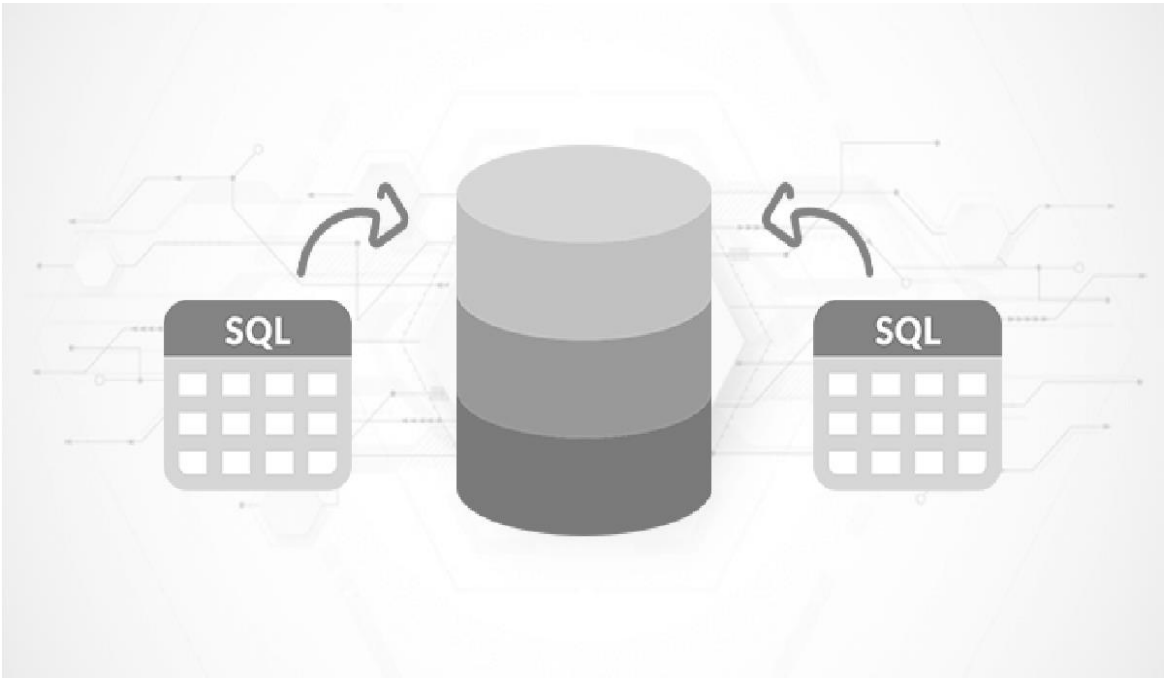
TABLAS EN SQL

"Un buen orden es la base de todas las cosas."

(Edmund Burke)

INTRODUCCION

Como vimos anteriormente las aplicaciones web se pueden dividir en dos componentes principales: un front-end que muestra y recoge información y un back-end para el almacenamiento de la información. Una base de datos puede contener una o más tablas. Una tabla es muy similar a una planilla de cálculo está formada por filas y columnas. Todas las filas tienen las mismas columnas y cada columna contiene un dato en sí. Una base de datos relacional es un tipo de base de datos que organiza los datos en tablas, y los vincula, basado en relaciones definidas.



CONCEPTOS DE NORMALIZACIÓN

La normalización es el proceso de organizar datos en una base de datos. Esto incluye la creación de tablas y el establecimiento de relaciones entre esas tablas de acuerdo con las reglas diseñadas tanto para proteger los datos como para que la base de datos sea más flexible mediante la eliminación de la redundancia y las dependencias incoherentes. Los datos redundantes desperdician espacio en disco y aumentan la probabilidad de que se produzcan errores e incoherencias. El segundo principio es que es importante que la información sea correcta y completa. Si la base de datos contiene información incorrecta, los informes que recogen información de la base de datos contendrán también información incorrecta y, por tanto, las decisiones que tome a partir de esos informes estarán mal fundamentadas.

