**SQL & MySQL**

**Diferencias entre SQL & MySQL**

* ***SQL*** es un lenguaje de programación orientado a consultas de bases de datos (Structured Query Language).
* ***MySQL*** es un sistema de administración de bases de datos (Database Management System, DBMS) o también llamado motor de bases de datos.

Existen muchos tipos de bases de datos, desde un simple archivo hasta sistemas relacionales orientados a objetos. *MySQL*, como base de datos relacional, utiliza múltiples tablas para almacenar y organizar la información. Fue escrito en C y C++ y se integra perfectamente con los lenguajes de programación más usados en todo el mundo.

**Una forma de conectarnos a nuestro servidor MySQL a través de nuestra terminal**

**../../../../../../Captura%20de%20pantalla%202022-02-11%20a%20las%207.25.32%20p.m..png**

Con ***mysql*** intentamos llamar directamente al manejador de la base de datos.

Con el comando ***–u*** preguntamos cuál es el usuario, el que navegará en la termina. Usuario: ***root.***

Con el comando ***–h*** preguntamos cuál es el host, en qué servidor *(IP o dominio)* está tu base de datos.

Servidor: ***localhost.***

Con el comando ***–p*** preguntamos cuál es la contraseña con la cual nos conectaremos a MySQL.

**Comandos básicos iniciales para navegar en MySQL a través de nuestra terminal**

***show databases;***-> lista las bases de datos que tiene el servidor.

***use “name\_database”;*** -> selecciona o se conecta a la base de datos de interés.

***show tables;***-> muestra las tablas que contiene la base de datos seleccionada.

***select database();*** -> muestra cuál es la base de datos seleccionada o en la que se está trabajando parcialmente.

***Nota****: Todos los comandos deben de terminar con****“;”.***

**Qué es una base de datos?**

Lugar de almacenaje de datos, de cualquier tipo, que se operan para ser convertidos en información útil y práctica. Generalmente esta información es usada para crear estrategias comerciales y/o poner en marcha operaciones de negocio; todo esto con la finalidad de simplemente lograr crecimiento, ser más productivos (o rentables) sobre un proyecto. A lo sumo, toda esta información que se crea sirve para la mejora de toma de decisiones.

De hecho, al comparar o *relacionar* varios tipos de datos relacionables entre sí impulsamos a la generación de más conocimiento y, en efecto, en eso nos ayuda una base de datos relacional. *A partir del relacionamiento de datos es que creamos información y conocimiento.*

**Documentación oficial sobre la totalidad de tipos de datos de MySQL:** <https://dev.mysql.com/doc/refman/8.0/en/data-types.html>

**Entidades & Atributos.**

**Sobre los principales objetos de manipulación dentro de una base de datos relacional.**







**Pasos para crear una base de datos relacional en nuestro proyecto (esta vez el proyecto será un sistema de "Platziblog").**

**En un proyecto real, para crear una base de datos relacional, lo primero que hay que hacer es:**

**1. Identificar las entidades (para nuestro caso, serían las entidades de un “Blog” de Platzi).**

*1ra entidad:* ***"Posts":*** *El cuerpo propiamente del Blog.*

*2da entidad:* ***"Usuarios":*** *Los que van a crear el contenido del Blog.*

*3ra entidad:* ***"Comentarios":*** *Caja de comentarios al contenido del Blog.*

*4ta entidad:* ***"Categorías":*** *Tienen que ver con las sesiones del "blog"; es decir, las pestañas que tiene el blog por separado por cada tema de interés. Las categorías pueden ser: "Política", "Deportes", "Economía", "Moda", "Tecnología", etc.*

**2. Identificar los atributos (para nuestro caso, serían los atributos de un “Blog” de Platzi).**

***Para la entidad "Posts":***

*1er atributo:* ***Un título.***

*2do atributo:* ***La fecha\_de\_publicación.***

*3er atributo:* ***el Contenido.***

*4to atributo:* ***El estatus*** *(si fue publicado o no, si está activo o no).*

*5to atributo:* ***Las etiquetas****, las cuales son Atributos Multivaluados; es decir, hay varias etiquetas realmente y se encierran en un ovalo con doble subrayado. La etiqueta es la que nos dicta que tipo de post estás leyendo,* ***son las etiquetas de las categorías****; es decir, las etiquetas que se ponen al final del post si hace parte de la sesión de "ciencias" o de la sesión de "tecnología" o de "deportes", etc. Pues bien, en un mismo post, pueden haber varias etiquetas; en ese caso, como son varias (una por post), es por lo anterior que en el atributo "etiqueta" hablamos de que sea una etiqueta "multivalor".*

