Bases de datos (parte 2)

Rodrigo Alexander Castro Campos
Última actualización: 7 de julio de 2015

Este documento presenta un resumen de la UEA "Bases de datos" de la UAM Azcapotzalco, tal como yo la impartí durante el trimestre 15-P. Este documento está pensado para uso personal, aunque es posible que pueda convertirse en unas notas de curso o que sea tratado como tal.

Durante el curso se ven algunos temas que no están incluidos en el temario oficial, como la codificación de caracteres y el lenguaje PHP. Esto lo hago pues considero que:

- En la práctica, es vital saber al menos un poco de codificación de caracteres. Esto es particularmente cierto al momento de desarrollar sistemas de información y sistemas portables.
- Es necesario aprender a utilizar una base de datos desde un lenguaje de programación para aplicaciones del lado del servidor.

La elección del lenguaje PHP es personal. Por lo menos, parece ser cierto que PHP sigue siendo relevante a pesar del surgimiento de otros lenguajes como Python.

El lenguaje SQL

Ya visto el cálculo relacional, es fácil imaginarse qué consultas pueden enviarse a un manejador de bases de datos relaciones. El estándar ISO/IEC 9075 define un lenguaje llamado SQL (Structured Query Language) por sus siglas en inglés, el cual se originó del cálculo relacional. A través de los años (y dadas las limitaciones del cálculo relacional) se han ido agregando diferentes características a este lenguaje (por ejemplo, la versión de 2011 introduce soporte para XML).

Sin embargo, los diferentes gestores de bases de datos relacionales muchas veces difieren considerablemente en su manera de implementar el lenguaje. Por ejemplo, Oracle define un tipo llamado INTEGER mientras que MySQL lo puede llamar `INTEGER` o también `INT`; ni SQL ni Oracle definen el tipo `TINYINT`, el cual MySQL sí tiene. En este curso usaremos MySQL y no nos preocuparemos por las diferencias que puedan existir entre el estándar SQL y otros manejadores de bases de datos; de todos modos, es importante que el alumno esté enterado de la realidad.

Puede considerarse que el lenguaje SQL tiene dos grandes partes:

- El lenguaje de definición de datos: se encarga de definir bases de datos y relaciones.
- El lenguaje de manipulación de datos: se encarga de insertar, modificar y eliminar tuplas de relaciones, además de operaciones de consulta.

Los conceptos que nosotros vimos durante el estudio del cálculo relacional con respecto a cómo definir relaciones y cómo operar sobre ellas tendrán el análogo en SQL.

Conexión a un gestor MySQL

Una instalación ordinaria de MySQL crea inicialmente un usuario llamado `root` (y con una contraseña vacía si no se especificó algo diferente). Existen restricciones de conexión adicionales,

de las cuales hablaremos posteriormente. Sin embargo, una pregunta natural es ¿cómo hago la conexión en primer lugar? Existen varias opciones:

- Directamente desde la consola de comandos, invocando al ejecutable de MySQL.
- Desde un programa propio que haga uso de la API que MySQL facilite.
- Desde un programa especializado.

Nosotros haremos ejemplos que usen las tres formas listadas anteriormente y utilizaremos PHP para el segundo punto. El programa especializado que usaremos es phpMyAdmin, el cual proporciona una interfaz web a MySQL. Podrían darse argumentos de que no es el mejor programa especializado, pero ocasionalmente es útil en situaciones de restricciones severas de acceso usando algún otro método; otro programa que el alumno puede usar es MySQL Workbench.

Conexión a un gestor MySQL

Usando el programa phpMyAdmin podemos ver gráficamente todas las bases de datos que están dadas de altas en el servidor MySQL local. Desde EasyPHP podemos acceder a phpMyAdmin dando clic derecho al icono del servicio del mismo y luego seleccionado "Administration"; en el explorador aparece la opción "MySQL Administration : PhpMyAdmin". Ver figuras 1, 2 y 3.

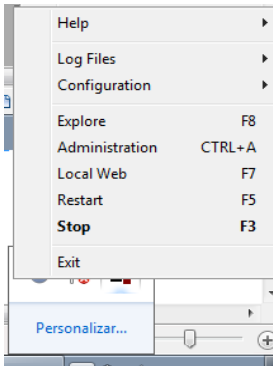

Figura 1 – Las opciones del servicio de EasyPHP y la opción "Administration"

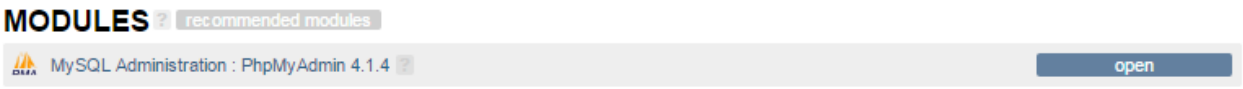

Figura 2 – La opción disponible en el explorador automáticamente abierto

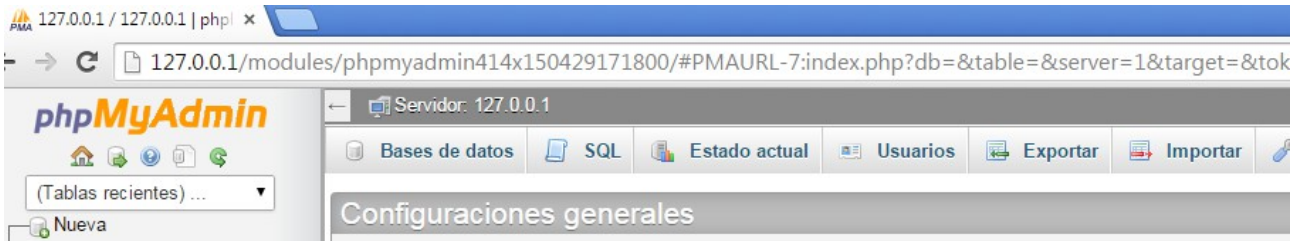

Figura 3 – La pantalla de inicio de phpMyAdmin

Debemos recordar que MySQL no es una base de datos sino un gestor de bases de datos. Dicho esto, un mismo gestor puede estar encargado de manejar más de una base de datos (por ejemplo, MySQL puede dar soporte a varias aplicaciones proporcionando una base de datos a cada una).

En el menú de la izquierda aparecen las bases de datos que ya hemos definido. Por obvias razones, si no hemos definido bases de datos entonces no aparecerán.

¿Cómo podemos crear una base de datos? Existen dos opciones: usando la interfaz gráfica de phpMyAdmin y usando una instrucción SQL. Para usar la interfaz gráfica podemos seleccionar la pestaña "Bases de datos" de phpMyAdmin y aparecerá un formulario en donde podremos ingresar el nombre (y la codificación o cotejamiento) de la base de datos (ver figura 4):

Figura 4 – La pestaña "Bases de datos" de phpMyAdmin.

Por otro lado, existe una manera programática que crear una base de datos usando el lenguaje SQL. Una base de datos se crea con la siguiente sentencia:

```
CREATE DATABASE nombre;
```

Esa instrucción se puede ingresar en la pestaña SQL. Nosotros creamos una base de datos llamada `ejemplo` (ver figura 5):

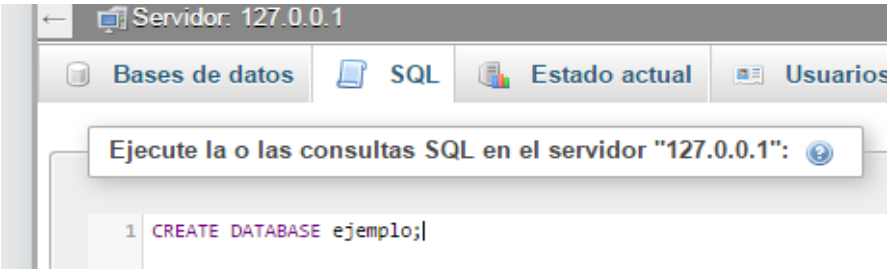

Figura 5 – La pestaña SQL en donde podemos ingresar instrucciones

Una sentencia en MySQL puede terminar con punto y coma (el cual es necesario si se va a ejecutar un lote de instrucciones). Las mayúsculas no son necesarias, pero es una convención que facilita diferenciar las (muchas) palabras clave de MySQL de los identificadores de relaciones o nombres de bases de datos.

Usando phpMyAdmin debemos estar atentos a un detalle: cuando creamos una base de datos usando la interfaz gráfica, ésta aparecerá inmediatamente en el menú izquierdo. Sin embargo, al

usar código SQL ésta no apararecerá de inmediato y tendremos que recargar la página para que ésta aparezca (ver figura 6):

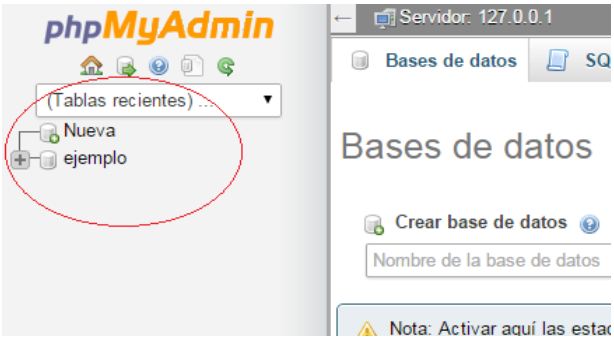

Figura 6 – Una base de datos creada debe aparecer en el menú izquierdo. De ser necesario, se deberá recargar la página para que aparezca.

Una base de datos puede eliminarse usando la interfaz de phpMyAdmin (pestaña "Bases de datos") o bien, con la siguiente sentencia SQL:

```
DROP DATABASE nombre;
```

Nosotros procuraremos usar la pestaña SQL para prácticamente todo lo que sea posible.

Es un error intentar crear una base de datos con un nombre duplicado o intentar eliminar una base de datos inexistente. Se puede crear o eliminar una base de datos condicionalmente:

```
CREATE DATABASE IF NOT EXISTS nombre;
DROP DATABASE IF EXISTS nombre;
```

Un esquema de una base de datos consiste en, entre otras cosas, de:

- El nombre de la base de datos.
- Los esquemas de sus relaciones y las tuplas almacenadas en ellas.
- Las funciones y procedimientos almacenados de la base de datos.
- Las vistas y desencadenadores definidos.

A continuación veremos cómo definir relaciones en una base de datos.