Para eliminar estos problemas existen las Formas Normales. Existen básicamente tres formas Normales (FN). Las reglas para la forma normal son acumulativas. Es decir, para que una entidad cumpla las reglas de la segunda forma normal, también debe cumplir las reglas de la primera forma normal. Aunque son posibles otros niveles de normalización, la tercera forma normal se considera el nivel más alto necesario para la mayoría de las aplicaciones. Al igual que con muchas reglas y especificaciones formales, los escenarios del mundo real no siempre permiten el cumplimiento perfecto. En general, la normalización requiere tablas adicionales y algunos clientes lo consideran engorrosos. Si decide infringir una de las tres primeras reglas de normalización, asegúrese de que la aplicación anticipa los problemas que puedan producirse, como datos redundantes y dependencias incoherentes.

PRIMERA FORMA NORMAL (1FN)

Al aplicar el proceso de normalización en una relación, se empieza por comprobar si la relación está en primera forma normal y, si no es el caso, se efectúan las modificaciones oportunas para conseguirlo. Para que una base de datos esté en primera forma normal (1FN) cada columna de una tabla debe ser atómica, es decir, que cada atributo debe contener un único valor del dominio. Por ejemplo, suponga que tiene varios modelos de zapatillas, pero cada modelo se ofrece con distintos colores. Esta podría ser la situación:

Código Zapatilla	Modelo	Precio	Colores
261-1368	GALAXY 5	5519	Azul, Negra, Roja
261-1523	GRAUNT COURT	6599	Negra, Roja

Como se ve, esta relación no está en primera forma normal (1FN) ya que el atributo colores es multivaluado. Tampoco la siguiente entidad estaría en primera forma normal:

Código	Modelo	Color1	Color 2	Color3
261-1368	GALAXY 5	Azul	Negra	Roja

Pregúntese: ¿Qué sucede si deseamos agregar un nuevo color de zapatillas a ese modelo? Agregar un nuevo campo denominado color4 no es la respuesta; requiere modificaciones en la tabla y no admite de forma fluida un número dinámico de colores. ¿Qué pasaría si agregamos un nuevo modelo de zapatillas que tiene un solo color? ¿Cuántos atributos quedarían vacíos? Para transformar una tabla en una normalizada a la forma 1NF, tenemos que garantizar que haya un único valor en la intersección de cada fila y columna. Esto se consigue eliminando el grupo repetitivo. En su lugar, inserte toda la información de los colores de las zapatillas en una tabla separada llamada colores y, a continuación, vincule la tabla modelos a colores.

SEGUNDA FORMA NORMAL (2FN)

Una tabla está en segunda forma normal (2FN) cuando está en 1FN y además todos los atributos que no forman parte de la clave principal tienen dependencia funcional de la clave completa y no de parte de ella. Las reglas definidas nos plantean las siguientes dudas: ¿Qué es una clave principal? ¿Qué es una dependencia funcional? Una clave principal es un conjunto de una o más columnas que identifican de manera única (no repetida) a una fila.

Una Dependencia Funcional es una relación de implicancia entre 2 columnas, se dice que un atributo tiene dependencia funcional de otro cuando a cada valor del primero le corresponde un solo valor del segundo. Ejemplo:

Código Zapatilla	Modelo
261-1368	GALAXY 5
261-1523	GRAUNT COURT

De la entidad zapatillas podemos determinar que a cada código de zapatilla le corresponde un único modelo de zapatilla. Por tanto, hay una dependencia funcional entre ellos: el código de zapatilla determina el modelo de esa zapatilla (no al revés). Eso quiere decir que para cada código de zapatilla solo puede haber un modelo. La Dependencia funcional completa es cuando un atributo depende de otro que es un atributo compuesto (un conjunto de atributos), se dice que la dependencia funcional es completa si el atributo dependiente no depende de ningún subconjunto del atributo compuesto. Supongamos que tenemos la clave principal compuesta por Código de Zapatilla y Código de Marca, la descripción de la marca depende de un subconjunto de la clave. No de toda la clave.

Código Zapatilla	Codigo Marca	Marca
261-1368	1	ADIDAS
261-1523	1	ADIDAS

Por lo tanto, no hay dependencia funcional completa. Si la clave primaria de una tabla se compone de un solo atributo automáticamente está en 2FN.