*6to atributo:* ***Un id, esta sería el atributo "llave" o clave de nuestra entidad "Posts"****; es lo que nos dirá que todos los* ***posts*** *serán diferentes entre sí, al menos, mínimamente, por el "id".*

***Para la entidad "Usuarios":***

*1er atributo:* ***Un “login”****, para poder hacer inicio de sesión con una cuenta.*

*2do atributo:* ***Un “password”****, para tener acceso a nuestra cuenta con una contraseña.*

*3er atributo:* ***Un apodo o “alias”.***

*4to atributo:* ***Un email.***

*5to atributo:* ***Un id, esta sería la etiqueta "llave" o clave de nuestra entidad "Usuarios"****; es lo que nos dirá que todos los* ***usuarios*** *serán diferentes entre sí, al menos, mínimamente, por el "id".*

***Pregunta***: Puede una imagen almacenarse en nuestra base de datos (o tabla) un atributo para *imágenes*? Sí, se puede, pero como buena práctica se recomienda que NO se almacene directamente la imagen dentro de la base de datos en sí, no, sino que se guarde en un *VARCHAR()* la URL hacia donde esa imagen esté almacenada (esto se hace con fines de optimización). Por ej.: 

***Sobre las relaciones***

**Las relaciones es la manera en la que empezamos a ligar nuestras diferentes entidades u objetos (De aquí surge la importancia de los Diagramas entidad-relación)**

Las relaciones se representan con un **Rombo.**

**Cómo funcionan las relaciones?**

Supongamos que tenemos la entidad *Automóvil* y la entidad *Dueño*, siendo en total dos (2) entidades:

Dueño

Automóvil

Ahora, lo que nos permite vincular o relacionar ambas entidades son las ***relaciones,*** *éstas por convención se definen por medio de* ***verbos.*** En este caso, un verbo más apropiado de la relación que hay entre las entidades “**automóvil” y “dueño” *es “*tener”.** Quedando visualmente este vinculo de entidades y relaciones así:

Automóviles

tienen

Dueños

Es necesario el uso de *líneas conectoras* para poder ligar o hacer la unión entre las entidades, a esta representación visual se le llama: **“Diagramas entidad-relación”**.

***Otro ejemplo de relación entre dos entidades podría ser:***

Jugadores

pertenecen

Equipos

Lo que distingue una entidad de un atributo es que el atributo es algo propio a una entidad; es decir, es una característica de la entidad u objeto; mas, cuando se habla de dos entidades, se habla de dos objetos por separado y completamente independientes entre sí. **Sin embargo, hay un tipo de atributo en especifico que recibe un tratamiento igual al de una entidad en términos de *relaciones, y son: Los atributos Multivaluados.*** Veamos esto en un caso que ya hemos trabajado.

Laptops

tienen

discos\_duros

Si se da cuenta, realmente los *discos duros* no son independientes a una Laptop; antes, son un elemento o característica propia de ellas (de las laptops); es decir, los *discos duros* no son una entidad como tal; sin embargo, como es un *atributo multivaluado*, recibe el mismo tratamiento en términos de relaciones que cualquier otra entidad (convirtiéndose, en ese sentido, en una entidad más). Lo anterior se debe a que los atributos multivaluados son más complejos y pueden relacionarse de varias formas con su entidad.

Ahora, para definir o saber puntualmente la cantidad de, por ejemplo, los *discos duros* que tiene una *laptop* se hace por medio de una nueva propiedad llamada: ***Cardinalidad;*** es decir, la ***cardinalidad*** vendría siendo lo que nos permite saber la cantidad de entidades o de atributos multivaluados que hay de lado y lado en cada relación resuelta. O también se puede definir la ***Cardinalidad*** como: **La facultad de saber la cantidad numérica de valores que tiene un atributo multivaluado (o una entidad) para otra entidad en cuestión (al ser relacionadas ambas)”. Tenga algo en cuenta, la totalidad de los *ID* de cada columna (de todos los registros de una tabla), sus *primary key* para ser más exactos, no siempre corresponde con el número de la cardinalidad para una entidad con relación a otra (A veces, por algún motivo, se suprime el *id* de algún registro de alguna tabla… no correspondiéndose así el conteo de los *id* con la cardinalidad de la tabla misma debido a que el manejador de la base de datos sigue su conteo sin tomar en cuenta el *id* suprimido, se lo “vuela” en el conteo).**