Definición de relaciones

Recordemos que en teoría relacional, el esquema de una relación consiste en su nombre y su encabezado (en el cual se definen los atributos: cada atributo está dado por un nombre y su tipo). Para definir el esquema `persona(nombre: cadena, apellido: cadena, edad: entero)` podemos usar la siguiente notación en SQL:

```
CREATE TABLE persona (
   nombre TEXT,
   apellido TEXT,
   edad INT
);
```

¿En qué base de datos se creará dicha relación (tabla)? En la base de datos que esté seleccionada en ese momento. ¿Cómo podemos seleccionar una base de datos? Usando phpMyAdmin, basta con darle clic a la base de datos deseada en el menú izquierdo; posteriormente podremos ingresar a la pestaña SQL para crear la relación (ver figura 7):

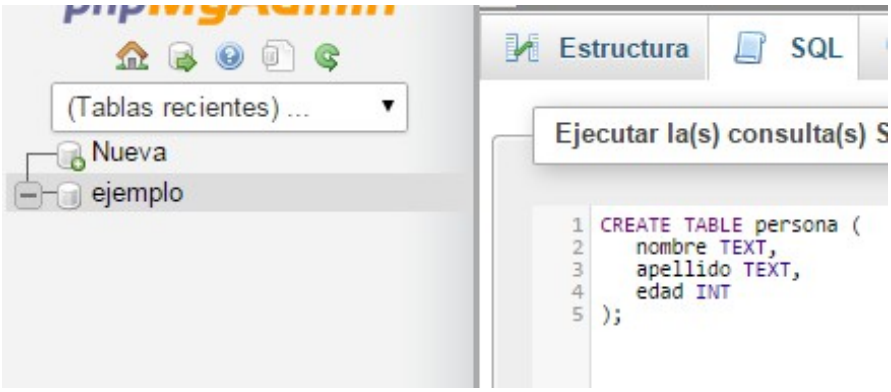

Figura 7 – La base de datos está seleccionada al momento de ingresar la instrucción

Podemos "seleccionar programáticamente" una base de datos con la siguiente instrucción SQL:

```
USE nombre;
```

No necesitamos seleccionar la base de datos desde la interfaz si usamos instrucción `USE` antes de ejecutar el `CREATE TABLE` (ver figura 8):

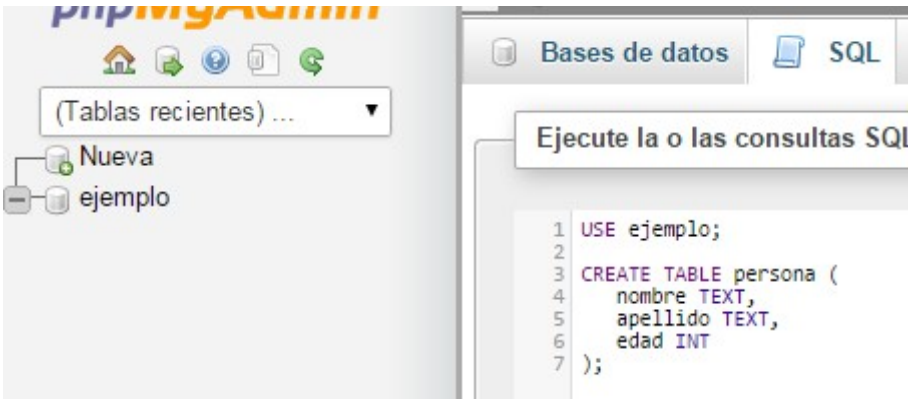

Figura 8 – La base de datos no está seleccionada en la interfaz, pero se selecciona programáticamente desde SQL

Al establecer una conexión a MySQL muchas veces es necesario especificar de antemano la base de datos (el "esquema" principal) al cual el usuario quiere acceder. En caso de que dicha selección no sea requerida al momento de establecer la conexión, debemos recordar que es necesario realizarla antes de poder manipular tablas.

Una vez seleccionada una base de datos, podremos definir relaciones nuevas. Recordemos que una relación tiene un encabezado, el cual define los atributos y sus tipos. La notación es la siguiente:

Como en el caso de bases de datos, es posible crear una relación o tabla condicionalmente con `CREATE TABLE IF NOT EXISTS`. También es posible eliminar una relación (y todas las tuplas que contenga) con `DROP TABLE` (y `DROP TABLE IF EXISTS`).

Inserción de tuplas

En cálculo relacional siempre trabajamos con relaciones "prellenadas" (es decir, con tuplas preexistentes). Por obvias razones, en un manejador de bases de datos como MySQL es posible insertar tuplas en una relación. Por ejemplo:

```
INSERT INTO persona (nombre, apellido, edad) VALUES ("pedro",
"perez", 15);
INSERT INTO persona (edad, apellido, nombre) VALUES (20, "ana",
"gomez");
```

El orden en el que los atributos son listados al momento de la inserción debe corresponder con el orden en el que son listados los valores de la tupla. Debemos recordar que la base de datos debe estar previamente seleccionada antes de poder insertar tuplas en relaciones.

A diferencia del cálculo relacional, una tabla de MySQL puede tener tuplas repetidas (es decir, en MySQL el cuerpo de una relación *no* es un conjunto, sino una lista de tuplas):

```
INSERT INTO persona (nombre, apellido, edad) VALUES ("alan",
"fernandez", 18);        # inserta una tupla
INSERT INTO persona (nombre, apellido, edad) VALUES ("alan",
"fernandez", 18);        # inserta otra tupla idéntica
```

El caracter `#` denota un comentario de línea en MySQL. Un comentario de bloque puede especificarse con la pareja `/*` ... `*/` al igual que en C.

Es posible insertar múltiples filas en un único INSERT usando la siguiente sintaxis:

```
INSERT INTO persona (nombre, apellido, edad) VALUES
   ("pedro", "perez", 15),
   ("ana", "gomez", 20),
   ("olivia", "rodriguez", 30);
```

El fragmento de código anterior usó saltos de líneas adicionales sólo para mejorar la lectura y no son necesarios.

Consultas simples del cálculo relacional en SQL

En SQL, se puede usar la sentencia `SELECT` para indicar simultáneamente proyección, selección y renombramiento. Supongamos que tenemos la siguiente sentencia:

```
SELECT nombre, edad AS viejo FROM persona WHERE apellido = "perez";
```

¿Cómo funciona el ejemplo anterior? Notemos lo siguiente:

- Después de la palabra `SELECT` van los atributos que deseamos listar. En este caso, el atributo `apellido` no está listado, por lo que la "proyección" de la consulta no lo incluye.
- La palabra `AS` indica que el atributo `edad` debe ser renombrado a `viejo` en la relación resultante.
- La palabra `WHERE` seguida de la comparación `apellido = "perez"` permite indicar una condición análoga a la de la selección de cálculo relacional. Como la selección se

realiza antes de enviar el resultado al cliente, la selección puede incluir atributos de la relación aunque no se proyecten.

La expresión de cálculo relacional equivalente a lo anterior sería:

$$\rho_{viejo/edad}(\pi_{nombre,edad}(\sigma_{apellido="perez"}(persona)))$$

MySQL y otros manejadores de bases de datos permiten "proyectar" expresiones que no dependen de atributos de la relación original; también permiten proyectar resultados construidos a partir de expresiones y atributos:

```
INSERT INTO persona (nombre, apellido, edad) VALUES ("pedro",
"perez", 15);
INSERT INTO persona (edad, apellido, nombre) VALUES (20, "ana",
"gomez");
```

SELECT nombre, 123, edad * 2 FROM persona		
nombre: TEXT	123: INT	edad * 2: INT
pedro	123	30
ana	123	40

Los nombres de las columnas "construidas" los decide MySQL, aunque podemos indicarle qué nombre queremos usando `AS` seguido del nombre deseado:

SELECT nombre, 123 AS raro, edad * 2 AS doble FROM persona		
nombre: TEXT	raro: INT	doble: INT
pedro	123	30
ana	123	40

Cuando se especifica un asterisco `*` en un `SELECT`, todo lo de una relación es proyectado:

SELECT * FROM persona		
nombre: TEXT	apellido: TEXT	edad: INT
pedro	perez	15
ana	gomez	20

Debe evitarse el uso de esta característica: es posible que nuevos atributos sean agregados posteriormente a una relación y que MySQL proyecte más columnas de las necesarias; esto consume tiempo y memoria de forma innecesaria.

La conjunción $\wedge$ y la disyunción $\vee$ pueden escribirse en MySQL con las palabras `AND` y `OR` repspectivamente:

```
SELECT nombre, apellido FROM persona WHERE edad = 20 OR (edad < 14
AND edad != 10);
```

Es importante saber usar las condiciones, pues se usarán mucho al actualizar y eliminar filas.

Eliminación de tuplas en SQL

Tuplas insertadas en una relación pueden ser eliminadas posteriormente. Para ilustrar la sintaxis, mostraremos el siguiente ejemplo:

```
DELETE FROM persona WHERE apellido = "perez";
```

Debemos tener mucho cuidado con la sentencia `DELETE`, pues borrará todas las tuplas que cumplan la condición (y si olvidamos la condición ¡se borrarán todas las tuplas!):

```
DELETE FROM persona;     # cuidado, borra todas las tuplas
```

Si en verdad queremos vaciar una relación, deberíamos ser explícitos y usar la sentencia `TRUNCATE`:

```
TRUNCATE persona;        # borra todas las tuplas
```

Modificación de tuplas en SQL

Dada la siguiente relación:

SELECT * FROM persona		
nombre: TEXT	ahorros: INT	edad: INT
pedro	100	15
ana	200	20

Si nosotros queremos incrementar la edad de ana en 1 y duplicar sus ahorros, podemos ejecutar la siguiente consulta:

```
UPDATE persona SET edad = edad + 1, ahorros = ahorros * 2 WHERE
nombre = 'ana';
```

Al igual que en las sentencias `DELETE`, la condición es opcional (¡pero no ponerla es equivalente a modificar todas las filas de la tabla!).