Nota

Si A y B son subconjuntos de atributos de una tabla, diremos que B depende funcionalmente de A (o también que A determina a B) si cada valor de A tiene asociado siempre un único valor de B.
*A → B (**se lee: A determina a B**)*

TERCERA FORMA NORMAL 3FN

Una base de datos está en tercera forma normal (3FN) si está en 2FN y no existen atributos que no pertenezcan a la clave primaria que puedan ser conocidos mediante otro atributo que no forme parte de la clave primaria. Es decir, que no existan dependencias funcionales transitivas. Veamos el siguiente ejemplo (continuamos con nuestra empresa de zapatillas):

Código Zapatilla	Modelo	Precio	Genero	Codigo Marca	Marca
261-1368	GALAXY 5	5519	Hombre	1	ADIDAS
261-1523	GRAUNT COURT	6599	Hombre	1	ADIDAS

Como vemos la clave principal en esta entidad es el código de zapatilla ya que define de manera única a cada fila. Pero que sucede cuando trabajamos con dependencias funcionales, la descripción de la marca depende del código de la marca (no del código de

la zapatilla). Es decir, si modificamos la descripción de la marca debería cambiar el código de marca. Por ello debemos sacar dichas columnas y formar una nueva entidad relacionándola con la entidad modelos de zapatillas.

TIPOS DE RELACIONES

Las relaciones fomentan la introducción de datos coherentes y aplican reglas en la base de datos. Las Bases de Datos admiten los siguientes tipos de relaciones.

1) **De uno a uno:** Una tupla de una entidad está asociado a solo una tupla de la otra entidad.
2) **De uno a muchos, o de muchos a uno:** Una tupla de una entidad está asociado a una o varias tuplas de otra entidad. O bien, una o varias tuplas de una entidad puede estar asociada a una tupla de otra entidad.
3) **Muchos a muchos:** En general, los sistemas de bases de datos relacionales no permiten implementar una relación directa de muchos a muchos entre dos entidades. Sin embargo, puede diseñar una base de datos que admita relaciones de muchos-a-muchos utilizando una tercera entidad intermedia.

CREAR UNA BASE DE DATOS

SQL (Structured Query Language) es un lenguaje que utilizan los motores de base de datos para comunicarse con las bases de datos, gracias a este lenguaje se puede definir, manipular y recuperar datos. Para crear una base de datos en SQL debemos escribir lo siguiente:

```
create database <nombre_bdd>;
```

CREAR UNA TABLA

Para crear tablas en SQL se usa la instrucción create table.

```
create table <nombre_tabla>;
```

Pero antes de trabajar con esta instrucción veremos algunas pequeñas cosas para el armado de nuestra tabla.

TIPOS BASICOS DE DATOS EN MySQL

Cada vez que tengamos que crear una tabla que sirva para almacenar datos de una aplicación Web, debemos poner a prueba nuestra capacidad para definir los tipos de datos que con mayor eficiencia puedan almacenar cada dato que necesitemos guardar. Los campos de las tablas MySQL nos dan la posibilidad de elegir entre tres grandes tipos de contenidos: datos numéricos, datos para guardar cadenas de caracteres (alfanuméricos) y datos para almacenar fechas y horas.

DATOS NUMERICOS

Dentro de los datos numéricos, podemos distinguir dos grandes ramas: enteros y decimales. Los campos numéricos pueden ser positivos y negativos cuando son del tipo signed y solo positivos cuando son del tipo unsigned. Si son unsigned, el valor máximo puede ser el doble que cuando son signed.

