signo. Con signo el rango de valores válidos va desde -128 a 127. Sin signo, el rango de valores es de 0 a 255

Bit ó Bool: un número entero que puede ser 0 ó 1

SmallInt: número entero con o sin signo. Con signo el rango de valores va desde -32768 a 32767. Sin signo, el rango de valores es de 0 a 65535.

MediumInt: número entero con o sin signo. Con signo el rango de valores va desde -8.388.608 a 8.388.607. Sin signo el rango va desde 0 a 16777215.

Integer, Int: número entero con o sin signo. Con signo el rango de valores va desde -2147483648 a 2147483647. Sin signo el rango va desde 0 a 429.4967.295

BigInt: número entero con o sin signo. Con signo el rango de valores va desde -9.223.372.036.854.775.808 a 9.223.372.036.854.775.807. Sin signo el rango va desde 0 a 18.446.744.073.709.551.615.

Float: número pequeño en coma flotante de precisión simple. Los valores válidos van desde -3.402823466E+38 a -1.175494351E-38, 0 y desde 1.175494351E-38 a 3.402823466E+38.

xReal, Double: número en coma flotante de precisión doble. Los valores permitidos van desde -1.7976931348623157E+308 a -2.2250738585072014E-308, 0 y desde 2.2250738585072014E-308 a 1.7976931348623157E+308

Decimal, Dec, Numeric: Número en coma flotante desempaquetado. El número se almacena como una cadena

Tipo de Campo	Tamaño
TINYINT	1 byte
SMALLINT	2 bytes

MEDIUMINT	3 bytes
INT	4 bytes
INTEGER	4 bytes
BIGINT	8 bytes
FLOAT(X)	4 ú 8 bytes
FLOAT	4 bytes
DOUBLE	8 bytes
DOUBLE PRECISION	8 bytes
REAL	8 bytes
DECIMAL(M,D)	M+2 bytes sí D > 0, M+1 bytes sí D = 0
NUMERIC(M,D)	M+2 bytes if D > 0, M+1 bytes if D = 0

Tipos Fechas

A la hora de almacenar fechas, hay que tener en cuenta que MySQL no comprueba de una manera estricta si una fecha es válida o no. Simplemente comprueba que el mes está comprendido entre 0 y 12 y que el día está comprendido entre 0 y 31.

Date: tipo fecha, almacena una fecha. El rango de valores va desde el 1 de enero del 1001 al 31 de diciembre de 9999. El formato de almacenamiento es de año-mes-día

DateTime: Combinación de fecha y hora. El rango de valores va desde el 1 de enero del 1001 a las 0 horas, 0 minutos y 0 segundos al 31 de diciembre del 9999 a las 23 horas, 59 minutos y 59 segundos. El formato de almacenamiento es de año-mes-día horas:minutos:segundos

TimeStamp: Combinación de fecha y hora. El rango va desde el 1 de enero de 1970 al año 2037. El formato de almacenamiento depende del tamaño del campo:

Tamaño	Formato
14	AñoMesDiaHoraMinutoSegundo aaaammddhhmmss
12	AñoMesDiaHoraMinutoSegundo aammddhhmmss
8	AñoMesDia aaaammdd
6	AñoMesDia aammdd
4	AñoMes aamm
2	Año aa

Time: almacena una hora. El rango de horas va desde -838 horas, 59 minutos y 59 segundos a 838, 59 minutos y 59 segundos. El formato de almacenamiento es de 'HH:MM:SS'

Year: almacena un año. El rango de valores permitidos va desde el año 1901 al año 2155. El campo puede tener tamaño dos o tamaño 4 dependiendo de si queremos almacenar el año con dos o cuatro dígitos.

Tipo de Campo	Tamaño
DATE	3 bytes
DATETIME	8 bytes
TIMESTAMP	4 bytes
TIME	3 bytes
YEAR	1 byte

Tipos Cadena de Texto

Una cadena de caracteres es una secuencia ordenada de longitud arbitraria.

Char(n): almacena una cadena de longitud fija. La cadena podrá contener desde 0 a 255 caracteres.

VarChar(n): almacena una cadena de longitud variable. La cadena podrá contener desde 0 a 255 caracteres.

Dentro de los tipos de cadena se pueden distinguir otros dos subtipos, los tipo Text y los tipo BLOB (Binary large Object)

La diferencia entre un tipo y otro es el tratamiento que reciben a la hora de realizar ordenamientos y comparaciones. Mientras que el tipo text se ordena sin tener en cuenta las Mayúsculas y las minúsculas, el tipo BLOB se ordena teniéndolas en cuenta.

Los tipos BLOB se utilizan para almacenar datos binarios como pueden ser ficheros.

TinyText y TinyBlob: Columna con una longitud máxima de 255 caracteres.

Blob y Text: un texto con un máximo de 65535 caracteres.

MediumBlob y MediumText: un texto con un máximo de 16.777.215 caracteres.

LongBlob y LongText: un texto con un máximo de caracteres 4.294.967.295. Hay que tener en cuenta que debido a los protocolos de comunicación los paquetes pueden tener un máximo de 16 Mb.

Enum: campo que puede tener un único valor de una lista que se especifica. El tipo Enum

acepta hasta 65535 valores distintos

Set: un campo que puede contener ninguno, uno ó varios valores de una lista. La lista puede tener un máximo de 64 valores.

Tipo de campo	Tamaño de Almacenamiento
CHAR(n)	n bytes
VARCHAR(n)	n +1 bytes
TINYBLOB, TINYTEXT	Longitud+1 bytes
BLOB, TEXT	Longitud +2 bytes
MEDIUMBLOB, MEDIUMTEXT	Longitud +3 bytes
LOB, LONGTEXT	Longitud +4 bytes
ENUM('value1','value2',...)	1 ó dos bytes dependiendo del número de valores
SET('value1','value2',...)	1, 2, 3, 4 ó 8 bytes, dependiendo del número de valores

Diferencia de almacenamiento entre los tipos Char y VarChar

Valor	CHAR(4)	Almacenamiento	VARCHAR(4)	Almacenamiento
"	"	4 bytes	"	1 byte
'ab'	'ab '	4 bytes	'ab'	3 bytes
'abcd'	'abcd'	4 bytes	'abcd'	
'abcdefgh'	'abcd'	4 bytes	'abcd'	5 bytes

La mayor parte de estos tipos de datos se repiten para todas las bases de datos. Algunas comparten sólo una parte, y otras agregan nuevos tipos; pero conociendo toda esta información básica, seremos capaces de encontrar y aprovechar las diferencias.

ZEROFILL y UNSIGNED

ZEROFILL sirve para, en un campo numérico, rellenar con 0 (cero) a izquierda todos los espacios, hasta completar el tipo de dato.

Por ejemplo, supongamos que tenemos una tabla Productos con un campo llamado código. Este campo es de tipo INTEGER (4 bytes), pero nosotros queremos que el primer código sea 00001 y no simplemente 1; tal como sucede en las facturas que cada día recibimos en nuestras compras.

Simplemente tildando ZEROFILL, nuestro MySQL nos devolverá el campo con este formato, evitándonos los molestos procesos sobre el dato.

UNSIGNED se tildará en forma automática al indicar el campo ZEROFILL, aunque también podemos tildarlo en forma independiente. Este modificador indicará que ese campo numérico no podrá ser negativo; dándonos un rango positivo mucho más grande.

Por ejemplo, un INTEGER abarca los números del -2147483648 al 2147483647. En cambio, si le colocamos el atributo UNSIGNED, abarcará desde 0 hasta 4294967295.

Ejercicio

1. Crear una tabla de prueba
2. Crear un campo numérico, un campo numérico con UNSIGNED y otro con ZEROFILL.
3. Cargar los campos, de a uno, con un valor negativo, luego uno positivo y por último uno mayor a 2147483647.
4. Consultar todos los resultados y cotejar las diferencias.

Consultas relacionadas

No todas las consultas son el resultado de un SELECT simple, contra una sola tabla. Esto se da especialmente cuando normalicemos nuestra Base de Datos (más adelante en este apunte). Para solucionar estas consultas, cruzando información de distintas tablas, contamos con una serie de comandos llamados JOIN.

Primero vamos a mostrar el contenido de nuestras dos tablas: Puesto y Sector.

- Puestos

Puesto	idSector
Administracion	1
Administracion	2
Distribuidores	1
Minorista	2
Mayorista	NULL

- Sector

idSector	Sector
1	Ventas
2	Compras
3	Gerencia

JOIN o INNER JOIN

Inner Join es la intersección de dos conjuntos de datos (en este caso tablas). Es decir, al utilizarlo traeremos todos los datos que coincidan con nuestra condición de unión.

Por ejemplo, si deseamos traer a todos los empleados con sus puestos, haremos lo siguiente:

```
SELECT sector.sector, puestos.puesto
FROM sector INNER JOIN puestos ON sector.id = puestos.idSector;
```

Sector	Puesto
Ventas	Administracion
Ventas	Distribuidores
Compras	Administracion
Compras	Minorista

Para no tener que escribir todo el tiempo los nombres de las tablas, podemos usar otro tipo de alias. Así, la misma consulta, quedaría de la siguiente forma:

```
SELECT s.sector, p.puesto
FROM sector s INNER JOIN puestos p ON s.id = p.idSector;
```

LEFT JOIN o LEFT OUTER JOIN

Left Join nos permite juntar estos dos conjuntos, tal como Inner Join, pero con la diferencia que nos traerá todos los resultados de la tabla que coloquemos a la izquierda. Esto quedará más claro con un ejemplo:

```
SELECT s.sector, p.puesto
FROM sector s LEFT JOIN puestos p ON s.id = p.idSector;
```

Sector	Puesto
Ventas	Administracion
Ventas	Distribuidores
Compras	Administracion
Compras	Minorista
Gerencia	NULL

Notemos la diferencia en respuesta de si usáramos INNER JOIN. A pesar de que el Sector “Gerencia” no tiene un puesto asignado, este se representará en nuestros resultados. Esto es a causa del LEFT (izquierda) JOIN, que nos devolverá TODOS los datos de la tabla a la izquierda, tengan o no coincidencia con la tabla de la derecha.