Listado de bases de datos y tablas

Una lista de las bases de datos disponibles puede conseguirse con la siguiente instrucción:

```
SHOW DATABASES;          # lista las bases de datos disponibles
```

Esta instrucción (al igual que la sentencia `SELECT`) regresa una relación (en este caso, de una sola columna y tantas filas como bases de datos existan). De manera similar se pueden listar las tablas de una base de datos (siempre y cuando ésta haya sido seleccionada anteriormente con la sentencia `USE`):

```
SHOW TABLES;             # lista las tablas disponibles
```

Acceso a MySQL desde PHP

En PHP disponemos de un envoltorio de la API nativa de MySQL llamado mysqli. Para establecer una conexión a la base de datos podemos crear un objeto de tipo `mysqli`, cuyo constructor toma la ubicación del servidor, el nombre de usuario, la contraseña y la base de datos preseleccionada:

```
$conexion = new mysqli('localhost', 'root', '', 'ejemplo');
// servidor en máquina local, usuario por omisión y sin contraseña
// la base de datos debe estar creada con anticipación
```

En caso de que no preseleccionemos una base de datos o deseemos seleccionar otra, podemos hacer lo siguiente:

```
$conexion->select_db('ejemplo2');       // cambio de base de datos
```

Dada la siguiente relación:

```
CREATE TABLE persona (
   nombre TEXT,
   apellido TEXT,
   edad INT
);
```

Nosotros podemos insertar filas desde PHP de la siguiente forma:

```
// enviar una petición que no depende de variables
$conexion->query("INSERT INTO persona (nombre, apellido, edad)
VALUES ('pedro', 'perez', 20)");

$nombre = 'ana';
$apellido = 'gomez';
$edad = 25;

// enviar una petición que sí depende de variables
// las variables se incrustan en la cadena petición
$conexion->query("INSERT INTO persona (nombre, apellido, edad)
VALUES ('$nombre', '$apellido', $edad)");
```

Es importante fijarnos en que los parámetros que correspondan a literales de cadenas en una petición deben ir delimitados por comillas (como nosotros usamos comillas dobles para PHP entonces usamos comillas sencillas para MySQL). En caso de duda sobre cómo se está enviando una petición, sugerimos hacer lo siguiente:

```
$peticion = "INSERT INTO persona (nombre, apellido, edad) VALUES
('$ana', '$gomez', $edad)";
echo $peticion;          // ver la petición en la salida
$conexion->query($peticion);       // enviar la petición
```

Una petición devuelve `false` en caso de error y `true` en caso de éxito cuando sea `INSERT`, `UPDATE` o `DELETE`. Por ejemplo:

```
var_dump($conexion->query("asdad"));              // bool(false)
```

```
var_dump($conexion->query("DELETE FROM persona"));   // bool(true)
```

Una petición `SELECT` devuelve un objeto de tipo `mysqli_result` en caso de éxito. Este objeto permite controlar el modo en que se realiza la lectura desde PHP de las relaciones devueltas por MySQL: ya que algunas relaciones pueden ser muy grandes, podemos realizar la lectura fila a fila o solicitar todo el resultado en un solo paso:

```
$r1 = $conexion->query("SELECT nombre, apellido FROM persona");
$arr1 = $conexion->fetch_all(MYSQLI_ASSOC); // todo el resultado

foreach ($arr1 as $tupla) {        // para cada tupla del resultado
   echo $tupla["nombre"], " ";          // imprime el nombre
   echo $tupla["apellido"], "\n";       // imprime la edad
}

// volvemos a hacer la petición y leeremos fila a fila
$r2 = $conexion->query("SELECT nombre, apellido FROM persona");

for (;;) {
   $tupla = $r2->fetch_assoc( );        // solicita una tupla

   if ($tupla == null) {
      break;                            // ya no hay tuplas
   }
                                        // sí hubo tupla
   echo $tupla["nombre"], " ";          // imprime el nombre
   echo $tupla["apellido"], "\n";       // imprime la edad
}
```

Seguridad en peticiones PHP

Cuando queremos enviar una variable que es una cadena en una petición, pueden surgir problemas:

```
$nombre = "wendy's";
$conexion->query("INSERT INTO restaurante (nombre) VALUES
('$nombre')");      // ¡error!
```

Para ver cuál es el error, veamos cómo se incrustó la variable `$nombre` en la cadena:

```
INSERT INTO restaurante (nombre) VALUES ('Wendy's')
```

El problema ahora es claro: la comilla incluida dentro de la propia variable está interactuando con la comilla de la petición en sí. Esto puede traer consigo problemas de seguridad, pues una persona mal intencionada puede aprovechar que tiene control sobre la sintaxis de la petición y así inyectar instrucciones que nosotros no deseábamos. Sin entrar a más detalles, podemos evitar este problema fácilmente con la función `escape_string` de `mysqli`:

```
$nombre = $conexion->escape_string("Wendy's");
$conexion->query("INSERT INTO restaurante (nombre) VALUES
('$nombre')");      // OK
```

Siempre que se quieran evitar problemas con una conexión a una base de datos MySQL, se debe dar preferencia a esta función sobre cualquier otra similar.

Columnas únicas

En algunas relaciones es de esperar que sólo exista una única tupla con los mismos valores en uno o varios atributos. Por ejemplo:

```
INSERT INTO alumnos (matricula, nombre) VALUES (203305906,
"rodrigo"); # OK
INSERT INTO alumnos (matricula, nombre) VALUES (214405999,
"rodrigo"); # OK, mismo nombre pero diferente matrícula
INSERT INTO alumnos (matricula, nombre) VALUES (214405999,
"pablo");   # alumno con otro nombre pero misma matrícula
            # seguramente es un error
```

Nosotros podemos informarle a MySQL qué atributos de una relación deben ser únicos, es decir, que no debe existir más de una tupla que coincida en esos valores. MySQL realizará la verificación antes insertar una tupla:

```
CREATE TABLE alumnos (
   matricula INT,
   nombre TEXT,
   UNIQUE (matricula)    // no deben haber matrículas repetidas
);

INSERT INTO alumnos (matricula, nombre) VALUES (203305906,
"rodrigo"); # OK
INSERT INTO alumnos (matricula, nombre) VALUES (214405999,
"rodrigo"); # OK, mismo nombre pero diferente matrícula
INSERT INTO alumnos (matricula, nombre) VALUES (214405999,
"pablo");   # error, esta fila no se inserta y MySQL nos avisa
```

Una restricción UNIQUE puede referirse a más de un atributo:

```
CREATE TABLE puntos (
   x INT,
   y INT,
   UNIQUE (x, y)    # la pareja debe ser única, no necesariamente
                    # los atributos individuales
);

INSERT INTO puntos (x, y) VALUES (5, 2);  # OK
INSERT INTO puntos (x, y) VALUES (9, 2);  # OK
INSERT INTO puntos (x, y) VALUES (5, 7);  # OK
INSERT INTO puntos (x, y) VALUES (5, 2);  # error, (5, 2) repetido
```

Valores nulos y valores por omisión

Es posible insertar el valor nulo (NULL en MySQL) en una tupla:

```
CREATE TABLE personas (
   nombre TEXT,
   edad INT
);

INSERT INTO personas (nombre, edad) VALUES ("pablo", NULL);
# no tenemos la edad de "Pablo"
```

A primera vista, pareciera que `NULL` significa "el dato no existe". Si quisiéramos calcular los nombres de las personas de las que no conocemos la edad, lo primero que intentaríamos hacer sería:

```
SELECT nombre FROM personas WHERE edad = NULL;
```

Sin embargo, ¡la consula anterior devuelve una relación sin tuplas! Más aún, lo siguiente se permite:

```
CREATE TABLE alumnos (
   matricula INT,
   nombre TEXT,
   UNIQUE (matricula)
);

INSERT INTO alumnos (matricula, nombre) VALUES (NULL, "juan");
# OK pero extraño: alumno sin matrícula
INSERT INTO alumnos (matricula, nombre) VALUES (NULL, "juan");
# ¡también OK a pesar de que hay una tupla idéntica!
```

¿Por qué sucede esto? Porque `NULL` no significa ausencia del dato, sino *dato desconocido*. Necesitaremos estudiar un poco de lógica tribooleana (Verdadero, Falso y Desconocido) para entender lo que está pasando:

A	B	A = B	A ≠ B	A ∧ B	A ∨ B
F	F	V	F	F	F
F	V	F	V	F	V
V	F	F	V	F	V
V	V	V	F	V	V
F	D	D	D	F	D
D	F	D	D	F	D
V	D	D	D	D	V
D	V	D	D	D	V
D	D	D	D	D	D

En pocas palabras, el resultado de comparar igualdad entre dos "desconocidos" da un resultado desconocido. Debemos recordar que un predicado o condición de selección en SQL sólo incluye una tupla si la condición es verdadera. Si la condición es desconocida *ésta no se incluye*.

Ya que el criterio de `UNIQUE` para detectar tuplas duplicadas es la comparación de igualdad (y la comparación entre valores donde alguno es `NULL` no es verdadera), `NULL` en un atributo restringido por `UNIQUE` es capaz de ignorar esta restricción.

Nosotros podemos indicarle a MySQL que no admita nulos en un atributo, o bien, ser explícitos en que sí los admita, de la siguiente forma:

```
CREATE TABLE alumnos (
   matricula INT NOT NULL,    # no puede ser nulo
   nombre TEXT NULL,          # sí puede ser nulo
   UNIQUE (matricula)
);

INSERT INTO alumnos (matricula, nombre) VALUES (203305906, NULL);
INSERT INTO alumnos (matricula, nombre) VALUES (NULL, "juan");
```

Un atributo que puede ser NULL puede no especificarse al momento de la inserción:

```
INSERT INTO alumnos (matricula) VALUES (214405999);
# OK, nombre nulo
```

Una manera más explícita de indicar valores por omisión es de la siguiente manera:

```
CREATE TABLE alumnos (
   matricula INT NOT NULL,    # no puede ser nulo
   nombre DEFAULT NULL,       # sí puede ser nulo y es por omisión
   UNIQUE (matricula)
);
```

Pueden usarse otros valores por omisión, pero no lo recomendamos.