Data Type	Min value (signed)	Max value (signed)	Min value (unsigned)	Max value (unsigned)
TINYINT	-128	127	0	255
SMALLINT	-32768	32767	0	65535
MEDIUMINT	-8388608	8388607	0	16777215
INT	-2147483648	2147483647	0	4294967295
BIGINT	-9223372036854775808	9223372036854775807	0	18446744073709551615

Los números con decimales son necesarios para almacenar precios, salarios, importes de cuentas bancarias, etc. Números que no son enteros. Tenemos que tener en cuenta que, si bien estos tipos de datos se llaman "de coma flotante", por ser la coma el separador entre la parte entera y la parte decimal, en realidad MySQL los almacena usando un punto como separador. En esta categoría, disponemos de tres tipos de datos: FLOAT, DOUBLE y DECIMAL. La estructura con la que podemos declarar un campo FLOAT implica definir dos valores: la longitud total (incluyendo los decimales y la coma), y cuántos de estos dígitos son la parte decimal. Por ejemplo:

FLOAT (6.2)

Esta definición permitirá almacenar como mínimo el valor -999.99 y como máximo 999.99 (el signo menos no cuenta, pero el punto decimal sí, por eso son seis dígitos en total, y de ellos dos son los decimales). La cantidad de decimales (el segundo número entre los paréntesis) debe estar entre 0 y 24, ya que ése es el rango de precisión simple. En cambio, en el tipo de dato DOUBLE, al ser de doble precisión, sólo permite que la cantidad de decimales se defina entre 25 y 53. Debido a que los cálculos entre campos en MySQL se realizan con doble precisión (la utilizada por DOUBLE) usar FLOAT, que es de simple precisión, puede traer problemas de redondeo y pérdida de los decimales restantes.
Por último, DECIMAL es ideal para almacenar valores monetarios, donde se requiera menor longitud, pero la "máxima exactitud" (sin redondeos). Este tipo de dato le asigna un ancho fijo a la cifra que almacenará. El máximo de dígitos totales para este tipo de dato es de 64, de los cuales 30 es el número de decimales máximo permitido. Más que suficientes para almacenar precios, salarios y monedas.

Nota

MySQL no tiene el tipo de datos BOOLEAN para poder representar valores booleanos MySQL usa el tipo entero más pequeño que es TINYINT(1).

DATOS ALFANUMÉRICOS

Para almacenar datos alfanuméricos (cadenas de caracteres) en MySQL tenemos toda una serie de tipos de datos. Nosotros estudiaremos los más usados: CHAR, VARCHAR Y BLOB. El tipo de dato CHAR permite almacenar textos breves, de hasta 255 caracteres de longitud como máximo, aunque no lo utilicemos. Por lo tanto, no es eficiente cuando la longitud del dato que se almacenará en un campo es desconocida a priori (típicamente, datos ingresados por el usuario en un formulario, como su nombre, domicilio, etc.). Podemos usarlo, por ejemplo, cuando ingresos por ejemplo números de patentes de autos. El tipo VARCHAR (caracteres variables) es útil cuando la longitud del dato es desconocida, cuando depende de la información que el usuario escribe en campos o áreas de texto de un formulario. La longitud máxima permitida es de 65.535 caracteres. Este tipo de dato tiene la particularidad de que cada registro puede tener una longitud diferente, que dependerá de su contenido; si en su registro el campo "nombre" (supongamos que hubiera sido definido con un ancho máximo de 20 caracteres) contiene solamente el texto: "Pepe", consumirá sólo cinco caracteres, cuatro para las cuatro letras, y uno más que indicará cuántas letras se utilizaron. El tipo de dato Blob (Binary Large Object) almacena la información en formato binario. Por ejemplo, lo podemos usar para almacenar la foto de perfil de una red social. Puede almacenar hasta 65.535 Bytes de datos.