***Cardinalidad: casos o diferentes tipos.***

***Cardinalidad: 1 a 1***



Cuando se habla de ***cardinalidad: 1 a 1,*** nos referimos a que sólo hay un valor, para lado y lado, en la relación etiqueta-etiqueta o etiqueta-atributo multivaluado; en nuestro ejemplo, para la entidad “persona”, nos estaríamos refiriendo a una sola persona como tal por separado; por otra parte, para el atributo multivaluado “datos-contacto”, nos referimos a los datos de contacto correspondientes a una única persona precisamente. ***Esta representación gráfica reflejada es un diagrama entidad-relación.*** Sin embargo, existe otro tipo de diagrama para representar visualmente la misma información dada por un caso de cardinalidad: 1 a 1,... y son los ***Conectores*** *de un diagrama físico de una base de datos****.***



El primer conector básicamente nos dice lo mismo: *“de un lado tengo uno y del otro lado tengo otro”;* mas, el segundo conector también significa lo mismo, sólo que refuerza la declaración de que en un lado efectivamente tengo uno *(y sólo uno)* y del otro lado tengo uno también *(y sólo uno).* Esa expresión de *“sólo uno”* se simboliza gráficamente con la doble raya a los extremos del conector.

***Cardinalidad: 0 a 1 (o Cardinalidad: 1 a 1 opcional)***

Acá puede darse la opción de que no exista ninguna unidad en uno de los dos lados. Veamos el siguiente caso.



Si analizamos la “sesión\_actual”, la sesión de un usuario, con el “usuario” mismo; podemos decir que toda “sesión\_actual” tiene que tener un usuario, pero un usuario puede *no estar en sesión* en determinado momento; por lo tanto, la *cardinalidad se define como de: 0 a 1,* pues, puede que no haya una *sesión actual.*

***Cardinalidad: 1 a N (o 1 a muchos)***

***La Cardinalidad: 1 a muchos*** quiere decir, como su nombre indica, que de un lado tenemos uno pero del otro lado tenemos muchos. Veamos el siguiente ejemplo.



Como logra observar, se plantea que una persona puede tener muchos automóviles.

***Cardinalidad: 0 a N (o 0 a muchos)***

Acá puede haber la opción de que no exista ninguno de uno de los dos lados, pero muchos en el otro lado restante. Veamos el siguiente caso.



Si analizamos al “paciente”, el que ocuparía la habitación de hospital, con la “hab\_hospital” misma; podemos decir que todo “paciente” tiene que tener asignado una “hab\_hospital”; pero, en algún caso, muchas de esas *habitaciones de hospital* quedan completamente desocupadas de *pacientes;* por lo tanto, la *cardinalidad se define como de: 0 a N,* pues, puede que no haya ningún *paciente* para una cantidad considerable de *habitaciones de hospital* disponibles.

***Cardinalidad: N a N (muchos a muchos)***

Esta es, quizás, el tipo de *cardinalidad* más especial; se separa un poco del resto de *cardinalidades* porque no contempla a una “*entidad principal”* con un único valor posible. *Generalmente son muchos de lado y lado*. Veamos el siguiente ejemplo.



Un alumno puede estar inscrito o tomar varias clases; puede, por ejemplo, tomar Español, Matemáticas o Fundamentos de bases de datos (1-N); pero adicionalmente una clase, cualquiera de esas tres mencionadas, puede estar *relacionada* o contiene a varios alumnos (N-1). El último conector de esta cardinalidad es más estricto que el primero; pues, nos cuenta que, en todos los casos, siempre habrá *muchos* de un lado y *muchos* de otro: no puede haber *0s,* no pueden haber *1s*, siempre habrán *muchos.* Ahora, lo más común es que se dé el primer conector y no el último entre la *cardinalidad: N a N.*

***Diagramas entidad-relación: a detalle***

**Es preciso decir que la introducción al tema de *Entidades, atributos y, posteriormente, relaciones* es fundamental para profundizar sobre el tema que nos concierne ahora: *Los diagramas entidad-relación.***

***Qué es un diagrama ER?***

Un diagrama lo que nos va a hacer es como una especie de mapa. Pongamos así. El equivalente a, por ejemplo, los planos que diseñan los arquitectos antes de construir un edificio, para nuestro caso, serían los diagramas ER antes de construir nuestras bases de datos relacionales. Ya con el diagrama es que se entenderá bien cuáles son las entidades con las que vamos a trabajar, sus relaciones y cuáles son los atributos de dichas entidades. Dicho eso, entonces, sabríamos qué papel jugarían todos esos elementos dentro de los sistemas o aplicaciones que vamos a desarrollar con nuestras bases de datos. Pongamos todo esto en contexto con nuestro mismo proyecto: *PlatziBlog.*