Llaves primarias

Una llave primaria puede formarse por un conjunto de atributos que son `UNIQUE` y deben ser `NOT NULL`. Por ejemplo, la relación anterior podría quedar de la siguiente forma:

```
CREATE TABLE alumnos (
   matricula INT NOT NULL,
   nombre DEFAULT NULL,
   PRIMARY KEY (matricula)
);
```

Sólo se permite una llave primaria por relación. MySQL sobreentiende que una llave primaria contiene atributos distintivos de cada tupla y que muchas de las peticiones sobre la relación harán uso de los mismos. Por esta razón, MySQL optimiza el acceso a los datos cuando se haga una consulta que haga referencia a los atributos de una llave primaria:

```
SELECT * FROM alumnos WHERE matricula = 203305906;
# consulta optimizada: matricula es una llave primaria
SELECT * FROM alumnos WHERE nombre = "pablo"
# consulta lenta: seguramente búsqueda lineal sobre las tuplas
```

¿Cómo se optimizan las consultas que usen una llave primaria? Depende: si en nuestro ejemplo garantizamos que las matrículas insertadas sucesivamente en la base de datos vienen en orden,

entonces MySQL podría almacenar la tabla ordenada en disco y una consulta podría ejecutar algo similar a búsqueda binaria (en lugar de búsqueda lineal).
En ocasiones una relación no tiene una llave primaria natural. Por ejemplo:

```
CREATE TABLE mascotas (
   nombre TEXT NOT NULL,      # puede haber nombres repetidos
   raza TEXT NOT NULL         # puede haber razas repetidas
);
```

Nosotros podemos crear un atributo ficticio que sirva de llave primaria:

```
CREATE TABLE mascotas (
   id INT NOT NULL,           # ficticio: identificador de mascota
   nombre TEXT NOT NULL,      # puede haber nombres repetidos
   raza TEXT NOT NULL,        # puede haber razas repetidas
   PRIMARY KEY (id)
);

INSERT INTO mascotas (id, nombre, raza) VALUES (1, "fido",
"perro");
INSERT INTO mascotas (id, nombre, raza) VALUES (2, "neko",
"gato");
INSERT INTO mascotas (id, nombre, raza) VALUES (3, "peluso",
"perro");
INSERT INTO mascotas (id, nombre, raza) VALUES (4, "peluso",
"gato");

SELECT * FROM mascotas WHERE nombre = "neko";
# consulta lenta: seguramente búsqueda lineal sobre las tuplas
SELECT * FROM mascotas WHERE id = 2
# consulta optimizada: id es una llave primaria
```

Ya que el atributo `id` es un atributo ficticio, nosotros pudimos haber insertado identificadores arbitrarios (pero es natural insertarlos en orden: 1, 2, 3, etc). En caso de que este comportamiento sea el indicado, podemos pedirle a MySQL que autogenere el identificador por nosotros:

```
CREATE TABLE mascotas (
   id INT NOT NULL AUTO_INCREMENT      # autocalculado
   nombre TEXT NOT NULL,
   raza TEXT NOT NULL,
   PRIMARY KEY (id)
);

INSERT INTO mascotas (nombre, raza) VALUES ("fido", "perro");
INSERT INTO mascotas (nombre, raza) VALUES ("neko", "gato");
INSERT INTO mascotas (nombre, raza) VALUES ("peluso", "perro");
INSERT INTO mascotas (nombre, raza) VALUES ("peluso", "gato");
```

Llaves foráneas

Suponga que deseamos modelar los siguientes `structs` de C++:

```
struct persona {
   std::string nombre;
};

struct perro {
   std::string nombre;
   persona* dueño;       // suponer que es válido usar 'ñ'
};

persona p1 = { "pablo" };     // una persona llamada "pablo"
persona p2 = { "pablo" };     // un "pablo" diferente
// &p1 != p2

perro r1 = { "fido", &p1 };
perro r2 = { "milo", &p1 };
perro r3 = { "rufo", &p2 };
// r1.dueño != r3.dueño
```

Ya que el atributo `nombre` de `persona` no sirve para distinguir entre dos personas diferentes (y que se llamen igual), necesitamos añadir un atributo `id` ficticio como lo hicimos anteriormente:

```
CREATE TABLE persona (
   id INT NOT NULL AUTO_INCREMENT,
   nombre TEXT NOT NULL,
   PRIMARY KEY (id)
);

INSERT INTO persona (nombre) VALUES ("pablo");    # id = 1
INSERT INTO persona (nombre) VALUES ("pablo");    # id = 2
```

Para expresar el `struct perro` en SQL nosotros no podemos usar apuntadores, pero sí podríamos usar el `id` único de la persona que es dueño del perro:

```
CREATE TABLE perro (
   nombre TEXT NOT NULL,
   dueño INT DEFAULT NULL     # puede no tener dueño
);

INSERT INTO perro (nombre, dueño) VALUES ("fido", 1);
# el dueño es la persona 1 (el primer "pablo")
INSERT INTO perro (nombre, dueño) VALUES ("milo", 1);
# el dueño es la persona 1 (el primer "pablo")
INSERT INTO perro (nombre, dueño) VALUES ("rufo", 2);
# el dueño es la persona 2 (el segundo "pablo")
```

Desafortunadamente, MySQL permitiría lo siguiente:

```
INSERT INTO perro (nombre, dueño) VALUES ("snoopy", -5);
# el dueño es la persona -5, ¡pero no hay persona con id = -5!
```

Al igual que con las restricciones UNIQUE, podemos pedirle a MySQL que verifique que la persona que es dueño del perro verdaderamente exista usando una llave foránea:

```
CREATE TABLE perro (
   nombre TEXT NOT NULL,
   dueño INT DEFAULT NULL,
   FOREIGN KEY (dueño) REFERENCES persona (id)
);

INSERT INTO perro (nombre, dueño) VALUES ("snoopy", -5);
# ahora debería ser un error, ¡pero MySQL lo sigue aceptando!
```

Desafortunadamente, el motor de almacenamiento por omisión de MySQL es MyISAM. Este motor permite especificar llaves foráneas, pero no realiza la verificación. El motor de almacenamiento InnoDB sí realiza la verificación:

```
CREATE TABLE persona (
   id INT NOT NULL AUTO_INCREMENT,
   nombre TEXT NOT NULL,
   PRIMARY KEY (id)
) ENGINE=InnoDB;

CREATE TABLE perro (
   nombre TEXT NOT NULL,
   dueño INT DEFAULT NULL,
   FOREIGN KEY (dueño) REFERENCES persona (id)
) ENGINE=InnoDB;

INSERT INTO perro (nombre, dueño) VALUES ("snoopy", -5);
# error, la persona con id = -5 no existe y la fila no se inserta
```

Las dos tablas (tanto la tabla que hace referencia como la tabla referida) deben tener el motor de almacenamiento InnoDB.

Acciones cuando una tabla referida es modificada

Supongamos que se tienen las siguientes relaciones y filas:

```
CREATE TABLE persona (
   id INT NOT NULL AUTO_INCREMENT,
   nombre TEXT NOT NULL,
   PRIMARY KEY (id)
) ENGINE=InnoDB;

CREATE TABLE perro (
   nombre TEXT NOT NULL,
   dueño INT DEFAULT NULL,
   FOREIGN KEY (dueño) REFERENCES persona (id)
) ENGINE=InnoDB;

INSERT INTO persona (nombre) VALUES ("pablo");
INSERT INTO persona (nombre) VALUES ("pedro");

INSERT INTO perro (nombre, dueño) VALUES ("fido", 1);
INSERT INTO perro (nombre, dueño) VALUES ("milo", 1);
```

```
INSERT INTO perro (nombre, dueño) VALUES ("rufo", 2);
```

Ahora supongamos que deseamos eliminar a `pablo` de la base de datos:

```
DELETE FROM persona WHERE nombre = "pablo";
```

¿Qué debe pasar con los registros de la relación `perro` que dependen de la tupla de `pablo`? Existen varias opciones:

- Ya que un perro podría no tener dueño, todos los perros de `pablo` podrían quedarse sin dueño (dueño = NULL) y permanecer en la base de datos:

```
CREATE TABLE perro (
   nombre TEXT NOT NULL,
   dueño INT DEFAULT NULL,
   FOREIGN KEY (dueño) REFERENCES persona (id) ON DELETE SET NULL
) ENGINE=InnoDB;
```

- En lugar de conservar los registros de los perros de `pablo`, podríamos eliminarlos junto con `pablo` mismo:

```
CREATE TABLE perro (
   nombre TEXT NOT NULL,
   dueño INT DEFAULT NULL,
   FOREIGN KEY (dueño) REFERENCES persona (id) ON DELETE CASCADE
) ENGINE=InnoDB;
```

En caso de no usar alguna de las dos opciones presentadas anteriormente, la eliminación de `pablo` fallará pues aún existen tuplas que depende de él (en este caso las de sus perros). Ambas opciones se pueden especificar también cuando el `id` de pablo cambie (pues los perros hacen referencia a `pablo` mediante su `id`, no mediante su nombre). Por ejemplo:

```
CREATE TABLE perro (
   nombre TEXT NOT NULL,
   dueño INT DEFAULT NULL,
   FOREIGN KEY (dueño) REFERENCES persona (id) ON UPDATE SET NULL
) ENGINE=InnoDB;
# un perro se queda sin dueño si el id de la persona cambia
```

El que una llave primaria sea modificada puede ser una señal de que la base de datos está mal diseñada. Por la misma razón, no recomendamos el uso de `ON UPDATE`.

Modificar atributos y restricciones en tablas preexistentes

Supongamos que tenemos la siguiente relación:

```
CREATE TABLE perro (
   nombre TEXT NOT NULL,
   dueño INT DEFAULT NULL,
   FOREIGN KEY (dueño) REFERENCES persona (id) ON UPDATE SET NULL
) ENGINE=InnoDB;
```

Para agregar un nuevo atributo `edad` a la tabla, podemos usar la siguiente sintaxis:

```
ALTER TABLE perro ADD edad INT NOT NULL;
```

Si el atributo es no nulo, entonces todas las tuplas pasarán a tener un valor por omisión para el nuevo atributo (0 para enteros, la cadena vacía para cadenas). Eliminar un atributo también es sencillo:

```
ALTER TABLE perro DROP nombre;
```

También es posible modificar un atributo:

```
ALTER TABLE perro CHANGE edad viejo INT DEFAULT NULL;
```

El atributo anteriormente llamado edad ahora se llama viejo. Si no se desea cambiar el nombre puede repetirse éste o bien, usar MODIFY:

```
ALTER TABLE perro CHANGE viejo viejo INT DEFAULT NULL;
ALTER TABLE perro MODIFY viejo INT DEFAULT NULL;
```

También podemos agregar restricciones:

```
ALTER TABLE perro ADD UNIQUE(viejo);
```

¿Cómo eliminar una restricción? Necesitamos eliminarla en base al nombre que se le asigna (ya sea un nombre especificado por el programador durante la creación de la base de datos o el nombre que MySQL le asigna automáticamente). Para darle un nombre a una restricción durante la definición de tabla, podemos usar la palabra `CONSTRAINT` seguida del nombre:

```
CREATE TABLE perro (
   nombre TEXT NOT NULL,
   dueño INT DEFAULT NULL,
   pulgas INT DEFAULT NULL,
   CONSTRAINT u1 UNIQUE (pulgas),
   CONSTRAINT c1 FOREIGN KEY (dueño)
      REFERENCES persona (id) ON UPDATE SET NULL
) ENGINE=InnoDB;
```

Eliminar la restricción entonces es simple:

```
ALTER TABLE perro DROP INDEX u1;
ALTER TABLE perro DROP FOREIGN KEY c1;
```

En caso de que no se hayan especificado nombres de restricciones, podemos usar la siguiente consulta para obtenerlos:

```
SHOW INDEX IN perro;
```

La sentencia anterior devuelve una relación con tantas tuplas como restricciones haya. El nombre de cada restricción está en la columna `key_name`.