RIGHT JOIN o RIGHT OUTER JOIN

Right Join funciona exactamente igual que Left Join, pero tomando todos los datos de la tabla a la derecha (right). Para esto podemos ver el siguiente ejemplo.

Primero, insertemos un puesto al cual no le asignaremos sector:

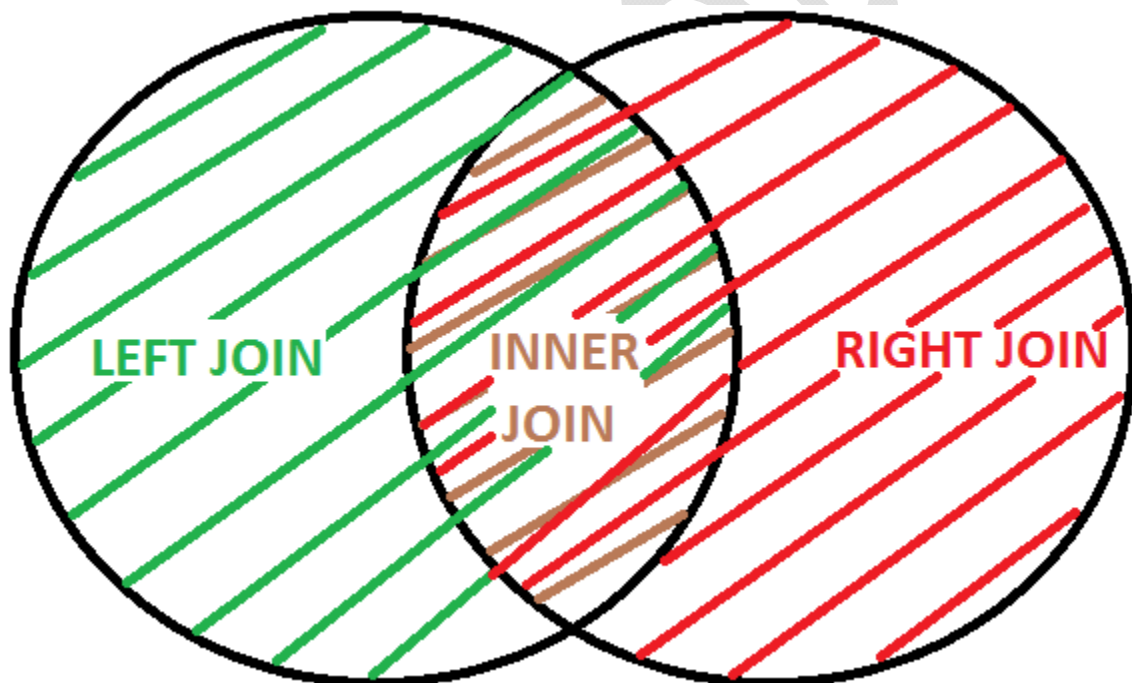
```
INSERT INTO puestos (puesto) ('Mayorista');
```

Ahora hagamos la misma consulta que antes, pero cambiando Left Join por Right Join, y veamos las diferencias:

```
SELECT s.sector, p.puesto
FROM sector s RIGHT JOIN puestos p ON s.id = p.idSector;
```

Sector	Puesto
Ventas	Administracion
Ventas	Distribuidores
Compras	Administracion
Compras	Minorista
NULL	Mayorista

Un pequeño esquema quizás nos aclare un poco más como son los 3 conjuntos:



JOINS con más de dos tablas

Si quisiéramos agregar más tablas, sólo debemos hacer lo siguiente:

```
SELECT pe.nombre, pe.apellido, s.sector, pu.puesto
FROM sector s INNER JOIN puestos pu ON s.id = pu.idSector
INNER JOIN personas pe ON pu.id = pe.idPuesto;
```

Así obtendremos todos los empleados (nombre y apellido), sumado su puesto y sector de trabajo.

También podemos combinarlo con otros Joins. Por ejemplo, si queremos todos los sectores, aun en el caso de que no tengan empleados:

```
SELECT pe.nombre, pe.apellido, s.sector, pu.puesto
FROM puestos pu INNER JOIN personas pe ON pu.id = pe.idPuesto
      RIGHT JOIN sector s ON s.id = pu.idSector;
```

Notemos que el orden de estos factores, puede alterar el producto.

Ejercicio:

1. Generar una consulta que nos diga cuantos empleados hay por puesto.
2. Agregar una persona, sin asignarle un puesto.
3. Listar todas las personas, con puesto y sector en el caso de corresponder.

Resolución:

1.

```
SELECT COUNT(pe.id), s.sector, pu.puesto
FROM puestos pu INNER JOIN personas pe ON pu.id = pe.idPuesto
      RIGHT JOIN sector s ON s.id = pu.idSector
GROUP BY s.id,pu.id;
```
2.

```
INSERT INTO personas (nombre,apellido,direccion,sueldo,fechaAlta)
VALUES ('Gonzalo','Martínez','Av Corrientes 1200',4500,CURRENT_DATE);
```
3.

```
SELECT pe.nombre, pe.apellido, s.sector, pu.puesto
FROM sector s INNER JOIN puestos pu ON s.id = pu.idSector
      RIGHT JOIN personas pe ON pu.id = pe.idPuesto;
```

UNION

Si lo que deseamos es traer todos los datos, de ambas tablas, tengan o no coincidencia, debemos utilizar Union:

```
SELECT s.sector, p.puesto
FROM sector s LEFT JOIN puestos p ON s.id = p.idSector
UNION
SELECT s.sector, p.puesto
FROM sector s RIGHT JOIN puestos p ON s.id = p.idSector;
```

Sector	Puesto
Ventas	Administracion
Ventas	Distribuidores
Compras	Administracion
Compras	Minorista
Gerencia	NULL
NULL	Mayorista

Funciones de Base de Datos

De agregado:

Son funciones que permiten obtener un resultado basado en los valores contenidos en una columna de una tabla, son funciones que mayormente se utilizarán en consultas de resumen (agrupadas mediante GROUP BY) ya que obtienen un “resumen” de los valores contenidos en las filas de la tabla. También se las conoce como funciones de agrupado.

Algunas de las funciones más utilizadas son:

MAX	Busca el valor máximo
MIN	Busca el valor mínimo
AVG	Saca el promedio de los valores, descartando nulos
SUM	Suma todos los valores
COUNT	Cuenta la cantidad de ocurrencias
STD	Retorna desviación estándar
STDDEV	Retorna desviación estándar, compatible con Oracle
VARIANCE	Retorna la varianza
CONCAT	Concatena cadenas de texto
LAST	Retorna el último valor
FIRST	Retorna el primer valor

Ejemplo:

```
SELECT COUNT(*)  
FROM personas;
```

Nos devolverá la cantidad de datos cargados en la tabla personas. Buscando un mejor rendimiento de nuestra consulta, podremos colocar el nombre de un campo en lugar del asterisco, por ejemplo COUNT(id), dándonos el mismo resultado. Preferentemente deberíamos utilizar el campo más pequeño en tamaño (bytes).

Ejemplo:

```
SELECT AVG(sueldo)  
FROM personas;
```

Nos retornará el promedio de sueldos de nuestra empresa.

Estas funciones también puede ser combinadas con todo lo que vimos anteriormente para el SELECT.

La sintaxis general de las funciones de agregado es:

funcion_agregado ([ALL | DISTINCT] expresión)

Ejercicio:

1. Contar la cantidad de personas que cobran más de 1000 de sueldo.
2. Seleccionar el sueldo máximo.
3. Calcular el promedio de sueldos, sin usar AVG().
4. Concatene en una consulta nombre y apellido.

Resolución:

1. SELECT COUNT(id) FROM personas WHERE sueldo > 1000;
2. SELECT max(sueldo) FROM personas;
3. SELECT SUM(sueldo)/COUNT(id) FROM personas;
4. SELECT CONCAT(nombre,' ',apellido) FROM personas;

Escalares

Una función escalar retorna un único valor, o un nulo. Hay de distinto tipo, entre las que se destacan las que podremos crear nosotros mismos.

Las funciones no están estandarizadas correctamente, haciendo que varíen en su implementación según el motor que estemos utilizando.

Si queremos crear nuestras propias funciones, debemos utilizar la instrucción "CREATE ROUTINE" de nuestro menú. Por ejemplo, uno de los accesos es:



La sintaxis básica es:

```

DELIMITER $$
CREATE FUNCTION nombre (
    parametro1 TIPO[=valorpordefecto1],
    [parametro2 TIPO =valorpordefecto2]
)
RETURNS tipo
BEGIN
    -- Instrucciones
    RETURN valor;
END $$
DELIMITER ;

```

DELIMITER era utilizado por los intérpretes MySQL para saber donde finalizaba nuestro bloque de código. Hoy por hoy su uso es dudoso.

Luego del nombre se colocan (opcionalmente) los parámetros de entrada con su tipo.

El valor por defecto no es obligatorio, y de colocarlo deberá ser una constante. Esto varía de motor en motor, haciendo que, por ejemplo, MySQL no lo soporte.