Si se da cuenta, hemos agregado al conjunto de entidades a las “etiquetas”; pues, si se recuerda, habíamos comentado que muchas veces los atributos multivaluados recibían un tratamiento similar, o se convertirían, en entidades separadas también. Lo anterior tiene sentido si recordamos que los atributos multivaluados tienen su vida propia; para este caso puntual, *están en más de una noticia, en más de un tipo de Blogposts.* Pasa que las *etiquetas* ya no dependen directamente de la entidad *Post*; sino que, pueden estar asociadas a diferentes entidades y no sólo a los *Posts (y pueden estar en varios Posts, y no sólo uno).* Entonces, éstas serían nuestras 5 entidades para nuestra base de datos relacional del proyecto “Platziblog”.

***Ahora, vamos a empezar a observar nuestras relaciones...***

**Relación 1:** Un usuario puede *escribir* muchos Posts (pero un mismo Post sólo puede ser escrito por un único usuario).

**Relación 2:** Un usuario puede *escribir* muchos comentarios (pero un mismo Comentario sólo puede ser escrito por un único usuario).

**Relación 3**: Un post puede *tener* muchos comentarios (pero un mismo Comentario sólo puede estar en un único Post).

**Relación 4:** Una categoría (un tema central) puede *tener* muchos posts; por ejemplo, la sesión de *Economía* puede tener varios posts que escriban de ella, que escriban sobre varios temas relacionados a lo *económico.* Pero un único Post sólo puede pertenecer a una sola Categoría.

**Relación 5:** Una etiqueta (una etiqueta de un tema central) puede tener muchos posts; pero, también, un post puede tener muchas etiquetas: creando así una cardinalidad *muchos a muchos.*



***Diagrama Físico***

**El diagrama físico deriva del anterior, *diagrama entidad-relación,* es un complemento al mismo que viene a ultimar los detalles necesarios para dar paso a la creación, ahora sí, de una base de datos relacional.**

***Tipos de dato: los básicos en cualquier base de datos (1er elemento del diagrama físico)***



***Aclaremos diferencias.***

**Tipos de datos de tipo *texto***

En los tipos de dato de tipo ***texto,***la diferencia que hay entre CHAR(n), que te permite almacenar caracteres y cadenas de texto; y VARCHAR(n), que también sirve para almacenar cadenas,… consiste en la optimización del uso de memoria; es decir, *de cuándo saber hacer lo mismo con menos.*

Si conoces a priori la cantidad de caracteres que vas a necesitar o a usar en una cadena de texto; dicho de otra manera, que serán asignados con la declaración y no solicitados por teclado, lo recomendable es que no ocupes más espacio de memoria que el que necesitas realmente para almacenarlos; y, para ese caso, lo ideal sería usar un tipo de dato CHAR(n); sin embargo, por ejemplo, en los casos en los que se le solicita por teclado a un usuario, independiente a uno, que *meta* los valores (caracteres), para una cadena de texto en cuestión, no es una buena práctica declarar un tipo de dato CHAR(n) porque no sabríamos con anticipación la cantidad exacta de memoria que se va a ocupar para ella; por lo que podríamos limitarla o, en el caso contrario, darle una cantidad de memoria excesiva que bien podría haber sido destinada a otras tareas u otras variables; dicho lo anterior, lo ideal sería usar, para estos casos, el tipo de dato VARCHAR(n) que es más flexible, más dinámico y no establece a priori un límite de espacio de memoria para asignar a una variable. Para ambos tipos de datos, CHAR(n) & VARCHAR(n), el parámetro *(n)* definiría el límite máximo de caracteres que podría almacenar una cadena de texto.

Ahora, el tipo de dato TEXT sirve para almacenar cadenas de texto también pero que resultan ser muy grandes. VARCHAR(n), en general, tiene un límite de 255 caracteres (las últimas actualizaciones de MySQL han incrementado este máximo); cuando se necesite guardar más caracteres que eso, se usa siempre el tipo de dato TEXT; en nuestro caso, por ejemplo, será el tipo de dato que usaremos para guardar el contenido textual de nuestros *Posts.* Hay tipos de datos que son subdivisiones a TEXT: TINYTEX & BIGTEXT. Básicamente la diferencia de los tres reside en la cantidad de espacio de memoria que queremos asumir, y estimamos tomar, con los textos que insertemos para la columna en cuestión. TINYTEX < TEXT < BIGTEXT (se definen con fines de optimización).