DATOS DE FECHA Y HORA

En MySQL, poseemos varias opciones para almacenar datos referidos a fechas y horas. Veamos las diferencias entre uno y otro, y sus usos principales, así podemos elegir el tipo de dato apropiado en cada caso. El tipo de dato DATE nos permite almacenar fechas en el formato: AAAA-MM-DD (los cuatro primeros dígitos para el año, los dos siguientes para el mes, y los últimos dos para el día). En los países de habla hispana estamos acostumbrados a ordenar las fechas en Día, Mes y Año, pero para MySQL es exactamente al revés. El rango de fechas que permite manejar desde el 1000-01-01 hasta el 9999-12-31. Un campo definido como DATETIME nos permitirá almacenar información acerca de un instante de tiempo, pero no sólo la fecha sino también su horario, en el formato: AAAA-MM-DD HH:MM:SS, siendo la parte de la fecha de un rango similar al del tipo DATE (desde el 1000-01-01 00:00:00 al 9999-12-31 23:59:59), y la parte del tiempo, de 00:00:00 a 23:59:59. El tipo de dato TIME nos permite almacenar horas, minutos y segundos, en el formato HH:MM:SS. El tipo de dato TIMESTAMP sirve para almacenar una fecha y un horario, de manera similar a DATETIME, pero su formato y rango de valores serán diferentes. El formato de un campo TIMESTAMP puede variar entre tres opciones: AAAA-MM-DD HH:MM:SS ,AAAA-MM-DD O AA-MM-DD. Es decir, la longitud posible puede ser de 14, 8 o 6 dígitos, según qué información proporcionemos. El rango de fechas que maneja este campo va desde el 1970-01-01 hasta el año 2038. También posee la particularidad de que podemos definir que su valor se mantenga actualizado automáticamente, cada vez que se inserte o que se actualice un registro usando la opción CURRENT_TIMESTAMP. De esa manera, conservaremos siempre en ese campo la fecha y hora de la última actualización de ese dato, que es ideal para llevar el control sin necesidad de programar nada. Por último, tenemos el tipo de dato YEAR en el podremos almacenar un año, tanto utilizando dos como cuatro dígitos.

VALOR PREDETERMINADO (DEFAULT)

Muchas veces necesitamos agilizar la carga de datos mediante un valor por defecto (default). Por ejemplo, pensemos en un sistema de pedidos, donde, al llegar el pedido a la base de datos, su estado sea "recibido", sin necesidad de que el sistema envíe ningún valor, sólo por agregar el registro, ese registro debería contener en el campo "estado" el valor de "recibido". Este es un típico caso de valor predeterminado o por default. Es importante señalar que no se puede dar un valor predeterminado a un campo de tipo TEXT ni BLOB (y todas sus variantes).

JUEGO DE CARACTERES

En una tabla se almacenan caracteres (cada letra, número, símbolo, es un carácter), y estos deben ser reconocibles por quien los lee. Como existen miles de caracteres en el mundo y su uso varía dependiendo de la región geográfica, idioma, etc. debemos colocarle a nuestra base de datos un conjunto de caracteres (charset) para que admita los caracteres que nos interesa. El charset puede definirse a nivel de servidor, base de datos, tabla y columna. El utf8 que debe usarse en mysql es el Utf8mb4 un conjunto de caracteres de 4 Bytes que aparte de permitir todos los símbolos del idioma español permite el uso de caracteres gráficos como emojis. Aparejado con el juego de caracteres, viene lo que se llama colación (collation), una colación es un conjunto de reglas para comparar caracteres en un conjunto de caracteres. Este también se puede definir igual que el juego de caracteres para servidor, base de datos, tabla y columna. En nuestro caso para nuestras tablas usaremos CHARSET= utf8mb4 y COLLATE= utf8mb4_general_ci.

Nota

Se puede asignar un conjunto de caracteres y colación a la base de datos ya que este es heredado por las tablas si no se especifica uno explícitamente. Por ejemplo, para crear la base de datos anterior debemos tipear: create database login character set utf8mb4 collate utf8mb4_general_ci;

SISTEMA DE VENTAS ONLINE

Crearemos un sistema para almacenar productos (en este caso zapatillas) donde habrá distintas marcas y modelos, cada zapatilla puede poseer distintos colores. Para ello crearemos la base de datos productos_db:

```
create database productos_db;
```

Una vez creada la base de datos, dentro de ella crearemos las tablas y sus relaciones. La primera tabla a crear será la entidad que contenga la información de las Marcas:

```
create table marcas (
id_marca int(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
descripcion varchar(50) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE utf8mb4_general_ci;
```

La segunda tabla contendrá los colores de las zapatillas:

```
create table colores (
id_color int(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
descripcion varchar(50) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE utf8mb4_general_ci;
```