La cláusula "RETURNS" indica el tipo de dato retornado.

El cuerpo de la función se define en un bloque "BEGIN...END", que contiene las instrucciones que retornan el valor. El tipo del valor retornado puede ser de cualquier tipo, excepto TEXT, NTEXT, IMAGE, CURSOR o TIMESTAMP.

Ejemplo:

Crearemos una función que nos devuelva el sueldo más alto:

```

DELIMITER $$
DROP FUNCTION IF EXISTS `personal`.`mayorSueldo` $$
CREATE FUNCTION `personal`.`mayorSueldo` () RETURNS INT
BEGIN
    DECLARE respuesta INTEGER;
    SELECT MAX(sueldo) INTO respuesta FROM personas;
    RETURN respuesta;
END $$
DELIMITER ;

```

La cláusula DECLARE sirve para declarar una variable, la cual nos servirá, en este caso, para guardar el valor que deseamos retornar en nuestra función.

Con INTO guardamos el valor devuelto por el SELECT dentro de la variable que declaramos.

Luego, con RETURN, retornamos el valor obtenido.

Lo podremos utilizar de la siguiente forma:

```
SELECT mayorSueldo();
```

Ejercicio:

1. Crear una nueva función. Esta recibirá un ID de persona y nos retornará el puesto en el que trabaja.
2. Utilizarla en una consulta SELECT.

Resolución:

1.

```
DELIMITER $$  
DROP FUNCTION IF EXISTS `personal`.`puestoPersona` $$  
CREATE FUNCTION `personal`.`puestoPersona` (p_id INTEGER) RETURNS VARCHAR(45)  
BEGIN  
    DECLARE respuesta VARCHAR(45);  
  
    SELECT Pu.puesto INTO respuesta  
    FROM personas Pe INNER JOIN puestos Pu ON Pe.idPuesto = Pu.id  
    WHERE Pe.id = p_id;  
  
    RETURN respuesta;  
END $$  
DELIMITER ;
```
2.

```
SELECT puestoPersona();
```

También tenemos las propias del sistema, que variarán, muy probablemente, de motor en motor. Estas son funciones propias, que nos proveen servicios tales como información del sistema, funciones matemáticas, numéricas, etc.. Las podemos encontrar divididas en 4 grupos básicos, según lo que retornan:

- Del sistema
- Fecha y hora
- Numéricas
- Cadena de caracteres

Del sistema:

Este tipo de función escalar nos provee información del servidor donde está corriendo nuestro motor de base de datos. Nos dará información del usuario y de fecha y hora, fundamentalmente. Algunas de las más destacadas en MySQL son:

Función	Uso
CURRENT_DATE	Fecha actual del sistema
CURRENT_TIME	Hora actual del sistema
CURRENT_TIMESTAMP	Fecha y hora actual del sistema
CURRENT_USER	Identifica al usuario actual conectado a la base de datos
SESSION_USER	Identifica el Authorization ID, si es que difiere del usuario actual
SYSTEM_USER	Identifica al usuario active en el sistema actualmente

Para consultarlas, simplemente podemos llamarlas en un SELECT del siguiente modo:

```
SELECT CURRENT_TIMESTAMP();
```

Ejercicio:

1. Agregar a la tabla personas el campo fecha de alta, del tipo DATE.
2. Cargar los datos de fecha de todos nuestros empleados.
3. Seleccionar sólo aquellos dados de alta la desde el 1 de septiembre hasta hoy.

Resolución

1. ALTER TABLE `personal`.`personas` ADD COLUMN `fechaAlta` DATE NOT NULL DEFAULT 0 AFTER `idPuesto`;
2. Carga manual
3. SELECT * FROM personal.personas p WHERE fechaAlta BETWEEN '2012-09-01' AND CURRENT_DATE();

Fecha y hora

Las funciones de fecha y hora nos darán la forma de modificar, consultar y operar sobre estos tipos de datos. Son extremadamente útiles, tanto para consultar como para operar.

Podemos encontrar todas las funciones en el sitio oficial para desarrolladores de MySQL. Actualmente el link es: <http://dev.mysql.com/doc/refman/4.1/en/date-and-time-functions.html>.

Algunas de las más interesantes son:

Función	Uso
ADDDATE()	Agrega días a la fecha

ADDTIME()	Agrega tiempo a la fecha
CONVERT_TZ()	Cambia el Time Zone de la fecha
EXTRACT(datetime_expression FROM expresión)	Permite extraer una parte de una fecha (año, mes, día, hora, minutos, segundos).
DATE_FORMAT()	Cambiamos el formato de la fecha
LAST_DAY(mes)	Retorna el último día del mes que le indicamos
NOW()	Nos retornará la fecha y hora actual
TIMEDIFF()	Diferencia entre dos fechas, en formato de tiempo
DATEDIFF()	Diferencia entre dos fechas, en formato de días

NOW() es la función más sencilla y que más utilizaremos, ya que nos devuelve la fecha actual del sistema. Es tan fácil como esto:

```
SELECT NOW();
```

Y nos retornará la fecha completa, en este formato:

```
2013-01-30 16:10:30
```

ADDDATE nos servirá para sumarle a una fecha un intervalo dado de días:

```
SELECT fechaAlta, DATE_ADD(fechaAlta, INTERVAL 31 DAY) FROM personas;
```

fechaAlta	DATE_ADD
2012-01-01	2012-02-01
2012-02-02	2012-03-04
2012-03-03	2012-04-03
2012-09-05	2012-10-06
2012-09-11	2012-10-12
2011-12-01	2012-01-01
2012-10-05	2012-11-05

ADDTIME hará lo mismo, pero sumándole intervalos de tiempo. Por ejemplo, para agregar 2 centésimas de segundo podríamos hacer lo siguiente:

```
SELECT ADDTIME('2012-12-31 23:59:59.9', '0 0:0:0.2');
```

La zona horaria (Time Zone) podremos cambiarla de dos formas. La primera, indicando la zona a través de sus siglas:

```
SELECT CONVERT_TZ('2012-01-01 12:00:00','GMT','US/EASTERN');
```

La segunda opción funciona exactamente igual a la anterior, sólo que en vez de poner las siglas, utilizaremos simplemente la zona horaria:

```
SELECT CONVERT_TZ('2012-01-01 12:00:00','+00:00','-03:00');
```

La fecha en MySQL, por defecto, se representa como AÑO-MES-DIA HORA:MINUTOS:SEGUNDOS. Con la función DATE_FORMAT() podremos variar esto al formato que más nos guste. Por ejemplo, podremos colocar DIA-MES-AÑO de la siguiente forma:

```
SELECT DATE_FORMAT('2012-09-12 22:23:00', '%d-%m-%Y');
```

Dando como resultado:

12-09-2012

La tabla completa de los modificadores de formato es la siguiente:

Sigla	Descripción
%a	Abrevia el nombre del día (Dom..Sab)
%b	Abrevia el nombre del mes (Ene..Dic)
%c	Mes en formato numérico (0..12)
%m	Mes en formato numérico (00..12)
%d o %e	Día en formato numérico (00..31)
%f	Microsegundos (000000..999999)
%H	Hora formato 24hs (00..23)
%h o %l	Hora formato 12hs (01..12)
%k	Hora en formato 24hs (0..23)
%l	Hora formato 12hs (1..12)
%i	Minutos en formato numérico (00..59)
%j	Día del año (001..366)
%M	Mes, nombre completo (Enero..Diciembre)
%p	AM o PM
%r	Tiempo en formato 12hs (hh:mm:ss seguido por AM o PM)
%S o %s	Segundos (00..59)
%T	Tiempo en formato 24hs (hh:mm:ss)
%U	Semanas (00..53), donde el domingo es el primer día de la semana
%u	Semanas (00..53), donde el lunes es el primer día de la semana
%V	Semanas (01..53), donde el domingo es el primer día de la semana; usado con %X
%v	Semanas (01..53), donde el lunes es el primer día de la semana; usado con %x
%W	Nombre del día de la semana completo (Domingo..Sábado)
%w	Día de la semana, numérico (0=Domingo..6=Sábado)
%X	Año para la semana, donde el domingo es el primer día de la semana, en formato numérico de 4 dígitos; usado con %V

%x	Año para la semana, donde el lunes es el primer día de la semana, en formato numérico de 4 dígitos; usado con %v
%Y	Año en formato numérico de 4 dígitos
%y	Año en formato numérico de 2 dígitos

Tabla extraída de http://dev.mysql.com/doc/refman/4.1/en/date-and-time-functions.html#function_date-format.