**Tipos de datos de tipo *numérico***

Estos tipos de datos nos ayudan a almacenar números y a realizar operaciones matemáticas, hay de varios tipos: INTEGER, que nos permite almacenar números enteros. Los siguientes tipos de datos son subtipos de INTEGER: BIGINT, que almacena enteros muy grandes, superiores a 99; y SMALLINT, que almacena enteros más pequeños, que son menores a 99 (2 bits); todo esto es ideal, precisamente, para lograr una mayor eficiencia y optimización en el uso de memoria temporal y, en consecuencia, para desarrollar una base de datos más *rápida.* Ej.: el tipo de dato TINYINT almacena enteros incluso más pequeños que SMALLINT, pues son menores a 9 (1 bit).

En cuanto al tipo de datos DECIMAL(n, s) o NUMERIC(n, s), éstos reciben dos parámetros como datos de entrada.En *n* se añade el número entero y en *s* se añaden los números decimales.

**Hay más tipos de datos como: FLOAT, REAL, DOUBLE**,… FLOAT, por su parte, sólo puede soportar hasta 6 decimales para cada registro de su columna numérica; en cambio, con DOUBLE tenemos más libertad para elegir la cantidad de decimales que el tipo de dato podría manejar, tanto que somos nosotros quienes asignamos a priori el número de decimales que podría soportar DOUBLE. Sin embargo, antes de definir como parámetro el número de decimales que queramos para DOUBLE, se le pasa el número total, entre enteros y decimales, para el mismo (como 1er parámetro del tipo de dato); es decir, DOUBLE realmente recibiría 2 parámetros. El 1ro es la totalidad de los dígitos del tipo de dato (incluyendo decimales) y el 2do es la cantidad de decimales que deseo reservar (*tomar*) a partir de la totalidad numérica disponible para DOUBLE que fue definida en el 1er parámetro. Este tipo de dato numérico queda algo así: DOUBLE (n, d), donde ***n***equivale al total de dígitos disponibles para insertar en el DOUBLE (incluyendo potenciales decimales) y ***d***es el total de decimales que se tomarán de *n****.* Por ej.: *../../../../../../Captura%20de%20pantalla%202022-02-13%20a%20las%2012.52.19%20a.m..png,*** quiere decir que por cada registro, del campo *Price,* se guardará un número máximo equivalente a 6 dígitos, del cual se tomarán o se reservarán dos (2) para ser decimales. ***Tipo****: 1995,22.*

Los tipos de datos de tipo numérico cuentan con un atributo conocido como: *UNSIGNED,* con esto básicamente le estaríamos diciendo a la consola que, para dicho campo numérico, sólo podríamos asignar valores numéricos positivos (se hace con fines de optimización). Si necesitamos almacenar edades, por ejemplo, nunca guardaremos valores negativos, entonces sería adecuado definir un campo "edad" de tipo *entero* sin signo. Si necesitamos almacenar el precio de los libros, definimos un campo de tipo *"float unsigned"* porque jamás guardaremos un valor negativo. Tenga en cuenta que este tipo de “atributo” se diferencia del *atributo* como lo conocíamos hasta ahora, no estamos hablando del *atributo (campo, columna)* de una tabla, no; sino, este último atributo se relaciona con los *atributos* que pueden tener tipos de datos; en este caso, tipos de datos de tipo numérico.

**Tipos de datos de tipo *fecha/hora***

Estos tipos de datos son muy útiles, por ejemplo, en nuestro caso actual (sistema Platziblog), es con este tipo de datos que se podrá insertar una fecha de publicación de los *posts* que publiquemos. Adicionalmente, con estos tipos de datos, al interior del sistema, nos sirve para saber cuándo fue creado un registro, cuando alguien lo modificó, cuando alguien lo borró, por ejemplo.

DATE, nos permite contener la fecha a secas (año, mes, día).

TIME, nos permite contener la hora del día de las 24 horas.

Por último, DATETIME y/o TIMESTAMP, lo que hacen es justamente registrar ambos: fecha y horas (incluso con registros a milisegundos). Es usado para datos de *tiempo* más precisos, sin embargo, es preciso dejar diferencias claras entre ambos… DATETIME se usa en un contexto diferente a TIMESTAMP.