Ahora, crearemos la tabla zapatillas, con todos los modelos de zapatillas que ofrecemos:

```
create table zapatillas (
id_zapatilla int(11) NOT NULL AUTO_INCREMENT PRIMARY KEY,
modelo varchar(50) NOT NULL,
precio decimal(7,2),
genero varchar(50),
id_marca int(11) NOT NULL,
fecha_alta TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
imagen varchar(255),
descripcion varchar(255)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE utf8mb4_general_ci;
```

Para almacenar los colores de cada modelo de zapatillas crearemos una nueva tabla denominada colores_zapatillas, la clave principal estará compuesta por las dos columnas que posee:

```
create table colores_zapatillas (
id_color int(11) NOT NULL,
id_zapatilla int(11) NOT NULL,
primary key (id_color, id_zapatilla)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE utf8mb4_general_ci;
```

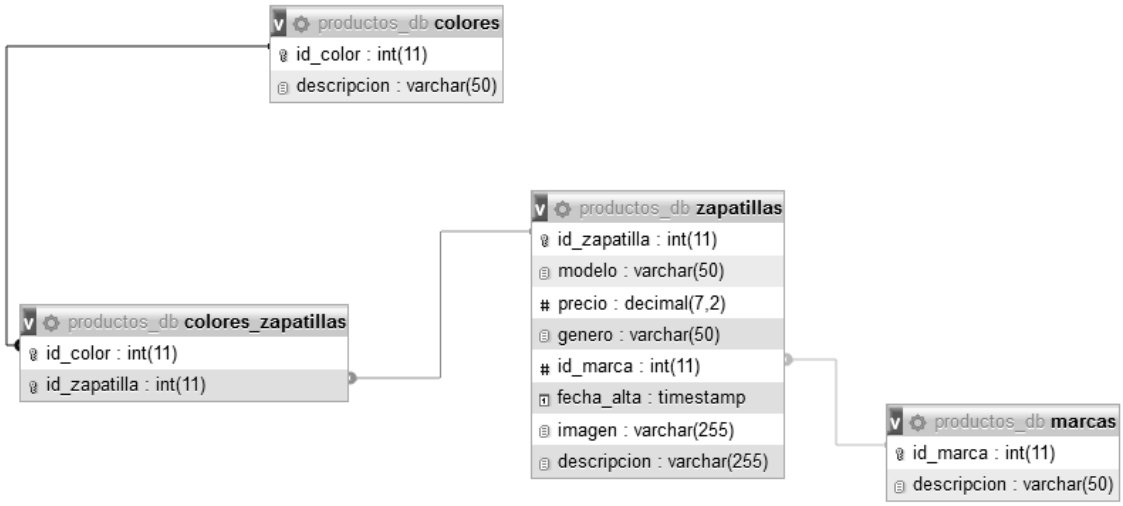
CLAVES FORANEAS

Uno de los objetivos de un buen diseño de base de datos es eliminar la redundancia de los datos (datos duplicados). Para lograr dicho objetivo, conviene desglosar los datos en distintas tablas donde cada una representa un tema distinto. Por ejemplo, supongamos que queremos asignar tareas a distintas personas de nuestra en una base de datos, supongamos también que todo se encuentra en una sola tabla, cada vez que ingresamos una tarea se la debemos asignar a una persona informando su nombre, e-mail de contacto y su número de interno. Esto con lleva a gastar mucho espacio en disco. Otro problema, tal vez mucho peor, ¿Qué pasaría si la persona cambia su interno por que se muda de oficina? Deberíamos modificar en la única tabla que tenemos todas las apariciones del viejo interno.
La solución es registrar la información de las tareas en una tabla y las personas en otra tabla y cuando necesitamos asignar la tarea a una determinada persona solo escribimos el id de la persona en la tabla tareas. Es decir, solo invocaríamos el id de la tabla personas sin colocar su nombre, e-mail y teléfono.
La id de la tabla personas en la tabla tareas es denomina clave foránea ya que contiene valores que coinciden con la clave primaria de otra tabla. Para añadir un registro nuevo en la tabla que contiene la clave foránea es necesario que el valor a insertar exista en la tabla relacionada. El otro punto para definir es que tabla o entidad se coloca la Clave Foránea, cuando la relación es uno a muchos (la más común) la clave foránea se coloca en la entidad de cardinalidad que hace la relación de muchos.
Dicho esto, nos pondremos a trabajar en las claves foráneas de nuestra base de datos productos_db:

```
--
-- Clave Foránea Marcas (uno) en la tabla zapatillas (varios)
--
ALTER TABLE zapatillas
ADD CONSTRAINT zapatillas_marcas FOREIGN KEY (id_marca) REFERENCES marcas
(id_marca);
--
-- Clave Foránea zapatillas (uno) y colores_zapatillas (varios)
--
ALTER TABLE colores_zapatillas
ADD CONSTRAINT colores_zapatillas FOREIGN KEY (id_zapatilla) REFERENCES
zapatillas(id_zapatilla);


--
-- Clave Foránea colores (uno) y colores_zapatillas (varios)
--
ALTER TABLE colores_zapatillas
ADD CONSTRAINT zapatillas_colores FOREIGN KEY (id_color) REFERENCES
colores(id_color);
```