El formato de EXTRACT, el cual permite extraer una parte de una fecha, sería el siguiente:

```
SELECT EXTRACT(datetime_expresion FROM campo_fecha) FROM tabla;
```

Siendo un ejemplo de su uso:

```
SELECT fechaAlta, EXTRACT(YEAR FROM fechaAlta) FROM personal.personas;
```

Retornándonos en el ejemplo el año de la fecha consultada:

fechaAlta	EXTRACT
2012-01-01	2012
2012-02-02	2012
2012-03-03	2012
2012-09-05	2012
2012-09-11	2012
2011-12-01	2011
2012-10-05	2012

Siendo algunos ejemplos de otros de sus datetime_expresion los valores:

- YEAR_MONTH
- DAY_SECOND
- DAY_MINUTE
- DAY_HOUR
- HOUR_MINUTE
- HOUR_SECOND
- YEAR
- MONTH
- DAY
- HOUR
- MINUTE
- SECOND
- MINUTE_SECOND

- MICROSECOND

Con la función LAST_DAY podremos saber cuál es el último día del mes para una fecha en particular. Por ejemplo:

```
SELECT fechaAlta , LAST_DAY (fechaAlta) FROM personas;
```

fechaAlta	LAST_DAY
2012-01-01	2012-01-31
2012-02-02	2012-02-29
2012-03-03	2012-03-31
2012-09-05	2012-09-30
2012-09-11	2012-09-30
2011-12-01	2011-12-31
2012-10-05	2012-10-31

Por último, veremos DATEDIFF(). Esta función calculará la diferencia entre dos fechas, retornando un resultado en el formato de días:

```
SELECT Now(),fechaAlta,DATEDIFF(Now(),fechaAlta) FROM personas;
```

Now	fechaAlta	DATEDIFF
2013-01-30 16:19:09	2012-01-01 00:00:00	395
2013-01-30 16:19:09	2012-02-02 00:00:00	363
2013-01-30 16:19:09	2012-03-03 00:00:00	333
2013-01-30 16:19:09	2012-09-05 00:00:00	147
2013-01-30 16:19:09	2012-09-11 00:00:00	141
2013-01-30 16:19:09	2011-12-01 00:00:00	426
2013-01-30 16:19:09	2012-10-05 00:00:00	117

Numéricas

Este tipo de función nos retornará un valor numérico sobre distintos tipos de expresiones. Nos servirán para contar bits, bytes, extraer secciones de una fecha, etc.. Las más utilizadas son:

Función	Uso
BIT_LENGTH(expression)	Retorna un entero con la cantidad de bits de la expresión
CHAR_LENGTH(expression)	Retorna un entero con la cantidad de caracteres de la expresión

OCTET_LENGTH(expression)	Retorna un entero con la cantidad de octetos de la expresión. El valor es igual a BIT_LENGTH/8.
POSITION(starting_string IN search_string)	Retorna un entero con la posición de donde comienza la expresión buscada.

Ejercicio:

1. Consultar todos los empleados dados de alta en 2012.
2. Consultar todos los empleados dados de alta en los últimos 7 días, sin importar en que día estamos hoy.

Resolución

1. `SELECT * FROM personas WHERE EXTRACT(YEAR FROM fechaAlta) = 2012;`
2. `SELECT * FROM personas WHERE fechaAlta BETWEEN ADDTIME(CURRENT_TIMESTAMP(), '-7 0:0:0.0') AND CURRENT_DATE();`

Cadena de caracteres

Las funciones de cadena de carácter afectarán a cadenas y retornarán el mismo tipo de dato. Nos sirven para concatenar expresiones, extraer partes, pasar a mayúsculas o minúsculas, etc.. Por ejemplo, contaremos con las siguientes:

Función	Uso
CONCAT ó CONCATENATE (expresión expresión)	Concatena 2 o más expresiones (columnas, literales o variables), retornando sólo una.
TRANSLATE	Convierte una cadena de un charset a otro
LOWER	Pasa una cadena de carácter a minúsculas
UPPER	Pasa una cadena de carácter a mayúsculas
TRIM	Quita los espacios en blanco a izquierda y derecha
SUBSTRING(expresión,inicio,fin)	Extrae una porción de una cadena
LOCATE (expresión, columna)	Retorna la primera ocurrencia de la subcadena expresión en la cadena columna

La función CONCAT es simple de usar, y muy útil:

```
SELECT CONCAT(nombre,apellido) FROM personas;
```

Tan útil como la anterior, encontramos a SUBSTRING permitiéndonos arrancar pedazos de una cadena:

```
SELECT SUBSTRING(nombre,1,3) FROM personas;
```

LOCATE también es sencillo de utilizar:

```
SELECT LOCATE('ez', apellido) FROM personas;
```

Ejercicio:

1. Seleccionar todos los nombres en mayúsculas y los apellidos en minúsculas.
2. Consultar nombre, apellido y sueldo de las personas con el siguiente formato: "Apellido, Nombre: \$sueldo".
3. Seleccionar el apellido de los empleados, indicando donde tiene la primera vocal 'a'.

Resolución

1.

```
SELECT LOWER(nombre),UPPER(apellido) FROM personas;
```
2.

```
SELECT CONCAT(apellido,', ',nombre,': $',sueldo) FROM personas;
```

3. `SELECT apellido,LOCATE('a',apellido) FROM personas;`

Todas las funciones vistas, y muchas más, se encuentran documentadas en <http://dev.mysql.com>.

Agrupaciones

Group By

Cuando vimos las funciones de agregado o agrupado faltó algo fundamental, el Group By. Esta sentencia sirve para agrupar resultados de nuestra consulta. La sentencia se diagramaría de la siguiente forma:

```
SELECT campos
FROM tabla
[WHERE condicion]
GROUP BY campoAgrupacion
[HAVING condicionAgrupamiento]
```

Por ejemplo, podríamos, en nuestra tabla de ejemplo, agrupar por puesto y sacar el sueldo promedio de cada uno:

```
SELECT pu.puesto,avg(pe.sueldo)
FROM personas pe INNER JOIN puestos pu ON pe.idPuesto=pu.id
GROUP BY pu.id;
```

Cuando mostramos la estructura del GROUP BY apareció una clausula HAVING que nunca habíamos visto. Ésta clausula nos proveerá un filtro para nuestra condición de agrupamiento. Por ejemplo, a la misma consulta que hicimos antes, le agregaremos un filtro para que sólo muestre los promedios mayores a 1000. Para esto debemos utilizar alias:

```
SELECT pu.puesto,avg(pe.sueldo) AS promedioSueldo
FROM personas pe INNER JOIN puestos pu ON pe.idPuesto=pu.id
GROUP BY pu.id
HAVING promedioSueldo > 1000;
```

Inclusive podríamos operar sobre una nueva operación o función:

```
SELECT pu.puesto,avg(pe.sueldo) AS promedioSueldo
FROM personas pe INNER JOIN puestos pu ON pe.idPuesto=pu.id
GROUP BY pu.id
```

HAVING promedioSueldo > 1000;

A pesar de que los alias nos son inútiles en el WHERE, dándonos un error de compilación; nos son de gran utilidad en la cláusula HAVING. Esto es por el orden en que se ejecuta una consulta:

1. JOIN - Junta los conjuntos
2. WHERE - Filtra las consultas
3. GROUP BY - Agrupa los resultados
4. HAVING - Filtra los resultados agrupados
5. ORDER BY - Ordena los datos

En este orden ejecutará nuestro motor una consulta una sentencia. Por esto es más óptimo utilizar JOIN por ejemplo, y para esto también nos ayudará el HAVING.

Esto no se debería hacer:

Existe una forma de no utilizar JOINS, pero está pésimo. Es poco performante, y no aprovecha las virtudes de SQL. Algunos motores se encargarán de optimizar estas consultas mal redactadas, otros no, provocando ralentizaciones innecesarias. La forma es:

```
SELECT nombre,apellido,puesto
FROM personas pe,puestos pu
WHERE pe.idPuesto=pu.id;
```

Esto traerá ambos conjuntos (tablas) y luego aplicará el filtro sobre este nuevo subconjunto. Es mucho más veloz hacerlo mediante JOIN y luego aplicar filtros sobre ese subconjunto reducido (en la mayoría de los casos de consultas complejas, si está bien hecho, bastante más reducido).

Ejercicio:

1. Traer para cada puesto, nombre, apellido y sueldo de quien más gane.
2. Consultar a quienes ganen menos de cada puesto, y su apellido termine con 'z'.
3. Consultar sólo al primero de cada puesto que gane por encima del promedio, hayan ingresado hoy y tengan una 'e' en su dirección.

Resolución:

1. *SELECT idPuesto,MAX(sueldo),nombre,Apellido*
FROM personas
GROUP BY idPuesto

2. *SELECT idPuesto,MIN(sueldo),nombre,Apellido*
FROM personas
WHERE apellido like '%z'
GROUP BY idPuesto
3. *SELECT idPuesto,nombre,apellido,sueldo*
FROM personas
WHERE direccion like '%e%'
GROUP BY idPuesto
HAVING sueldo > AVG(sueldo)

BORRADOR

Normalización de Base de datos

El proceso de normalización de bases de datos consiste en aplicar una serie de reglas para:

- Evitar la redundancia de los datos.
- Evitar problemas de actualización de los datos en las tablas.
- Proteger la integridad de los datos.

Existen tres formas normales:

1. Primera forma normal: elimina los valores repetidos de una BD.
 - a) Todos los atributos son atómicos. Un atributo es atómico si los elementos del dominio son indivisibles, mínimos. En otras palabras, no debemos tener un atributo llamado nombreYApellido.
 - b) La tabla contiene una PK única, que distingue a ese registro de los otros.
 - c) La PK no contiene atributos nulos, no permite vacíos.
 - d) No debe existir variación en el número de columnas de los registros.
2. Segunda forma normal: todos los atributos que no son clave principal deben depender únicamente de la clave principal (dependencia funcional). Por ejemplo, no tener datos del puesto en nuestra entidad Empleado.
 - a) Se encuentra en 1FN (Primera Forma Normal).
 - b) Todos sus atributos que no son de la clave principal tienen dependencia funcional completa respecto de todas las claves existentes en el esquema. En otras palabras, para determinar cada atributo no clave se necesita la clave primaria completa, no sólo una subclave. Por ejemplo, en caso de tener una PK compuesta por dos o más atributos, se debe hacer referencia a la totalidad de estos atributos.
 - c) La 2FN se aplica a las relaciones que tienen claves primarias compuestas por dos o más atributos. Si una relación está en 1FN y su clave primaria es simple (tiene un solo atributo), entonces también está en 2FN.
3. Tercera forma normal: Elimina cualquier dependencia transitiva. Una dependencia transitiva es aquella en la cual las columnas que no son llave son dependientes de otras columnas que tampoco son llave.
 - a) Se encuentra en 2FN.
 - b) Cada atributo que no está incluido en la clave primaria no depende transitivamente de la clave primaria.