Normalización en bases de datos

Diseñar una base de datos relacional es ocasionalmente complicado debido a las limitaciones del modelo. Edgar Codd (el inventor de la teoría) establece varios niveles de "normalización" que una base de datos puede tener: entre mayor el nivel de normalización "más cerca del ideal teórico" está (sin embargo, el ideal teórico no siempre es buena idea en la práctica). Los niveles de normalización son los siguientes:

Primera forma normal

Dada una tupla, cada uno de sus elementos debe corresponder con un único valor del dominio. Por ejemplo, la siguiente relación no está en la primera forma normal:

alumno		
matricula: entero	nombre: cadena	telefono: cadena
201501	juan	1234-5678 / 5529-9999
201502	jorge	4321-8765

La primera forma normal no se cumple en esta relación pues en el atributo `telefono` (que es una cadena) se están almacenando dos teléfonos de `juan` en la misma celda. Sin modificar el esquema de la relación, podemos llevar a ésta a la primera forma normal:

alumno		
matricula: entero	nombre: cadena	telefono: cadena
201501	juan	1234-5678
201501	juan	5529-9999
201502	jorge	4321-8765

Sin embargo, como puede observarse, el querer almacenar los dos teléfonos de `juan`, uno por fila, nos obliga a repetir información.

Segunda forma normal

Dada la siguiente relación:

alumno		
matricula: entero	nombre: cadena	telefono: cadena
201501	juan	1234-5678
201501	juan	5529-9999
201502	jorge	4321-8765

El atributo `matricula` por sí solo puede formar una llave (la matrícula es única y representa a un alumno de manera inequívoca).

A nosotros nos gustaría tener una única fila por alumno, pero actualmente tenemos dos para el alumno 201501. El atributo nombre no es el problema: el alumno con matrícula 201501 siempre

será `juan` y el alumno con matrícula 201502 siempre será `jorge` (y en este ejemplo, un alumno tiene un único nombre); el culpable de que tengamos múltiples filas para un mismo alumno es el atributo `telefono` (un alumno puede tener múltiples teléfonos).

Una tabla está en segunda forma normal si no existen atributos que obliguen a que una llave candidata no mínima (con subconjuntos que también sean llaves candidatas) aparezca más de una vez en una tabla. En este caso, tendremos que sacar el atributo `hobby`:

alumno	
matricula: entero	nombre: cadena
201501	juan
201502	jorge

telefonos	
matricula: entero	telefono: cadena
201501	1234-5678
201501	5529-9999
201502	4321-8765

Una ventaja que tiene esta forma es que actualizar el nombre de un alumno sólo requeriría modificar una única tabla.

Tercera forma normal

Dada la siguiente relación:

alumno			
matricula: entero	nombre: cadena	carrera: cadena	creditos: entero
201501	juan	computacion	491
201502	jorge	computacion	491

La información de los créditos de la carrera es dependiente de computación, y es independiente de las claves de la relación. Podemos factorizar lo anterior.

alumno		
matricula: entero	nombre: cadena	carrera: cadena
201501	juan	computacion
201502	jorge	computacion

carrera	
nombre: cadena	créditos: entero
computacion	491

Una relación está en tercera forma normal si ningún atributo no-clave depende de algún otro atributo no-clave. En la relación `carrera` entonces `nombre` podría volverse una clave, con lo cual ambas relaciones cumplirían la tercera forma normal. En caso de que los créditos de la carrera cambien, sólo sería necesario modificar una única fila.

Forma Boyce-Codd; cuarta y quinta formas normales

Consisten en encontrar patrones de dependencia que a primera vista pueden no ser inmediatos (y factorizar nuevamente, tal vez teniendo que incrementar la complejidad de las relaciones resultantes). Nosotros no hablaremos más al respecto, pero debemos recordar al lector que el tema de normalización no debe verse únicamente desde un punto de vista teórico: en general debe usarse el sentido común para encontrar un buen diseño de bases de datos.

Existen algunas relaciones que no pueden llevarse a formas más altas de normalización usando únicamente la teoría relacional. En SQL podemos indicar verificaciones adicionales con la restricción CHECK:

```
CREATE TABLE puntos (
   x INT NOT NULL,
   y INT NOT NULL,

   CHECK(x < y)
);
```

Desafortunadamente, ningún motor de almacenamiento de MySQL las respeta.

Índices

Aunque es frecuente buscar un alumno por matrícula, ¿qué pasa si queremos buscar un alumno por nombre? Seguramente será más lento (el nombre es una cadena), sin embargo hay una razón adicional: la matrícula es una llave primaria, mientras que el nombre no puede ser llave ni `UNIQUE`.

Para poder indicarle a MySQL que deseamos crear un índice para el nombre de un alumno (aunque no sea UNIQUE), podríamos intentar lo siguiente usando la palabra `INDEX`:

```
CREATE TABLE alumnos (
   matricula INT NOT NULL,
   nombre TEXT NOT NULL,

   PRIMARY KEY (id),
   INDEX (nombre)             # ¡error!
);
```

Sin embargo, cuando queremos crear un índice sobre un atributo que es una cadena de longitud variable (el nombre de los alumnos no tiene por qué tener el mismo número de letras), hay problemas:

- Generalmente un índice es implementado con un árbol B o un árbol B+ (este último permite realizar búsquedas de intervalos eficientemente).
- El tamaño de los nodos de los árboles B o B+ es (generalmente) fijo y depende del tamaño del bloque que use la plataforma para acceder a los datos del disco (ej. 4096 bytes).

- ¿Cuántos elementos (cadenas `TEXT` en nuestro caso) caben por nodo si desconocemos la longitud de la cadena?

Cuando usemos `INDEX` sobre cadenas de longitud variable, debemos indicar cuántos bytes debe usar MySQL para crear el índice: cadenas que coincidan en esa cantidad de bytes son equivalentes para propósitos del índice:

```
CREATE TABLE alumnos (
   matricula INT NOT NULL,
   nombre TEXT NOT NULL,

   PRIMARY KEY (id),
   INDEX (nombre(16))         # ok
);
```

Para atributos de ancho fijo como `INT`, indicar la longitud en bytes de los valores del índice completamente innecesario:

```
CREATE TABLE puntos (
   x INT NOT NULL,
   y INT NOT NULL,

   INDEX (x),
   INDEX (y)
);
```

Revisión de llaves foráneas, orden de declaraciones y desactivación temporal de revisión

Es necesario tener cuidado en el orden de definición de tablas cuando se estén usando llaves foráneas, por ejemplo:

```
CREATE TABLE personas (
   id INT NOT NULL AUTO_INCREMENT,
   nombre TEXT NOT NULL,

   PRIMARY KEY (id)
) ENGINE=InnoDB;

CREATE TABLE telefonos (
   persona INT NOT NULL,
   telefono TEXT NOT NULL,

   FOREIGN KEY (persona) REFERENCES personas (id)
) ENGINE=InnoDB;
```

En el código anterior, sería un error definir primero la tabla `telefonos`, pues ésta depende de la tabla `personas` (así que `personas` debe definirse primero, tal como se hizo).

Ahora supongamos que queremos recrear las tablas por alguna razón (sólo esas tablas, sin borrar la base de datos completa) e intentamos hacer lo siguiente:

```
DROP TABLE IF EXISTS personas;          # agregamos el DROP TABLE
CREATE TABLE personas (
   id INT NOT NULL AUTO_INCREMENT,
   nombre TEXT NOT NULL,

   PRIMARY KEY (id)
) ENGINE=InnoDB;

DROP TABLE IF EXISTS telefonos;         # agregamos el DROP TABLE
CREATE TABLE telefonos (
   persona INT NOT NULL,
   telefono TEXT NOT NULL,

   FOREIGN KEY (persona) REFERENCES personas (id)
) ENGINE=InnoDB;
```

Lo anterior no funcionará: la tabla telefonos depende de personas, por lo que es un error querer eliminar la tabla personas mientras telefonos siga haciendo referencia a ella. Por supuesto, una manera de primero hacer todos DROP (en el orden inverso en el que se definieron las tablas) y luego crear las tablas. Una manera más sencilla es apagar temporalmente la revisión de llaves foráneas:

```
SET FOREIGN_KEY_CHECKS = 0;             # apagamos la revisión

DROP TABLE IF EXISTS personas;          # agregamos el DROP TABLE
CREATE TABLE personas (
   id INT NOT NULL AUTO_INCREMENT,
   nombre TEXT NOT NULL,

   PRIMARY KEY (id)
) ENGINE=InnoDB;

DROP TABLE IF EXISTS telefonos;         # agregamos el DROP TABLE
CREATE TABLE telefonos (
   persona INT NOT NULL,
   telefono TEXT NOT NULL,

   FOREIGN KEY (persona) REFERENCES personas (id)
) ENGINE=InnoDB;

SET FOREIGN_KEY_CHECKS = 1;             # prendemos la revisión
```

Para los siguientes ejemplos usaremos la siguiente base de datos:

```
USE ejemplo;
SET FOREIGN_KEY_CHECKS = 0;

DROP TABLE IF EXISTS carreras;
CREATE TABLE carreras (
   id INT NOT NULL AUTO_INCREMENT,
   nombre TEXT NOT NULL,
   correo TEXT NOT NULL,
```

```
   PRIMARY KEY (id),
   UNIQUE (nombre(16))
) ENGINE=InnoDB;

DROP TABLE IF EXISTS alumnos;
CREATE TABLE alumnos (
   matricula INT NOT NULL,
   nombre TEXT NOT NULL,
   apellido TEXT NOT NULL,
   carrera INT NOT NULL,
   estado INT NOT NULL,
   creditos INT NOT NULL,

   PRIMARY KEY (matricula),
   INDEX (apellido(16)),
   FOREIGN KEY (carrera) REFERENCES carreras (id)
) ENGINE=InnoDB;

DROP TABLE IF EXISTS telefonos;
CREATE TABLE telefonos (
   matricula INT NOT NULL,
   telefono TEXT NOT NULL,

   FOREIGN KEY (matricula) REFERENCES alumnos (matricula)
    ON DELETE CASCADE
) ENGINE=InnoDB;

SET FOREIGN_KEY_CHECKS = 1;
```

Las definiciones anteriores y una secuencia de inserciones pueden conseguirse en la página web del curso. En general es más rápido hacer múltiples inserciones en un solo `INSERT` que hacer una única inserción por `INSERT`: MySQL guarda los datos en disco para cada sentencia; una inserción por `INSERT` hará un acceso a disco por fila mientras que con múltiples inserciones por `INSERT` MySQL intentará guardar en disco todas esas filas en la menor cantidad de accesos a disco. MySQL también tiene un límite de "paquete" (un límite en la longitud en bytes de una sentencia individual) por lo que no será posible insertar millones de filas en un único `INSERT`.