Por ejemplo, **TIMESTAMP** está basado en el número ***epoch*** que empieza a registrar fechas desde el **1 enero de 1970** hasta la fecha. Es preciso decir que su punto de partida comienza desde el *1 de Enero de 1970* debido a que sigue el patrón de tiempo con el que se determinó el inicio de las computadoras,… ellas también siguen este mismo modelo de fecha. Adicionalmente este tipo de fecha TIMESTAMP, a diferencia de DATETIME, es un tipo de fecha que se encuentra *serializada;* es decir, realmente su formato son números enteros, dichos números se registran en segundos y con ellos se pueden realizar operaciones matemáticas. El tipo de dato de fecha/hora TIMESTAMP es ideal si lo que se quiere es llevar un seguimiento de fechas en tiempo real (que la fecha se refleje de forma dinámica, tipo *dashboard,* y avance en la medida que avanza el reloj) o, bien, cuando se quieren registrar fechas en tiempo parcial o *a día de hoy*. Por lo anterior, por este tipo de exigencias, TIMESTAMP es más eficiente que DATETIME (es más rápido); entonces, cada vez que pueda, siempre trate de usar TIMESTAMP.

Hay una palabra reservada propia al tipo de dato fecha TIMESTAMP que es **CURRENT\_TIMESTAMP**. CURRENT\_TIMESTAMP lo que hace es definir los registros, del atributo de fecha en cuestión, con el mismo tiempo que marca la hora de la computadora en la que se sitúa el manejador de la base de datos,… siendo esta hora el valor que se asume bajo los registros de esta columna. Por lo general, CURRENT\_TIMESTAMP se define con DEFAULT: es práctico en los casos en los que no se inserta nada dentro del registro de la columna de tipo TIMESTAMP, pues, él por *default* llenará dicho registro con un valor correspondiente al tiempo que marca la hora del computador (lo dicho).

Dentro del atributo de dato TIMESTAMP todo esto se ve más o menos así: 

Generalmente se usa mucho cuando se quiere monitorear, en tiempo real, la introducción de un nuevo registro dentro de una tabla (esto no siempre se quiere, depende de la tabla); de hecho, para hacer este tipo de *monitoreos* se crea un atributo también que, muchas veces por convención, se le nombra como: **created\_at,** quedando en consola así:



Ahora, con cada actualización de un registro ya existente, si queremos registrar el momento exacto en que se actualizó dicho registro para cierta tabla, podemos asistirnos también del siguiente atributo y sus cualidades:



*La anterior sentencia, updated\_at, también deja registros ante tuplas o registros nuevos. Téngalo en cuenta.*

Caso aparte, cuando se quiera registrar fechas pasadas o del *pasado* (que no son de la actualidad)lo ideal es usar el tipo de fecha **DATETIME;** pues, DATETIME se especializa en eso: pueden guardar cualquier valor de tipo fecha sin restricción alguna, como por ejemplo las fechas de nacimiento o fechas *antes de Cristo*, etc.

**Tipo de datos de tipo *lógico***

BOOLEAN, significa que puede tener básicamente dos valores (binario): verdadero (1) o falso (0), nos sirve de manera aleatoria para hacer una validación entre una bandera; por ejemplo, decir si un Blogpost está activo o inactivo; de esta manera, en consecuencia, cuando queramos sacar sólo los Blogposts que se encuentren activos, podemos utilizar un *booleano* para dejar por fuera todos los demás.

**Tipo de dato especial: ENUM()**

ENUM() básicamente consiste en decirle a la consola qué datos (cualquiera que usted desee) se reciben dentro del atributo manipulado; es decir, con ENUM() somos nosotros quienes determinamos (pre-establecemos) los únicos datos disponibles que se le podrían pasar al atributo o columna en cuestión, para la totalidad de sus registros. Por Ej.: Si quiero un atributo que pregunte por el genero (*gender*) de la persona que llena el formulario y sólo quiero que la persona pueda elegir entre la opción “M” (de masculino), “F” (de femenino) & “ND” (de “No determina”) podría definir a priori estas opciones como las únicas disponibles para ser insertadas en los registros de esta columna de tipo ENUM(), tal que así: **gender ENUM(‘M’, ‘F’, ‘ND’),** siendo *gender* mi atributo. Como ya se dará cuenta cada una de las opciones pasadas se determinan, cada una, con un solo carácter, e idealmente, va en Mayúsculas.