Consideraciones para el modelado lógico de la Base de Datos

Comprender el marco de aplicación del negocio del cliente es fundamental para poder generar un modelado correcto.

Debemos cumplir con las 3 formas de normalización vistas, para hacer un modelo correcto de una dinámica apropiada; aunque a veces se debe romper algunas reglas para llegar a una performance más alta. En otras palabras, la rigurosidad del modelo dependerá muchas veces de la aplicación.

Si la pregunta es hasta donde debemos llegar, esta es una ciencia subjetiva. Para una base de datos de bajo nivel de consulta, para un solo cliente, quizás sea exagerado cumplimentar las 3 formas; pero posiblemente no esté de más, contemplando una posible escalada de este sistema (quizás no para este sistema, sino para otro similar de una empresa más grande).

Ejemplo:

Un dato sin normalizar no cumple con ninguna regla de normalización. Para explicar con un ejemplo en qué consiste cada una de las reglas, vamos a considerar los datos de la siguiente tabla.

ID_ORDEN	FECHA	ID_CLIENTE	NOM_CLIENTE	ESTADO	NUM_ITEM	DESC_ITEM	CANT	PRECIO
2301	23/2/03	101	MARTI	CA	3786	RED	3	35
2301	23/2/03	101	MARTI	CA	4011	RAQUETA	6	65
2301	23/2/03	101	MARTI	CA	9132	PAQ-3	8	4.75
2302	25/2/03	107	HERMAN	WI	5794	PAQ-6	4	5
2303	27/2/03	110	WE-SPORTS	MI	4011	RAQUETA	2	65
2303	27/2/03	110	WE-SPORTS	MI	3141	FUNDA	2	10

Si analizamos estos registros veremos que no cumplen siquiera con la 1FN. Las columnas NUM_ITEM, DESC_ITEM, CANT y PRECIO tienen datos repetidos. Para solucionarlo deberemos realizar dos pasos:

- Tenemos que eliminar los grupos repetidos.
- Tenemos que crear una nueva tabla con la PK de la tabla base y el grupo repetido.

Los registros quedan ahora conformados en dos tablas que llamaremos ORDENES y ARTICULOS_ORDENES:

ORDENES:

ID_ORDEN	FECHA	ID_CLIENTE	NOM_CLIENTE	ESTADO
2301	23/2/03	101	MARTI	CA
2302	25/2/03	107	HERMAN	WI
2303	27/2/03	110	WE-SPORTS	MI

ARTICULOS_ORDENES:

ID_ORDEN	NUM_ITEM	DESC_ITEM	CANT	PRECIO
2301	3786	RED	3	35
2301	4011	RAQUETA	6	65
2301	9132	PAQ-3	8	4.75
2302	5794	PAQ-6	4	5
2303	4011	RAQUETA	2	65
2303	3141	FUNDA	2	10

Ahora que nuestra base de datos cumple con la 1FN, podemos llevarla a 2FN. Para esto debemos realizar los siguientes pasos:

- Determinar cuáles columnas que no son llave no dependen de la llave primaria de la tabla.
- Eliminar esas columnas de la tabla base.
- Crear una segunda tabla con esas columnas y la(s) columna(s) de la PK de la cual dependen.

La tabla ORDENES está en 2FN. Cualquier valor único de ID_ORDEN determina un sólo valor para cada columna. Por lo tanto, todas las columnas son dependientes de la llave primaria ID_ORDEN.

Por su parte, la tabla ARTICULOS_ORDENES no se encuentra en 2FN ya que las columnas PRECIO y DESC_ITEM son dependientes de NUM_ITEM, pero no son dependientes de ID_ORDEN. Por esto, a continuación, eliminaremos algunas columnas de dicha tabla para pasarlas a una nueva: ARTICULOS.

ARTICULOS_ORDENES:

ID_ORDEN	NUM_ITEM	CANT
2301	3786	3
2301	4011	6
2301	9132	8
2302	5794	4
2303	4011	2
2303	3141	2

ARTICULOS:

NUM_ITEM	DESC_ITEM	PRECIO
3786	RED	35
4011	RAQUETA	65
9132	PAQ-3	4.75
5794	PAQ-6	5
4011	RAQUETA	65
3141	FUNDA	10

Ya normalizamos nuestra tabla a 2FN. La tercera forma normal nos dice que tenemos que eliminar cualquier columna no llave que sea dependiente de otra columna no llave. Los pasos a seguir son:

- Determinar las columnas que son dependientes de otra columna no llave.
- Eliminar esas columnas de la tabla base.
- Crear una segunda tabla con esas columnas y con la columna no llave de la cual son dependientes.

Al observar las tablas que hemos creado, nos damos cuenta que tanto la tabla ARTICULOS, como la tabla ARTICULOS_ORDENES se encuentran en 3FN. Sin embargo la tabla ORDENES no lo está, ya que NOM_CLIENTE y ESTADO son dependientes de ID_CLIENTE, y esta columna no es la llave primaria.

Para normalizar esta tabla, moveremos las columnas no llave y la columna llave de la cual dependen dentro de una nueva tabla CLIENTES.

ORDENES:

ID_ORDEN	FECHA	ID_CLIENTE
2301	23/2/03	101
2302	25/2/03	107
2303	27/2/03	110

CLIENTES:

ID_CLIENTE	NOM_CLIENTE	ESTADO
101	MARTI	CA
107	HERMAN	WI
110	WE-SPORTS	MI

Ahora si, al fin, nuestra base de datos cumple con las 3 formas normales.

**FALTAN EJEMPLO DE NO
CUMPLIMIENTO DE LAS 3 FORMAS**

Sub Query

¿Qué son las Sub Query?

Las subconsultas son SELECT utilizados dentro de otro SELECT, o también de un INSERT, UPDATE o DELETE. Este tipo de consultas son mayormente utilizadas para reportes complejos. Se pueden utilizar en cualquier parte de la consulta.

Entendiendo el principio de funcionamiento

Para que entendamos su funcionamiento, comencemos por lo más simple:

```
SELECT (SELECT 'Hola mundo!');
```

El nivel de anidamiento no tiene límite, pasando las respuestas de un SELECT a otro:

```
SELECT (SELECT (SELECT 'Hola mundo!'));
```

Usos prácticos:

- *Filtros:*

Imaginemos que deseamos saber cuáles de nuestros empleados perciben salarios por encima de la media:

```
SELECT nombre, apellido, sueldo  
FROM personas  
WHERE sueldo >= (SELECT AVG(sueldo) FROM personas);
```

Hemos utilizado nuestra subconsulta para conseguir el valor promedio de sueldo y compararlo con el sueldo de cada empleado.

Si deseáramos saber cuáles son los empleados que tienen el sueldo más grande, nuestra consulta sería:

```
SELECT nombre, apellido, sueldo  
FROM personas  
WHERE sueldo >= (SELECT MAX(sueldo) FROM personas);
```

Llegado este punto algunos dirán: usemos ORDER BY y LIMIT y vamos a llegar al mismo resultado; pero se equivocan. Las ventajas de las subconsultas para estos casos son:

- LIMIT no es parte del estándar, por ende no lo encontraremos en todos los motores.

- Si hubiera más de una persona con el sueldo máximo, LIMIT sólo nos traería una según el criterio de ordenamiento, pudiendo negarnos información importante.

- *Tablas*

Cuando el motor procesa la consulta, pretende que el resultado de nuestra subconsulta es una tabla.

Para entender el concepto, veamos este ejemplo poco útil:

```
SELECT Tabla1.dato
FROM (SELECT 'Hola' AS dato) as Tabla1;
```

Es inevitable cuando nuestra Sub Query es parte del FROM que la renombremos con un alias, sino dará error.

Un uso más real podría ser saber cuánta gente cobra el sueldo más alto en nuestro sistema:

```
SELECT MAX(tbl.nr) AS nr
FROM
(
    SELECT sueldo, COUNT(sueldo) AS nr
    FROM personas
    GROUP BY sueldo
) AS tbl;
```

Para quienes aun no estén convencidos de utilizar Sub Querys, vamos a probar algo más complejo.

Primero, carguemos más datos en nuestra base de datos:

```
INSERT INTO personas (nombre,apellido,direccion,sueldo,idPuesto,fechaAlta)
VALUES ('Gustavo','Dominguez','Punta Arena 1523',3000,1,now()),
      ('Julio','Gutierrez','Bahía Arena 12',5000,2,now()),
      ('Daniel','Sanchez','Bahía Blanca 152',2000,3,now()),
      ('Carla','Mirinda','Salta 525',7000,1,now()),
      ('Daniela','Pérez','Catamarca 8152',2000,4,now()),
      ('Diego','Díaz','Ciudad de la Paz 5192',7500,5,now()),
      ('Darío','González','Yrigoyen 242',2700,3,now()),
      ('Julia','Martínez','Bransen 2222',3200,5,now()),
      ('Josefina','Brown','Alberdi 754',7100,2,now()),
      ('Eustaquio','San Juan','San Martín 152',2000,4,now()),
      ('Evangeline','Pancracio','San Juan 215',4600,2,now());
```

Ejercicio:

1.