Ordenamientos, límites y paginados

De la base de datos de ejemplo, ¿cómo podemos listar las matrículas, nombres y apellidos de los alumnos, ordenados por matrícula?

```
SELECT matricula, nombre, apellido FROM alumnos ORDER BY matricula;
```

La cláusula `ORDER BY` permite especificar una lista de atributos por el cual ordenar. Por omisión, la definición es ascendente.

¿Qué pasa si queremos listarlos por orden por créditos inscritos (de mayor a menor) y en caso de empate, desempatar por matrícula (de menor a mayor)?

```
SELECT matricula, nombre, apellido FROM alumnos
   ORDER BY creditos DESC, matricula ASC;
```

Cuando se lista más de un atributo en `ORDER BY`, se da preferencia al primero. La palabra `DESC` indica orden descendente y la palabra `ASC`, ascendente. La palabra `ASC` puede omitirse (recordemos que es el ordenamiento por omisión) pero suele ser buena idea ser explícitos para claridad del código.

¿Qué pasa si queremos listar los primeros 15 alumnos por orden por créditos inscritos (de mayor a menor)? Podemos usar LIMIT:

```
SELECT matricula, nombre, apellido FROM alumnos
   ORDER BY creditos LIMIT 15;
```

La cláusula `LIMIT` *`N`* le indica a MySQL que puede parar el cálculo y la entrega de resultados una vez encontradas las primeras *`N`* filas del resultado. ¿Qué pasa si queremos el segundo grupo de 15 alumnos por orden de créditos inscritos?

```
SELECT matricula, nombre, apellido FROM alumnos
   ORDER BY creditos
   LIMIT 15 OFFSET 15;        # ignorar los 15 primeros
```

La palabra `OFFSET` indica el desplazamiento relativo (en número de filas) con respecto al resultado que hubiera sido calculado sin usar `LIMIT`. Calcular el tercer grupo de alumnos también es fácil:

```
SELECT matricula, nombre, apellido FROM alumnos
   ORDER BY creditos
   LIMIT 15 OFFSET 30;        # ignorar los 30 primeros
```

Una sintaxis alternativa es:

```
SELECT matricula, nombre, apellido FROM alumnos
   ORDER BY creditos
   LIMIT 30, 15;
```

Creemos que la primera sintaxis es más clara.

Para la creación de paginados, a veces es útil saber cuántas filas hubiera tenido el resultado si no se hubiera usado LIMIT (por ejemplo, si hay 43 resultados en total y por página se muestran 10, entonces se necesitarán 5 páginas). Esto se puede hacer de la siguiente manera:

```
SELECT SQL_CALC_FOUND_ROWS
   matricula, nombre, apellido FROM alumnos
   ORDER BY creditos
   LIMIT 30, 15;

SELECT FOUND_ROWS( );    # regresa el total de filas que hubiera
                         # tenido el SELECT anterior sin LIMIT
```

Por cuestiones de rendimiento, no recomendamos el uso de esta característica.

Producto cruz y junturas

Dadas las siguientes filas en las relaciones de la base de datos de ejemplo:

carreras		
id	nombre	correo
1	comp	comp@uam.mx
2	mate	mate@uam.mx

alumnos					
matricula	nombre	apellido	carrera	estado	creditos
201501	pablo	perez	1	1	50
201502	juan	gomez	2	2	0

El producto cruz de las tablas puede calcularse de la siguiente manera:

```
SELECT * FROM carreras, alumnos
```

y tendrá por resultado:

SELECT * FROM carreras, alumnos								
id	nombre	correo	matricula	nombre	apellido	carrera	estado	creditos
1	comp	comp@uam.mx	201501	pablo	perez	1	1	50
1	comp	comp@uam.mx	201502	juan	gomez	2	2	0
2	mate	mate@uam.mx	201501	pablo	perez	1	1	50
2	mate	mate@uam.mx	201502	juan	gomez	2	2	0

Primera cosa que hay que notar: ¡hay dos columnas nombre! MySQL permite nombres de columnas duplicados, por lo que no es necesario el renombramiento. ¿Qué debemos hacer si queremos proyectar de la tabla anterior ambos nombres? tendríamos que especificar de qué tabla vienen:

```
SELECT carreras.nombre, alumnos.nombre FROM carreras, alumnos
```

SELECT carreras.nombre, alumnos.nombre FROM carreras, alumnos	
nombre	nombre
comp	pablo
comp	juan
mate	pablo
mate	juan

o bien, usar renombramiento durante la proyección:

```
SELECT
   carreras.nombre AS nombre_carrera,
   alumnos.nombre AS nombre_alumno
```

```
   FROM carreras, alumnos;
```

¿Cómo obtener la matrícula y el nombre de la carrera que está cursando cada alumno? En el producto cruz, hay filas que incluyen datos de alumnos con datos de una carrera que no es la suya; de alguna manera debemos quitar estas filas:

SELECT * FROM carreras, alumnos								
id	nombre	correo	matricula	nombre	apellido	carrera	estado	creditos
1	comp	comp@uam.mx	201501	pablo	perez	**1**	1	50
~~1~~	~~comp~~	~~comp@uam.mx~~	~~201502~~	~~juan~~	~~gomez~~	~~2~~	~~2~~	~~0~~
~~2~~	~~mate~~	~~mate@uam.mx~~	~~201501~~	~~pablo~~	~~perez~~	~~1~~	~~1~~	~~50~~
2	mate	mate@uam.mx	201502	juan	gomez	**2**	2	0

Lograrlo es sencillo: las filas que nos interesan deben cumplir la condición

```
carreras.id = alumnos.carrera
```

es decir, las filas donde la información del alumno coincide con la información de su carrera:

```
SELECT matricula, carreras.nombre AS nombre_carrera
   FROM carreras, alumnos WHERE carreras.id = alumnos.carrera;
```

SELECT matricula, carreras.nombre AS nombre_carrera FROM carreras, alumnos WHERE carreras.id = alumnos.carrera	
matricula	nombre_carrera
201501	comp
201502	mate

Este es un ejemplo θ-juntura (`carreras` $\bowtie_{\texttt{carreras.id=alumnos.carrera}}$ `alumnos`). También es llamada simplemente juntura interna. La juntura natural hubiera requerido que ambos atributos se llamaran igual, lo cual no es cierto en este caso: uno se llama `id` (el de `carreras`) y el otro `carrera` (el de `alumnos`).

Supongamos que ahora tenemos las siguientes tablas:

carreras		
id	nombre	correo
1	comp	comp@uam.mx
2	mate	mate@uam.mx

alumnos					
matricula	nombre	apellido	carrera	estado	creditos
201501	pablo	perez	1	1	50

En este ejemplo no hay alumnos que cursen la carrera 2. La juntura externa izquierda (`LEFT OUTER JOIN`) actúa sobre dos tablas y un par de atributos (que se espera que coincidan). Las filas

del lado izquierdo aparecen en el resultado y si no hay filas que coincidan del lado derecho, se llenan con nulos.

```
SELECT * FROM carreras LEFT OUTER JOIN alumnos ON carreras.id = alumnos.carrera
```

SELECT * FROM carreras LEFT OUTER JOIN alumnos ON carreras.id = alumnos.carrera								
id	nombre	correo	matricula	nombre	apellido	carrera	estado	creditos
1	comp	comp@uam.mx	201501	pablo	perez	1	1	50
2	mate	mate@uam.mx	NULL	NULL	NULL	NULL	NULL	NULL

La `RIGHT OUTER JOIN` es similar pero dando preferencia a la tabla derecha (y rellenando con nulos cuando no hay coincidencias del lado izquierdo).

La `FULL OUTER JOIN` es similar y no da preferencia a ninguna tabla (aparecen los resultados de ambas, rellenando con nulos cuando no haya coincidencias con la otra).

Operaciones de conjuntos y peticiones anidadas

La unión de dos relaciones puede realizarse con el operador UNION:

```
SELECT * FROM alumnos WHERE creditos <= 45
   UNION
SELECT * FROM alumnos WHERE creditos >= 55
```

La intersección de dos relaciones puede realizarse con el operador INTERSECT y la diferencia con EXCEPT, pero estas no están implementadas en MySQL:

```
SELECT * FROM alumnos WHERE creditos <= 60
   INTERSECT
SELECT * FROM alumnos WHERE creditos >= 40       # error en MySQL

SELECT * FROM alumnos WHERE creditos <= 60
   EXCEPT
SELECT * FROM alumnos WHERE creditos  = 40       # error en MySQL
```

¿Cómo podemos implementarlas sin jugar hábilmente con las condiciones? Podemos usar el operador IN: este operador toma una tupla del lado izquierdo (puede ser una 1-tupla, es decir, un valor) y una relación del lado derecho y devuelve verdadero si el operando izquierdo aparece en la relación de la derecha (usualmente calculada en una petición anidada):

```
SELECT * FROM alumnos WHERE creditos <= 60 AND matricula IN
   (SELECT matricula FROM alumnos  WHERE creditos >= 40)
# intersección entre alumnos con  creditos ≤ 60, creditos ≥ 60

SELECT * FROM alumnos WHERE creditos <= 60 AND matricula NOT IN
   (SELECT matricula FROM alumnos WHERE creditos = 40)
# diferencia entre alumnos con creditos ≤ 60 menos creditos = 40
```

Peticiones anidadas

Existen varios tipos de peticiones anidadas:

- Peticiones de un único escalar: son aquellas que devuelven un único valor (una fila con una columna). Por ejemplo, suponga que se tiene la siguiente tabla:

alumnos					
matricula	nombre	apellido	carrera	estado	creditos
201501	pablo	perez	1	1	50
201502	juan	gomez	2	2	0

  Ya que una matrícula debiera ser llave primaria, la siguiente petición debería entregar a lo mucho una única fila:

```
SELECT creditos FROM alumnos WHERE matricula = 201501;
```

  Entonces, podemos usar la siguiente petición para buscar todos los alumnos con la misma cantidad de créditos que el alumno 201501:

```
SELECT * FROM alumnos WHERE creditos =
(SELECT creditos FROM alumnos WHERE matricula = 201501)
```

  En este caso, un error si la petición anidada devuelve más de una fila. Se pueden usar los operadores relacionales <, <=, etc. disponibles en MySQL.