***Lo relevante:*** *Los tipos de datos son usados para definir el tipo de dato de los atributos de las entidades.*

***Constraints: restricciones (2do elemento del diagrama físico).***

**Las restricciones son las reglas que le pasamos a las bases de datos, sus limitaciones; más específicamente, *qué tipo de datos podemos introducir para cada atributo (campo o columna) y en qué condiciones.***

***Hay de varios tipos.***



**NOT NULL:** Cuando tú introduces un registro o una serie de datos a una tabla de una base de datos, el valor por defecto siempre es el valor nulo o *null.* Lo que estás diciendo con esta regla es: en esta columna particular, por ejemplo, en la columna del *nombre,...* no se va a permitir que vaya en blanco; por lo tanto, agregaría la restricción NOT NULL a mi tabla, más precisamente, en mi columna *nombre.* Quiere decir que si te tratan de mandar un registro en el que no viene el *nombre* de la persona, la base de datos inmediatamente va regresar un error exigiendo que se complete el campo del *nombre* y, hasta que no se complete, dicho registro no se podrá guardar en la base de datos. Es muy interesante el uso de este recurso porque te ayuda a definir los datos obligatorios que esperas recibir sí o sí dentro de tu sistema. *Null no es lo mismo que “vacío”.*

**UNIQUE:** Básicamente es decir que un valor pasado a un campo, de una columna seleccionada, debe ser *único* en toda nuestra tabla. Un ejemplo muy claro que ya hemos visto anteriormente es el registro de un *email* por parte de los usuarios.En una tabla de *usuarios* queremos que nadie tenga el mismo *email* dos veces; si yo tengo mi usuario, el cual saqué con una cuenta de mi correo electrónico, no es posible que otra persona saque otra cuenta usando el mismo correo o *email* que yo usé; explicado lo anterior, esta limitación es algo que como regla deberíamos establecer,... entonces, para esos casos especiales, nos conviene usar la restricción: UNIQUE. Entonces, en resumen, si establecemos la restricción UNIQUE en la columna *email,* cuando traté alguien más sacar una cuenta con mi correo electrónico o con un correo de un tercero ya en uso, va a regresar un error diciendo: “No podemos meter este registro porque ya existe otro con el mismo correo electrónico”.

**PRIMARY KEY:** En los diagramas hemos visto que tenemos la necesidad de identificar de manera única (y obligatoriamente) los registros visualizados en una tabla, anteriormente en un ejemplo (dentro de una tabla) esto fue identificado con la columna: **id.** Ahora, es a través de un *campo clave* ode un *campo llave,* asignado, con el que se da la posibilidad de que se pueda identificar cada registro de una tabla de manera única y singular; ese campo clave justamente va a tener la restricción PRIMARY KEY (y suele ser el único campo que la tiene, incluso aunque otros campos sean NN & UNIQUE a la vez, como el *email*); esto nos da una serie de ventajas. **En primer lugar nos garantiza que es NOT NULL y que es UNIQUE (lo que necesitamos garantizar),** es así porque si tu estás introduciendo un campo, que es *llave*, y que pretende ayudarnos a identificar de manera única cada registro de una tabla… necesitamos que sea *único*, justamente, y que en consecuencia el registro no esté vacío; por el contrario, si no los introducimos, la consola nos va a empezar a devolver errores porque ya va a haber valores nulos. En el otro caso defectuoso, si los introducimos repetidos, ya no se identificarán de manera única. Adicionalmente, esta restricción nos va a permitir hacer la unión entre una tabla y otra, o entre una entidad y otra. **Sépalo: Es la PRIMARY KEY la que nos ayuda realmente hacer las relaciones entre entidades**. Todo campo declarado con PRIMARY KEY es un índice o índex (ya explicaremos qué es esto).

Sin embargo, **una pregunta interesante** (porque puede ocurrir): **¿Qué podría pasar si se intenta repetir dos *id*, suponiéndose que se trata de dos índices únicos o *primary keys,* en una tabla? ¿Cómo podemos trabajar alrededor de eso y hacer que la base de datos trabaje a nuestro favor?,** esdecir, que no nos mande errores...

A esta eventualidad se le conoce como: ***Duplicate key.*** Cuando se intenta repetir dos *primary key* la consola, por defecto, no correrá el código; justamente porque se trata de un error, no es buena práctica que se repiten dos índices que, se supone, son únicos. Un ej. de cómo se ve este error: ****** .

Ahora, hay diferentes formas de tratar esto (en caso que queramos continuar por encima del error mismo… y/o estemos comprometidos a resolverlo; porque realmente, lo ideal, es que no le toque duplicar ninguna entrada). El recurso o la palabra reservada **ON DUPLICATE KEY** es la que nos habilita un conjunto de posibles resoluciones como, por ejemplo, para los casos en los que se necesita aceptar una entrada duplicada de un atributo *Unique* o *Primary Key;* es decir, tener dos o más de dos registros con atributos *Unique* o *Primary Key* repetidos. Sin embargo, ya se dará cuenta que esto no es una solución correcta.