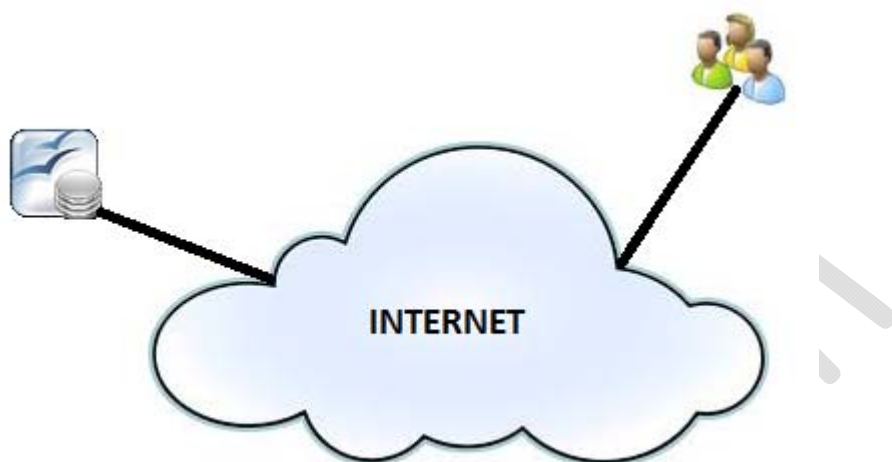
Resolución:

1.

BORRADOR

Stored Procedureⁱⁱ

Llamamos Stored Procedure (procedimientos almacenados) a un programa (o procedimiento) almacenado en la base de datos. Una de las grandes ventajas de los Stored Procedures es que correrán en el servidor y no en la máquina del usuario. Esto se da ya que usualmente nuestro esquema de trabajo es el siguiente:

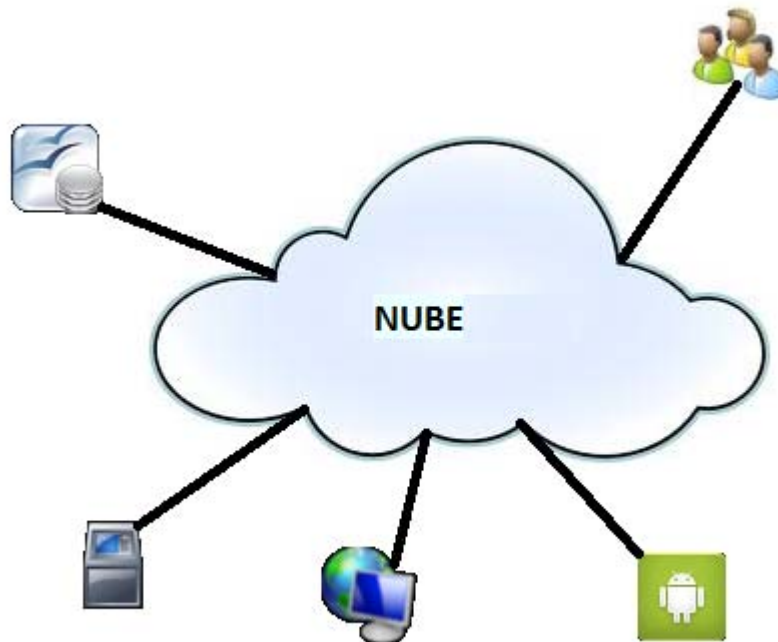


Siendo que el usuario no utilizará la misma máquina donde se encuentra corriendo nuestro motor de base de datos.

Muchas veces estos servidores serán grandes máquinas, con un rendimiento por encima del promedio, evitando que sea la máquina del usuario la que procese grandes cantidades de información, y evitando un innecesario ida y vuelta de información; ya que dentro del mismo procedimiento podremos validar información, hacer consultas, establecer transacciones (que ya veremos); encapsulando procesos complejos en un único lugar.

También facilitan la tarea al tener una base de datos consultada desde diferentes servicios, quizás hasta programados en diferentes lenguajes, que necesitan los mismos datos o realizar las mismas acciones.

Por ejemplo, imaginemos un banco. Las mismas tareas podrán ser realizadas desde los cajeros automáticos, empleados en sus puestos de atención al cliente, internet a través de sus sitios y hasta desde aplicaciones nativas en los distintos sistemas operativos de los celulares. Sería de gran utilidad que muchas funciones se encuentren directamente en el servidor y no tener que rehacer el código por cada lenguaje de programación utilizado para las diferentes aplicaciones.



Una de sus desventajas es que estos generalmente variarán en su implementación según que motor estemos utilizando.

Un Stored Procedure puede devolver un valor único, una tabla, o simplemente realizar una operación sin retornar ningún valor, si así lo deseáramos.

Nosotros veremos su implementación en MySQL.

Para crear un Stored Procedure, nuevamente iremos al ícono de CREATE ROUTINE, tal como en las funciones:



Por defecto, nuestro editor elije crear un Stored, cuya sintaxis básica es la siguiente:

```
DELIMITER $$
CREATE PROCEDURE [nombre_base].[nombre_stored]
(
    -- Variables de entrada
    [IN parametro1 TIPO],
    [IN parametro2 TIPO]
```

```

)
BEGIN
    -- CODIGO
    SELECT 'Hola Mundo!';
END

```

Verán, a lo largo de los años de trabajo, casos de procedimientos que incorporan en su cabecera (por debajo de los parámetros de entrada y por encima del BEGIN) datos como los siguientes:

```

COMMENT 'Comentario'
LANGUAGE SQL
DETERMINISTIC
SQL SECURITY DEFINER

```

COMMENT: un comentario, común y corriente, con fines de documentación.

LANGUAGE: con fines de portabilidad de la base de datos; por defecto se colocará SQL.

DETERMINISTIC: si un procedimiento devolverá SIEMPRE el mismo resultado ante la misma entrada, se considera determinístico. Esto se utiliza con fines de replicación y bitácora (log). Por defecto será NOT DETERMINISTIC.

SQL SECURITY: al momento de llamar un procedimiento chequeará los privilegios del usuario. INVOKER verificará los permisos de quien llama al Stored; DEFINER los del creador. Por defecto será DEFINER.

Estos datos por lo general no se escriben, utilizándose los valores predeterminados.

Llamar a un procedimiento es muy sencillo:

```
CALL nombre_stored ([parametro1], [parametro2]);
```

Para modificar un procedimiento almacenado, cambiaremos en su definición, que vimos hace instantes, CREATE por ALTER.

Para borrar un procedimiento creado, utilizaremos DROP:

```
DROP PROCEDURE IF EXISTS p2;
```

IF EXISTS nos previene del error ante la inexistencia de dicho Stored.

Tipos de parámetros

Los parámetros podrán ser declarados de varias formas, según su uso:

- IN: parámetro de entrada. Si no se coloca nada, serán IN por defecto.
- OUT: parámetro de salida.
- INOUT: parámetro de entrada y salida.

Con estos simples conocimientos, intentemos crear 3 Stored Procedure diferentes: uno con parámetros IN, otro con OUT y otro con INOUT.

Para cargar con un valor a un parámetro de entrada (OUT/INOUT), debemos utilizar el comando SET:

SET variable = valor;

Ejercicio – Parte 1:

1. Crear un Stored Procedure que reciba una variable de texto y la muestre mediante un SELECT.
2. Crear otro que reciba una variable OUT de tipo DATETIME. Llamémoslo proc_OUT. Carguemos la variable con la fecha actual del sistema.
3. Crear un último procedimiento que reciba un INTEGER de tipo INOUT. Llamémoslo proc_INOUT. Modifiquemos nuestra variable multiplicándola por 5.

Resolución:

1. DELIMITER \$\$
CREATE PROCEDURE proc_IN
(
 IN par VARCHAR(100)
)
BEGIN
 SELECT par;
END\$\$
2. DELIMITER \$\$
CREATE PROCEDURE proc_OUT
(
 OUT par DATETIME
)
BEGIN
 SET par = NOW();


```
END$$
```

```
3. DELIMITER $$  
CREATE PROCEDURE proc_INOUT  
(  
    INOUT par INTEGER  
)  
BEGIN  
    SET par = par*5;  
END$$
```

Bien, ya vimos como funciona un parámetro del tipo IN, lo cual es muy sencillo. El caso de los parámetros OUT e INOUT es distinto.

Comencemos con el OUT. Su sintaxis será:

```
CALL nombre_procedimiento(@variable1[, @variable2]);  
SELECT @variable1[,@variable2];
```

El valor no será impreso por el Stored Procedure, sino que será cargado dentro de nuestra variable, la cual nosotros deberemos imprimir a mano. Esto nos permitirá crear una especie de funciones con múltiples valores de retorno.

El caso del INOUT agregará sólo una complejidad: cargarle un valor a nuestra variable antes de pasársela al procedimiento:

```
SET @variable1 = 1;  
CALL nombre_procedimiento(@variable1[, @variable2]);  
SELECT @variable1[,@variable2];
```

Ejercicio – Parte 2:

4. Llamar al procedimiento IN
5. Llamar al procedimiento OUT
6. Llamar al procedimiento INOUT

Resolución:

5.

```
CALL proc_IN('Hola a todos.');
```
6.

```
CALL proc_OUT(@variable);  
SELECT @variable;
```

```
7. SET @variable = 1;
CALL proc_INOUT(@variable);
SELECT @variable;
```

En la declaración del nuestro Stored podremos encontrarnos con lo siguiente

```
CREATE DEFINER='root'@'localhost' PROCEDURE `divisiones`
```

Siendo el DEFINER el usuario quién lo creo y en que servidor en que nos encontramos. No es necesario escribirlo cuando creamos o modificamos ningún Procedimiento ni Función.

Variables

En el bloque de código delimitado por las clausulas BEGIN/END podremos declarar nuestras propias variables, y usarlas para distintos propósitos.