- Peticiones de múltiples escalares: son aquellas que devuelven varios valores (varias filas con una columna). Se permite usar el operador `IN` (incluído en) y hacer comparaciones relacionales junto con los operadores `ANY` (cualquiera) y `ALL` (todos). Por ejemplo:

```
SELECT * FROM alumnos WHERE creditos > ALL
(SELECT creditos FROM alumnos WHERE carrera = 2)

# calcula la lista de alumnos que tengan más créditos
inscritos que todos los alumnos de la carrera 2

SELECT * FROM alumnos WHERE creditos > ANY
(SELECT creditos FROM alumnos WHERE carrera = 2)

# calcula la lista de alumnos que tengan más créditos
inscritos que alguno de los los alumnos de la carrera 2
```

- Peticiones de filas: son aquellas que devuelven una única fila (pero puede tener varias columnas). Funcionan de una manera muy similar a las peticiones de un escalar. Las comparaciones como < son lexicográficas (y dependen del orden en el que se listen las columnas).

- Peticiones de relaciones y peticiones correlacionadas: son aquellas que devuelven más de una fila (y tener varias columnas). Se puede preguntar si la petición de relación anidada es o no es vacía con el operador `EXISTS`:

```
SELECT * FROM carreras WHERE EXISTS
(SELECT * FROM alumnos WHERE alumnos.carrera = carreras.id)
# selecciona las carreras que tienen al menos un alumno
```

En este ejemplo, debemos notar que la petición anidada no se refiere explícitamente a la tabla carreras, sin embargo en la condición puede nombrar carreras.id debido a que la petición externa hace referencia a dicha tabla. Esta petición a grandes rasgos se ejecuta de la siguiente manera:

```
PARA CADA FILA f DE carreras
   SI EXISTE UNA FILA EN alumnos DONDE alumnos.carrera = f.id
      INCLUIR f en el resultado
```

Esto puede tener un comportamiento |carreras| * |resultado|, lo cual puede ser terriblemente lento. Los índices pueden ayudar en este caso. Para verificar cómo es que MySQL ejecutará una petición, se puede usar la sentencia `EXPLAIN`:

```
EXPLAIN SELECT * FROM carreras WHERE EXISTS
(SELECT * FROM alumnos WHERE alumnos.carrera = carreras.id)
```

También se pueden usar peticiones anidadas en la cláusula FROM. Por ejemplo, la siguiente petición:

```
SELECT alumnos.nombre FROM alumnos;
```

puede escribirse de la siguiente manera:

```
SELECT temp.nombre FROM
   (SELECT alumnos.nombre, alumnos.apellido FROM alumnos) AS temp;
```

Funciones de agregación

Una función de agregación es una función que se calcula utilizando un conjunto de filas (en lugar de una única fila). Por ejemplo, podemos calcular el promedio de los créditos inscritos por todos los alumnos:

```
SELECT AVG(creditos) FROM alumnos;
```

La petición anterior devuelve una única fila (y en este caso, una única columna). Para expresiones, usualmente MySQL asigna un nombre de columna que corresponde con la expresión usada para calcularla. En estos casos, es más fácil utilizar la función `fetch_row` de PHP la cual, a diferencia de `fetch_assoc` que devuelve una tupla con elementos denotados por los nombres de los atributos, devuelve una fila con los valores de la tupla indizados por posición:

```
$res = $conexion->query('SELECT AVG(creditos) FROM alumnos');
$tupla = $res->fetch_row( );
echo $tupla[0];     // el promedio de los créditos
```

También podemos usar renombramiento en caso de querer seguir usando `fetch_assoc`. La función de agregación `COUNT(*)` devuelve el número de filas del resultado:

```
SELECT COUNT(*) FROM alumnos WHERE creditos > 10;
```

La función `COUNT` puede tomar una expresión o nombre de atributo; devolverá la cuenta de aquellas filas que no resultan en un valor nulo.

Dada una base de datos con 1000 alumnos y 100 carreras (donde cada alumno tiene una carrera), uno pensaría inicialmente que la siguiente petición devolvería a lo mucho 100, pero no es así:

```
SELECT COUNT(carrera) FROM alumnos;
```

Lo anterior devolverá 1000: hay 1000 filas donde la carrera es no nula. Para no contar duplicados, podemos hacer lo siguiente:

```
SELECT COUNT(DISTINCT carrera) FROM alumnos;
```

Volviendo al promedio de créditos, ¿qué pasa si queremos calcular el promedio de créditos para todos los alumnos de cada carrera por separado, en lugar de todos los alumnos de todas las carreras? Esto lo podemos hacer con `GROUP BY`:

```
SELECT SUM(creditos) FROM alumnos GROUP BY carrera;
```

La petición anterior devolverá 100 filas si hay alumnos en todas las carreras; cada una de las 100 filas tendrá el promedio de la carrera respectiva. Recomendamos proyectar una columna adicional para poder ver mejor el resultado (así como está la petición, no sabremos a qué carrera corresponde cada promedio):

```
SELECT carrera, AVG(creditos) AS promedio FROM alumnos GROUP BY
carrera;
```

¿Qué pasa si sólo queremos reportar carreras cuyo promedio sea mayor que 10? Ese promedio no lo conocemos de antemano sino hasta que se hayan sumado todas las filas, por lo que no podemos usar un `WHERE`. Para estos casos podemos usar `HAVING`:

```
SELECT carrera, AVG(creditos) AS promedio FROM alumnos GROUP BY
carrera HAVING promedio > 10;
```

Por otro lado, es posible hacer el agrupamiento aún más fino si se especifica una columna adicional:

```
SELECT carrera, AVG(creditos) AS suma_creditos FROM alumnos GROUP
BY carrera, apellido;
```

La petición anterior primero partirá las filas por carrera y luego por apellido; al final tendremos el promedio de créditos para cada apellido diferente de alumnos de cada carrera. Existen muchas funciones de agregación que pueden ser consultadas en la documentación de MySQL.

Lectura y escritura de filas en archivos y copia de filas de una tabla a otra

Se puede volcar el conjunto de tuplas que resultaron de una consulta SELECT en un archivo. Por ejemplo:

```
SELECT * FROM telefonos INTO OUTFILE "archivo.txt";
```

La petición anterior escribirá un archivo (por omisión, cada fila en su propia línea y con valores separados por tabuladores) en la carpeta de la base de datos actual. En el caso de EasyPHP, esta carpeta está en `EasyPHP/binaries/mysql/data/`*`nombre_db`*.

Se pueden insertar las filas guardadas en un archivo de la siguiente manera:

```
LOAD DATA INFILE "archivo.txt" INTO TABLE telefonos;
```

El número de atributos y el orden en el que están en el archivo deben coincidir con el formato que hubiera producido `INTO OUTFILE` para esa misma tabla. En general, apagar la revisión de llaves foráneas y luego ejecutar un `LOAD DATA INFILE` es la manera más rápida de insertar filas en una tabla en MySQL.

De manera muy similar, se puede usar una petición anidada para insertar filas de una tabla a otra:

```
# suponer que telefonos1 y telefonos2 tienen las mismas columnas
INSERT INTO telefonos1 (matricula, telefono)
   SELECT matricula, telefono FROM telefonos2;
```

Se insertará cada fila que se sea parte del resultado en la petición anidada. El nombre de las columnas no tiene por qué coincidir, pero deberían coincidir en cuanto a número, tipos y orden en el que se lista.

Número de filas en un resultado y número de filas afectadas

El número de filas del resultado de una petición `SELECT` puede obtenerse con `num_rows` de MySQL. En PHP este valor es propiedad del objeto `mysqli_result`:

```
$res = $conexion->query('SELECT * FROM alumnos
   WHERE creditos > 10');
echo $res->num_rows;
```

El valor de `num_rows` es independiente de la transferencia de las filas obtenidas desde MySQL a PHP (y por lo tanto, puede ser usado con menor sobrecarga). No tocaremos el tema de devolución de filas sin almacenamiento intermedio, pero mencionaremos que `num_rows` puede comportarse de una manera diferente bajo este modo.

Ya que `num_rows` es el número de filas del resultado, `SQL_CALC_FOUND_ROWS` no afectará este valor.

Para obtener el número de filas afectadas por una sentencia UPDATE o DELETE, podemos hacer lo siguiente:

```
$conexion->query('DELETE FROM alumnos WHERE creditos < 10');
echo $conexion->affected_rows;
```

Ya que no hay un objeto `mysqli_result` para estas peticiones, `affected_rows` es una propiedad del objeto `mysqli`.

Por defecto, una fila se considera no afectada si se le aplica un `UPDATE` que no cambia ningún valor de la fila. Ocasionalmente es útil saber si al menos MySQL encontró la fila que queríamos actualizar. Por ejemplo:

```
$conexion->query('UPDATE alumnos SET creditos = 0
   WHERE matricula = 201501');
echo $conexion->affected_rows;
     // imprime 0 si el alumno no existe
     // imprime 0 si el alumno sí existe pero ya tenía 0 créditos
```

Podemos pedirle a MySQL que affected_rows devuelva el número de filas que al menos fueron encontradas durante un `UPDATE`. Esto lo podemos hacer con `real_connect` y `MYSQLI_CLIENT_FOUND_ROWS` sin usar directamente el constructor de `mysqli`:

```
$conexion = mysqli_init( );
$conexion->real_connect('localhost', 'root', '', 'ejemplo',
     null,
     null,
     MYSQLI_CLIENT_FOUND_ROWS);

$conexion->query('UPDATE alumnos SET creditos = 0
   WHERE matricula = 201501');
echo $conexion->affected_rows;
     // imprime 0 si el alumno no existe
     // imprime 1 si el alumno sí existe
```

Sentencias preparadas

Es frecuente que queramos ejecutar una petición casi idéntica repetidamente. Por ejemplo:

```
$conexion = new mysqli('localhost', 'root', '', 'ejemplo');
$borrar = [ 201501, 201505, 201510, 201525, 201531, 201550 ];

foreach ($borrar as $m) {
   $conexion->query("DELETE FROM alumnos WHERE matricula = $m");
}
```

Para un humano es fácil razonar que:

- Todas las peticiones a ejecutar están bien sintácticamente: la única diferencia entre ellas es el valor de la matrícula.
- El análisis de qué tabla se modifica en cada petición y qué índice usar para ubicar la fila a borrar debiera ser el mismo.