Nuestro diagrama DER quedaría de la siguiente manera:



AGREGAR UNA NUEVA COLUMNA A LA TABLA

Podremos agregar una columna usando la instrucción ALTER TABLE, en esta instrucción definiremos además el nombre de la columna, el tipo de dato asociado y los atributos si lo posee:

```
alter table zapatillas add column disciplina VARCHAR(50);
```

MODIFICAR LA DEFINICION DE LA COLUMNA

Si necesitamos modificar alguna definición de la columna podemos usar la instrucción anterior, pero cambiando su modificador:

```
alter table zapatillas modify column disciplina VARCHAR(30);
```

Nota

Otra instrucción que puede sernos útiles es alter table NOMBRE_TABLA drop column NOMBRE_COLUMNA; , donde NOMBRE_TABLA es el nombre de la tabla y NOMBRE_COLUMNA es la columna a eliminar físicamente de la tabla.

INSERTANDO REGISTROS EN UNA TABLA

La instrucción para agregar nuevos registros a la tabla se denomina INSERT INTO y es posible escribir la instrucción de dos formas. La primera forma específica tanto los nombres de las columnas como los valores que se insertarán:

```
insert into <nombre_tabla> (column1, column2, column3,...columnN)
values (value1, value2, value3,...valueN);
```

Para esta instrucción debemos colocar primero el nombre de la tabla donde se incorporará el registro seguido de la lista de campos (separados por comas) donde se insertarán dichos valores. La segunda forma se usa en el caso de que se agreguen valores para todas las columnas de la tabla, en ese caso no es necesario que especifique los nombres de las columnas en la consulta SQL. Sin embargo, debemos asegurarnos de que el orden de los valores esté en el mismo orden que las columnas de la tabla. En este caso la sintaxis INSERT INTO sería más simple:

```
insert into <nombre_tabla>
values (value1, value2, value3,...valueN);
```

Ahora probaremos con unos inserts para ver si todo funciona correctamente.

```
--
-- Volcado de datos para la tabla Marcas
--
INSERT INTO marcas (id_marca, descripcion) VALUES
('', 'Nike'),
('', 'Adidas'),
('', 'Converse'),
('', 'Fila'),
('', 'Lacoste'),
('', 'New Balance');
--
--
-- Volcado de datos para la tabla Colores
--
INSERT INTO colores (id_color, descripcion) VALUES
('', 'Azul'),
('', 'Roja'),
('', 'Marrón'),
('', 'Negra'),
('', 'Blanca'),
('', 'Verde'),
('', 'Amarillo'),
('', 'Gris');
```

```
--
-- Volcado de datos para la tabla zapatillas
--
INSERT INTO ZAPATILLAS
(modelo,precio,genero,id_marca,imagen,descripcion,disciplina) VALUES ('Core
Lite Racer',8000,'Hombre',2,'adidas_lite.jpg','Zapatillas Adidas Core Lite
Racer De Entrenamiento Para Hombre','Runner');


--
-- Volcado de datos para la tabla colores_zapatillas
--
INSERT INTO colores_zapatillas (id_color, id_zapatilla) VALUES
(1, 1),
(4, 1);
```