La nomenclatura es la siguiente:

```
DECLARE variable TIPO [DEFAULT valorPorDefecto];
```

Por ejemplo:

```
DECLARE múltiplo,divisor DEFAULT 5;
```

```
DECLARE cadena VARCHAR(100);
```

```
DECLARE fecha TIMESTAMP DEFAULT CURRENT_DATE;
```

Trabajemos un poco con variables:

```
DELIMITER $$
```

```
CREATE PROCEDURE miNombreEs (IN nombre VARCHAR(50))
```

```
BEGIN
```

```
    DECLARE mi_nombre VARCHAR(20) DEFAULT 'Mi nombre es ';
```

```
    SELECT CONCAT (mi_nombre,nombre);
```

```
END$$
```

```
CREATE DEFINER='root'@'localhost' PROCEDURE `divisiones`
```

```
(IN dividendo TINYINT,IN divisor TINYINT)
```

```
BEGIN
```

```
    DECLARE resultado DOUBLE;
```

```
    SET resultado = dividendo / divisor;
```

```
    SELECT resultado;
```

```
END
```

Ejercicios:

1. Crear un procedimiento para multiplicar.
2. Pasar dos cadenas a un Stored y que las concatene en una variable interna, previo a mostrar el resultado en pantalla.

BORRADOR

Estructuras de control

Las estructuras de control nos permitirán modificar el flujo de ejecución de nuestro código, alterándolo según condiciones que creamos necesarias.

MySQL soporta las siguientes estructuras: IF, CASE, WHILE, ITERATE, LEAVE LOOP and REPEAT dentro de los Stored Procedures. Las más utilizadas son IF, CASE y WHILE.

El formato del **IF** es el siguiente:

```
IF expresion_booleana1 THEN
    -- Código 1
[ELSEIF expresion_booleana2 THEN
    -- Código 2
]
[ELSE
    -- Código 3
]
END IF;
```

El ítem marcado como expresion_booleana podrá ser cualquier expresión que devuelva un TRUE o FALSE. Por ejemplo: variable < 5; variable_booleana; SUBSTRING(variable,0,3) = 'Ana'; etc.

La estructura marca que SI se cumple la EXPRESION, ENTONCES ejecutará el código interno 1; SINO pero se cumple EXPRESION 2, ENTONCES ejecutará el código interno 2; SINO ejecutará el código 3.

```
IF variable > 5 THEN
    SELECT 'La variable es mayor a 5';
ELSEIF variable < 5 THEN
    SELECT 'La variable es menor a 5';
ELSE
    SELECT 'La variable es igual a 5';
END IF;
```

La estructura del **CASE** es la siguiente:

```
CASE variable
    WHEN valor1 THEN
        -- Código
    WHEN valor2 THEN
        -- Código
    ELSE
        -- Código
END CASE;
```

O también puse ser:

```

CASE
    WHEN variable = valor1 THEN
        -- Código
    WHEN variable = valor2 THEN
        -- Código
    ELSE
        -- Código
END CASE;

```

Esta estructura de control es ideal para reemplazar a los IF múltiples, ejecutándose con la misma lógica. Por ejemplo:

```

CASE variable
    WHEN 1 THEN
        SELECT 'La variable es igual a 1';
    WHEN 2 THEN
        SELECT 'La variable es igual a 2';
    ELSE
        SELECT 'La variable no cumple ninguna condición';
END CASE;

```

La estructura del **WHILE** es la siguiente:

```

WHILE expresion_booleana DO
    -- Código
END WHILE

```

El WHILE nos servirá para ejecutar bloques de código en forma cíclica, repitiéndolo tantas veces como necesitemos, mientras se cumpla la expresión booleana que coloquemos. Por ejemplo:

```

DECLARE variable TINYINT;
SET variable = 0;
WHILE variable < 5 DO
    SELECT CONCAT('Variable: ',variable);
    SET variable = variable + 1;
END WHILE;

```

La estructura de **REPEAT** es la siguiente:

```

REPEAT
    -- Código
UNTIL expresion_booleana
END REPEAT;

```

Al igual que en el WHILE, el código se repetirá mientras la expresión sea verdadera:

```

SET variable = 0;

```

```

REPEAT
    SET variable = variable + 1;
    SELECT CONCAT('Variable: ',variable);
UNTIL variable > 5
END REPEAT;

```

La estructura de **LOOP** incorpora a **LEAVE** e **ITERATE**, siendo su forma:

```

Etiqueta: LOOP
    -- Código
    -- Condición para iterar
    ITERATE Etiqueta;
    --Fuera de la condición
    LEAVE Etiqueta;
END LOOP Etiqueta;

```

La sentencia **ITERATE** indica que seguiremos iterando dentro del bucle. Si no pasamos por esta, deberíamos pasar por **LEAVE** para indicar que deseamos romper la iteración de la etiqueta que marquemos.

El funcionamiento es complejo, aunque similar al **WHILE**, quedará más claro en el siguiente ejemplo:

```

SET variable = 0;
etq1: LOOP
    SET variable = variable + 1;
    SELECT CONCAT('Variable: ',variable);
    IF variable < 5 THEN
        ITERATE etq1;
    END IF;
    LEAVE etq1;
END LOOP etq1;

```

En el ejemplo iteraremos una variable desde 0 hasta 4. En cada iteración incrementaremos su valor y lo imprimiremos en pantalla:

```

SET variable = variable + 1;
SELECT CONCAT('Variable: ',variable);

```

Luego, si el valor de nuestra variable es menor a 5, seguiremos iterando:

```

IF variable < 5 THEN
    ITERATE etq1;
END IF;

```

Caso contrario, rompemos el bucle:

LEAVE etq1;

Ejercicios:

1. Crear un procedimiento llamado 'aprobo' que reciba un número del 1 al 10. Si el valor es menor a 4, retorna por pantalla 'Aplazado'; si el valor es entre 4 y 7, 'Aprobado condicional'; y si es mayor a 7, 'Aprobado'.
2. Crear un procedimiento que reciba números del 0 al 6 llamado diasDeLaSemana. Según el valor deberá retornar de 00-Domingo a 06-Sábado.
3. Crear el procedimiento 'tablas', al cual le pasamos un número y no retorna la tabla de multiplicar del 0 al 10 (0xvariable=...;1xvariable=...;...;10xvariable=...).

Resolución:

1.

Cursores

Los cursores son utilizados para iterar dentro de un conjunto de resultados de una consulta y procesar cada una de las filas de forma independiente.

MySQL soporta cursores dentro de sus procedimientos almacenados. Su nomenclatura es la siguiente:

```
DECLARE cursor_nombre CURSOR FOR consulta;  
DECLARE CONTINUE HANDLER FOR NOT FOUND OPEN cursor_nombre;  
FETCH cursor_nombre INTO variable1 [, variable2, ...];  
CLOSE cursor_nombre;
```

Desmenucemos el código:

- DECLARE cursor_nombre CURSOR FOR consulta;

Declaramos el cursor y lo cargamos con los resultados de la consulta SELECT deseada.

- DECLARE CONTINUE HANDLER FOR NOT FOUND SET varAux = valor;

Especificamos que hacer cuando no encuentre ningún resultado más. Este valor podrá ser el que rompa el ciclo de un bucle.

- OPEN cursor_nombre;

Abrimos el cursor.

- FETCH cursor_nombre INTO variable1 [, variable2, ...];

Cargamos los valores de cada columna del SELECT dentro de variables. La cantidad de variables debe ser igual a la cantidad de columnas consultadas.

- CLOSE cursor_nombre;

Finalmente, cerramos el cursor.

Veamos un ejemplo de uso:

```
DELIMITER $$
CREATE DEFINER=`root`@`localhost` PROCEDURE `procCursor`(OUT resultado INT)
BEGIN
    DECLARE cur1 CURSOR FOR SELECT sueldo FROM personas;
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET b = TRUE;
    OPEN cur1;

    DECLARE aux,suma INT;
    DECLARE b BINARY;
    SET b = FALSE;
    SET suma = 0;
    WHILE b = FALSE DO
        FETCH cur1 INTO aux;
        IF b = FALSE THEN
            SET suma = suma + aux;
        END IF;
    END WHILE;
    CLOSE cur1;
    SET resultado = suma;
END
```

Este código WHILE se ejecutara mientras 'b' sea FALSE. ¿En qué momento va a cambiar su valor 'b'?

El valor de 'b' cambiará al llegar al final de los datos almacenados en nuestro cursor. Esto se lo indicamos al programa en una de las primeras sentencias:

```
DECLARE CONTINUE HANDLER FOR NOT FOUND SET b = TRUE;
```


Aquí sumaremos todos los sueldos de todos los empleados que figuren en la tabla de personal.