Sin embargo, para MySQL las peticiones anteriores no están relacionadas: el servidor las recibe como meras cadenas y por lo mismo, pueden ser potencialmente diferentes. MySQL analizará cada petición por separado. Usando sentencias preparadas podemos indicarle a MySQL que queremos ejecutar una petición múltiples veces, en donde cada ejecución sólo variará en el valor de algunos escalares:

```
$conexion = new mysqli('localhost', 'root', '', 'ejemplo');
```

```
$stmt = $conexion->prepare(
     'DELETE FROM alumnos WHERE matricula = ?'
);
$stmt->bind_param('i', $m);

$borrar = [ 201501, 201505, 201510, 201525, 201531, 201550 ];

foreach ($borrar as $m) {
   $stmt->execute( );
}
```

La función miembro `prepare` crea una sentencia preparada en donde sólo están pendientes los valores escalares indicados por el caracter `?` en la petición; el objeto devuelvo es de tipo mysqli_stmt. La función miembro `bind_param` sirve para enlazar variables de PHP con los escalares faltantes y se debe indicar el tipo del escalar (`i` para enteros, `d` para flotantes, `s` para cadenas), por ejemplo:

```
$stmt = $conexion->prepare(
     'INSERT INTO telefonos (matricula, telefono) VALUES (?, ?)'
);   // la sentencia preparada tiene dos escalares no especificados
$stmt->bind_param('is', $m, $t);
          // el primer escalar es un entero enlazado a $m
          // el segundo escalar es una cadena enlazado a $t
$i = 201501;
$t = "01800-123":
$stmt->execute( );       // insertar (201501, ' 01800-123')
```

Dado que se especifica el tipo de los escalares, es innecesario usar `escape_string` para protegerse de inyecciones SQL: la sintaxis general de la petición no se afectará por el valor de éstos.

También es posible usar un mecanismo similar para leer datos devueltos por una sentencia preparada:

```
$stmt = $conexion->prepare(
     'SELECT nombre, apellido FROM alumnos WHERE matricula = ?'
);   // la sentencia preparada tiene un escalar no especificado
$stmt->bind_param('i', $m);   // es un entero enlazado a $m

$m = 201501;
$stmt->execute( );

$stmt->bind_result($n, $a);   // la sentencia preparada devuelve
     // tuplas con dos valores (nombre, apellido), enlazar a $n, $a

while ($stmt->fetch( )) {     // mientras haya filas por leer
     echo $n, ' ', $a, "\n";  // imprimr los valores recibidos
}
```

Sólo sugerimos el uso de sentencias preparadas cuando se ejecutará varias veces la misma petición (salvo valores escalares). Si sólo se ejecutará la petición una vez, una sentencia preparada puede suponer mayor sobrecarga (dos comunicaciones con el servidor: preparar la sentencia y el envío de escalares). La protección automática con respecto a inyección SQL en escalares debe sopesarse.

Otros tipos de datos

Existen varios tipos de datos disponibles en MySQL. La información completa sobre estos puede encontrarse en la documentación de MySQL. Nosotros resumiremos dicha información en lo siguiente:

Tipos enteros

Un atributo entero puede declararse como entero sin signo (por ejemplo INT UNSIGNED). Además de INT, existen otros tipos enteros:

Tipo	**B = Bytes usados (rango de -2^{b-1} a $2^{b-1} - 1$)**
`TINYINT`	1
`SMALLINT`	2
`MEDIUMINT`	3
`INT`	4
`BIGINT`	8

Además, el tipo `BOOL` es sinónimo de `TINYINT(1)`, donde el número entre paréntesis es la cantidad de dígitos que MySQL despliega. El tipo `BIT`, por otro lado, está pensando para ocupar un único bit en disco y `BIT(M)` es un arreglo de bits con $1 \leq M \leq 64$.

Tipos de cadena

El tipo `CHAR(M)` con $0 \leq M \leq 255$ es un tipo de cadena de ancho fijo `M`. El tipo `VARCHAR(M)` con $0 \leq M \leq 65535$ es un tipo de cadena de longitud variable. El consumo en disco de ambos tipos de cadena es:

Tipo	**L = longitud de la cadena real guardada**
`CHAR(M)`	`M`
`VARCHAR(M)` con $0 \leq M \leq 255$	1 + `L`
`VARCHAR(M)` con $256 \leq M \leq 65535$	2 + `L`

En pocas palabras, `VARCHAR` sólo requiere especio proporcional a la longitud de la cadena que realmente está siendo almacenada, más algunos bytes requeridos para almacenar el tamaño real de dicha cadena. El tipo `TINYTEXT` es sinónimo de `VARCHAR(255)`, el tipo `TEXT` de `VARCHAR(65535)`, `MEDIUMTEXT` sería el equivalente a `VARCHAR(16777216)` y `LONGTEXT` de `VARCHAR(4294967296)`.

Los tipos `BLOB` son parecidos a los tipos `TEXT` excepto que están pensandos para almacenar cadenas binarias y MySQL no intenta realizar transformaciones ni equivalencias entre ellas (como lo hace con las cadenas normales, como en comparaciones insensibles a mayúsculas). La codificación de cadenas no `BLOB` puede especificarse en la sentencia `CREATE TABLE`:

```
CREATE TABLE ejemplo (
   id INT NOT NULL AUTO_INCREMENT,
```

```
   nombre TEXT NOT NULL,

   PRIMARY KEY (id)
) ENGINE=InnoDB CHARACTER SET=utf8mb4;
```

La codificación usada en el envío de datos desde PHP y MySQL puede especificarse (o consultarse) de la siguiente manera:

```
$conexion->set_charset('utf8mb4');
echo $conexion->charset_set_name( );
```

Tipos flotantes y decimales

Los tipos `FLOAT` y `DOUBLE` utilizan 4 y 8 bytes respectivamente y se utilizan para almacenar números racionales (con una precisión aproximada). Por otro lado, existe el tipo `DECIMAL(M, D)` que permite guardar el valor de un racional representable exactamente en notación decimal, con `M` dígitos en total y `D` después del punto decimal. Por ejemplo, el número 123.45 puede almacenarse en un campo de tipo `DECIMAL(5, 2)`. Recomendamos ampliamente el uso de este tipo en aplicaciones que requieran guardar cantidades monetarias.

Tipos fecha y tiempo

Recomendamos al lector consultar la documentación de MySQL. Sólo mencionaremos que existen tipos diseñados para almacenar fechas (`DATE`, `YEAR`), horas (`TIME`), fecha y hora (`DATETIME`, `TIMESTAMP`) y funciones para leer la fecha y hora actual como `NOW( )` y para manipular fechas como `DATE_SUB`.

Procedimientos almacenados

VISTO EN CLASE, PRÓXIMAMENTE EN LAS NOTAS

Funciones almacenadas

VISTO EN CLASE, PRÓXIMAMENTE EN LAS NOTAS

Desencadenadores

Un desencadenador permite ejecutar un fragmento de código antes (`BEFORE`) o después (`AFTER`) de un `INSERT`, `UPDATE` o `DELETE`. Por ejemplo:

```
DROP TRIGGER IF EXISTS imprime;
CREATE TRIGGER imprime BEFORE DELETE ON alumnos FOR EACH ROW
     /*código*/ ;
```

El fragmento de código se ejecuta para cada una de las filas que disparan el desencadenador. En MySQL, un desencadenador puede hacer referencia a los valores previos al evento (`OLD.atributo`) o posteriores al evento (`NEW.atributo`); para abortar la modificación de una fila puede elevarse un error con SIGNAL SQLSTATE '45000'; por ejemplo:

```
DELIMITER $
CREATE TRIGGER actualiza BEFORE UPDATE ON alumnos FOR EACH ROW
     BEGIN
        IF NEW.creditos < OLD.creditos THEN
           SIGNAL SQLSTATE '45000';
        END IF;
     END $
DELIMITER ;
```

Sólo puede existir un desencadenador por tabla por evento.

Transacciones

Una transacción es una unidad de trabajo que se realiza sobre una base de datos de manera consistente e independiente de otras transacciones. En bases de datos relaciones, se pide que una implementación de transacciones cumpla el siguiente conjunto de propiedades:

- Atomicidad (*atomicity*): Una transacción debe poder ejecutarse completa, o bien, no ejecutarse en absoluto. Si antes de terminar la ejecución de una transacción multi-sentencia se desea abortarla u ocurre una falla (por ejemplo, falta de energía eléctrica), las sentencias ejecutadas de la transacción deben quedar sin efecto.
- Consistencia (*consistency*): Toda transacción debe cumplir con la semántica impuesta en la base de datos (cumplimiento de restricciones, ejecución de desencadenadores).
- Aislamiento (*isolation*): Las transacciones deben ejecutarse correctamente aún en concurrencia.
- Durabilidad (*durability*): Una vez que una transacción es ejecutada, los efectos de ésta deben poder reflejarse sin errores en el almacenamiento permanente.

En MySQL, el motor de almacenaminto más usado con soporte para transacciones es InnoDB. El motor MyISAM no tiene soporte para transacciones.

Una transacción inicia con la sentencia `START TRANSACTION` y se indica el fin de su ejecución con la sentencia `COMMIT`. La sentencia `ROLLBACK` cancela una transacción (y la deja sin efecto). La siguiente es una transacción de ejemplo:

```
DELIMITER $
CREATE PROCEDURE mueve_credito(matricula1 INT, matricula2 INT)
BEGIN
   DECLARE fuente INT;

   START TRANSACTION;
   SELECT creditos INTO fuente FROM alumnos
     WHERE matricula = matricula1;
   UPDATE alumnos SET creditos = creditos - 1
     WHERE matricula = matricula1;
   UPDATE alumnos SET creditos = creditos + 1
     WHERE matricula = matricula2;

   IF fuente > 0 THEN
      COMMIT;
```

```
   ELSE
      ROLLBACK;
   END IF;
END $
DELIMITER ;
```

No es necesario que una transacción esté en un procedimiento almacenado (aunque es usual que esto ocurra). Se puede iniciar una transacción desde PHP usando la función miembro `begin_transaction` de `mysqli`; existen también las funciones miembro `commit` y `rollback`. Una manera equivalente de obtener una transacción es apagar el `autocommit`:

```
SET autocommit = 0;                # para MySQL
$conexion->autocommit(false);      // para PHP
```

Ninguna sentencia ejecutada a partir de la desactivación del modo `autocommit` (durante la conexión actual) se reflejará en el almacenamiento persistente hasta ejecutar un `commit`.

¿Cómo se implementan (a grandes rasgos) las transacciones? En una bitácora, el motor InnoDB anuncia que está por comenzar una transacción y trae a memoria todas las filas que necesitará modificar; las sentencias de la transacción se ejecutarán en memoria. Una vez que se ejecuta el COMMIT de una transacción, se notifica en la bitácora que la transacción tuvo éxito.

La actualización en el almacenamiento persistente puede tardar un poco más. En caso de algún fallo (en donde se pierdan los datos actualmente cargados en memoria), MySQL podrá utilizar la bitátoca para recrear los cambios que aún no había actualizado en disco.

NIVELES DE AISLAMIENTO VISTO EN CLASE, PRÓXIMAMENTE EN LAS NOTAS

Vistas

VISTO EN CLASE, PRÓXIMAMENTE EN LAS NOTAS

Manejo de usuarios y permisos

VISTO EN CLASE, PRÓXIMAMENTE EN LAS NOTAS