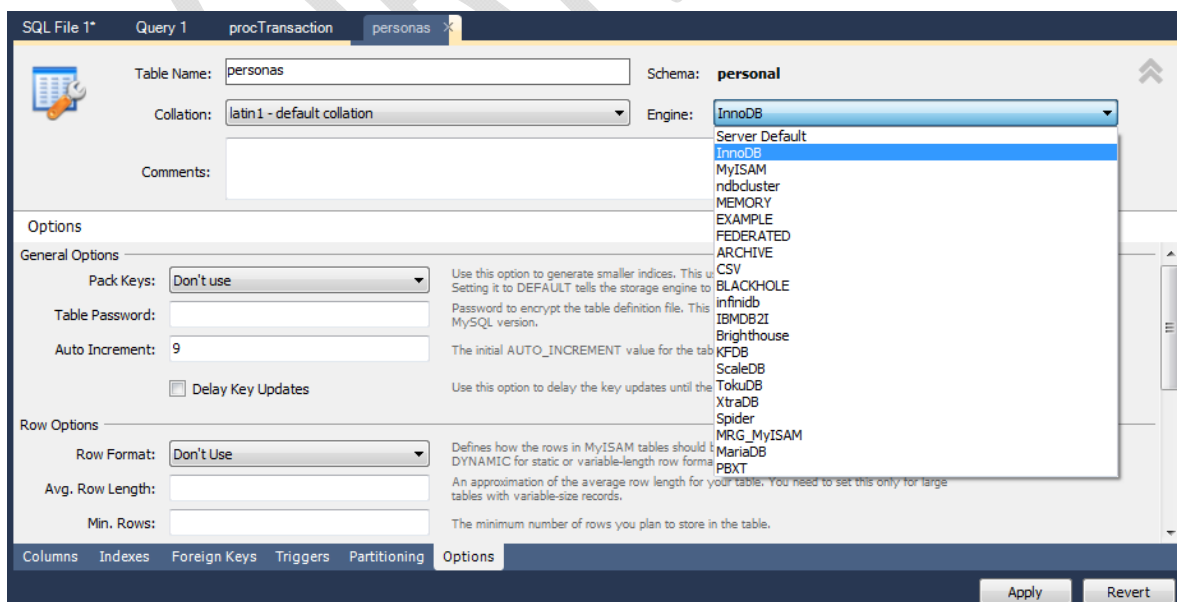
Los cursores tienen 3 características que debemos tener en cuenta, para no desilusionarnos al momento de usarlos:

- No sensible: si se actualizan datos dentro de la tabla consultada después de ejecutar la sentencia OPEN, este cambio no se reflejará en nuestro cursor.
- Sólo lectura: no podremos usar el cursor para actualizar datos dentro de una tabla; sirven sólo para consultas.
- Desplazamiento hacia adelante: tienen sólo una forma de recorrerse, hacia adelante. No podremos consultar desde el final hacia el principio, ni por índices.

Transacciones

Una transacción es una unidad atómica (única e indivisible) de las operaciones realizadas contra una o varias tablas, que pueden estar repartidas en una o varias bases de datos. Los efectos que estas sentencias tengan sobre la base de datos pueden ser aceptados y ejecutados al mismo tiempo (COMMIT) o podremos echar atrás el proceso sin que tengan efecto alguno sobre los datos (ROLLBACK).

Actualmente nuestras tablas, por defecto, usarán un motor del tipo MyISAM, que no soporta transacciones. Elijamos modificar nuestras tablas, vamos a la solapa 'Options', ahí dentro a 'Engine' y elijamos InnoDB o BDB:



Estos motores tendrán diferencias de optimización y manejo de datos, según necesidades muy específicas. Por lo general no hará falta cambiarlo, pero en ciertos casos es indispensable.

Por defecto, MySQL tiene AUTOCOMMIT activado; así ante cada sentencia se modificarán los datos de las tablas.

Para desactivar el AUTOCOMMIT, debemos ejecutar:

```
SET AUTOCOMMIT=0;
```

Si no sabemos en cuál estado está el AUTOCOMMIT de nuestro motor, podemos averiguarlo haciendo lo siguiente:

```
SELECT @@autocommit;
```

Si realizamos esta acción, deberemos ejecutar COMMIT cada vez que queramos realizar una operación que afecte los datos en disco (UPDATE, DELETE o INSERT) o ROLLBACK si deseamos cancelarla.

En cambio si lo que deseamos es desactivar esto momentáneamente y por una serie finita de comandos dentro de un procedimiento, podremos utilizar START TRANSACTION:

```
DECLARE var_sueldo INT;  
START TRANSACTION;  
SELECT sueldo INTO var_sueldo FROM personas WHERE id = 1;  
UPDATE personas SET sueldo = var_sueldo * 1.2 WHERE id = 3;  
COMMIT;  
SELECT 'Commit';
```

Si cambiamos COMMIT por ROLLBACK veremos que los datos no resultan afectados una vez ejecutado el procedimiento:

```
DECLARE var_sueldo INT;  
START TRANSACTION;  
SELECT sueldo INTO var_sueldo FROM personas WHERE id = 1;  
UPDATE personas SET sueldo = var_sueldo * 1.2 WHERE id = 2;  
ROLLBACK;  
SELECT 'Rollback';
```

Si realizamos un ROLLBACK sobre una tabla que no sea transaccional, el motor nos avisará con una advertencia del tipo ER_WARNING_NOT_COMPLETE_ROLLBACK; aunque igualmente modificará los datos de esta tabla.

Un ROLLBACK puede ser una operación lenta, que a veces se ejecuta aun sin el pedido del usuario; por ejemplo cuando ocurre un error no manejado dentro de procedimiento.

FUERA DE PROGRAMA

También podremos modificar el nivel de aislamiento de nuestra transacción.

<http://dev.mysql.com/doc/refman/5.0/es/set-transaction.html>

<http://zetcode.com/databases/mysqltutorial/transactions>

BORRADOR

Sentencias preparadas

MySQL 5.0 proporciona soporte para comandos preparados en la parte del servidor.

Supongamos que necesitamos una forma de consultar tablas de forma dinámica. Por ejemplo, tenemos una serie de tablas numeradas para distintos usos:

- Tabla01
- Tabla02
- Tabla03

Y deseamos consultar sus datos con la misma consulta dentro de un Stored Procedure. ¿Cómo haríamos?

```
DELIMITER $$
CREATE PROCEDURE `consultarTablas`(IN in_table CHAR(2))
BEGIN
    SET @consulta = CONCAT('SELECT *
        FROM Tabla',in_table);
    PREPARE stmt_name FROM @consulta;
    EXECUTE stmt_name;

    DEALLOCATE PREPARE stmt_name;
END
```

Primero cargamos nuestra consulta como un texto común y corriente, aprovechándonos de las cualidades de la función CONCAT.

PREPARE le asigna un nombre a una sentencia que contenemos en una variable de texto o en un literal de cadena de texto. Luego podremos utilizar esta consulta preformada cuantas veces necesitemos. Esto optimiza nuestra consulta, haciendo mayor la performance.

EXECUTE ejecutará la consulta.

Cuando terminamos de utilizar nuestra consulta, podremos hacer DEALLOCATE para descargarla de memoria:

```
DEALLOCATE PREPARE stmt2;
```

Probémoslo:

```
CALL consultarTablas('01');
```

En nuestra consulta podremos utilizar un comodín '?', el cual utilizaremos para marcar donde irá una variable. Para entenderlo mejor, veamos el ejemplo:

```
DELIMITER $$
CREATE PROCEDURE `consultarPersona`(IN valorId INT)
BEGIN
    SET @consulta = 'SELECT *
                    FROM personas
                    WHERE id = ?';
    PREPARE stmt_name FROM @consulta;
    EXECUTE stmt_name USING @valorId;

    DEALLOCATE PREPARE stmt_name;
END
```

Probémoslo:

```
CALL consultarPersona (1);
```

Ejercicio:

1. Crear un procedimiento al cual le pasemos un número y nos devuelva la tabla de multiplicar.
2. Crear un procedimiento al cual le pasemos los dos valores de los catetos de un triángulo y nos devuelva el tamaño de su hipotenusa (la raíz cuadrada de: el cuadrado del primero + el cuadrado del segundo).
3. Consultar a todas las personas por ID, una por una, utilizando los conocimientos adquiridos a lo largo del curso.

Resolución:

1. DELIMITER \$\$
CREATE PROCEDURE `tablaMultiplicar`(IN valor INT)
BEGIN
 DECLARE variable TINYINT;
 SET @consulta = 'SELECT ? * ? AS valores';
 PREPARE stmt_name FROM @consulta;

 SET @variable = 0;

```

WHILE @variable < 10 DO
    EXECUTE stmt_name USING @valor, @variable;
    SET @variable = @variable + 1;
END WHILE;

DEALLOCATE PREPARE stmt_name;
END
2. DELIMITER $$
CREATE PROCEDURE `calculoHipotenusa`
(IN cateto1 INT,
 IN cateto2 INT)
BEGIN
    SET @consulta = 'SELECT SQRT(POW(?,2) + POW(?,2)) AS hipotenusa';
    PREPARE stmt_name FROM @consulta;
    EXECUTE stmt_name USING @cateto1, @cateto2;

    DEALLOCATE PREPARE stmt_name;
END
3. DELIMITER $$
CREATE PROCEDURE `consultarPersonas`()
BEGIN
    DECLARE variable TINYINT;
    SET @consulta = 'SELECT * FROM personas WHERE id = ?';
    PREPARE stmt_name FROM @consulta;

    SELECT COUNT(id) INTO @maximo FROM personas;

    SET @variable = 1;
    WHILE @variable < @maximo DO
        EXECUTE stmt_name USING @variable;
        SET @variable = @variable + 1;
    END WHILE;

    DEALLOCATE PREPARE stmt_name;
END

```

Webs

<http://oreilly.com/catalog/sqlnut/chapter/ch04.html>

<http://dev.mysql.com/doc/refman/4.1/en/date-and-time-functions.html>

<http://es.scribd.com/doc/13268821/Funciones-de-agregados>

Base de datos de prueba que nos provee la gente de MySQL

<http://downloads.mysql.com/docs/world.sql.gz>.

ⁱ Información tomada de <http://www.desarrolloweb.com/articulos/1054.php>

ⁱⁱ Ayuda principal: <http://net.tutsplus.com/tutorials/an-introduction-to-stored-procedures>