**1ra opción** (nunca lo intente, se enseñará sólo por fines académicos): **ON DUPLICATE KEY IGNORE ALL,** lo que hace es literalmente ignorar el error de la entrada duplicada y ejecutarla tras bambalinas sin problema alguno (NO LO HAGA NUNCA). **Una aplicación que no lance errores, peor, que no le haga caso a los errores, es una aplicación que está condenada al fracaso.**

**2da opción (ideal): ON DUPLICATE UPDATE.** Generalmente cuando se intenta registrar una entrada duplicada no se tiene la intención real de hacerlo, es decir, por lo general se debe a una falla humana o un inconsciente. **Un escenario bastante común, en este tipo de potenciales errores, es que quizás lo que se haya intentado hacer era cambiar el registro de un atributo (lo cual se hace con un *update* y bajo otra sintaxis) y lo que realmente se provocó fue, *por accidente,* intentar hacer dicho cambio (dicho *update*) por medio de la inserción de un nuevo registro (insert), conservando todos los demás valores iguales en los otros atributos (incluyendo los *Unique* y/o *Primary Keys*);** es decir, se intentó correr un ***insert*** en vez de un ***update.***

**De hecho, no es un caso tan lejano, por ejemplo,** tenemos la tarea de insertar inicialmente los 4 primeros registros para una tabla llamada “***clients***”, tal que así:



Sin embargo, se desea *cambiar* (update) el valor de *active,* de 1 a 0, del cliente con *id = 4,* es decir, pasar *active* de 1 a 0 para Pedro Sanchez...

Esto se hace con una sintaxis completamente diferente que tiene que ver con el comando ***update.*** Lastimosamente, ¿Qué pasa muchas veces? Pasa que, ignorando completamente la sintaxis estructural del comando ***update,*** intentamos cambiar el valor de algún atributo, en este caso *active* (de 1 a 0)*,* con la sintaxis de un comando ***insert.***

Continuando el mismo ejemplo, así se vería más o menos el código con el comando con *insert*:



Esto nos bota error de inmediato, tal que así: 

Entonces, para corregir este error, bajo esa misma línea de código, podríamos hacer lo siguiente (en congruencia a nuestro objetivo que es cambiar *active* de *1 a 0* y no duplicar ninguna entrada):



Esto lo que hará es, justamente, definirle un nuevo valor al registro de *active,* lo que se quería hacer en nuestra última sentencia (lo que pretendíamos ejecutar y fue rechazado por *Duplicate key*); es decir, pasar *active* de 1 a 0.

A lo sumo se vería así:



Resultado:



Fue un éxito! Observe que se cambió el valor de *active* para el registro 4*,* lo que realmente se quería, sin modificar el resto del código ni duplicar nuestras *primary keys*!

**FOREIGN KEY:** Se traduce como *llave foránea.* Por ejemplo, cuando queremos juntar dos tablas y decir que “esta tabla” está relacionada con “esta otra”, lo que hacemos es que la *primary key* de una tabla se añade como *foreign key* dentro de otra. De la anterior manera es que se liga una tabla con otra (para poder relacionarlas posteriormente). Esta es la función de una *foreign key,* es una llave foránea que viene de otra tabla, viene de afuera. La *foreign key* debe tener las mismas características que la *primary key*; es decir, que sean del mismo tipo; pero, en este caso, no tienen que ser *únicas* porque nuestra clave foránea sí se puede repetir en otras tablas. Pongamos en contexto esto para que se entienda bien. Por ejemplo, tenemos tres entidades: *usuarios, posts y comentarios;* el ***usuario*** tiene un atributo de clave única que es **ID**; es decir, su *primary key;* si uno le pasa esta *primary key* a las entidades ***comentarios*** *y* ***posts*** pasaría a ser ***foreign keys***; esto se hace de forma intencional para poder saber con precisión qué usuario fue quién escribió tanto el *comentario(s)* como el *post(s).* Más adelante revisaremos los atributos propios a las entidades ***comentarios*** *y* ***posts*** y se dará cuenta que, realmente, no tienen ningún atributo que nos permita saber quién fue el ***usuario*** que las escribió; entonces, en ese caso, se hace necesario el uso de una ***Foreign key.***

**Importante:** Es preciso mencionar que, por lo general, se ve que en la *relación* entredos entidades; la que termina exportando su **primary key,** a otra entidad que la recibe como **foreign key**, es la entidad *univalor* *(uno, 1)* en términos de cardinalidad,... esto si la comparamos con otra entidad que sería *multivalor (muchos, n);* es esta última la que adapta una **foreign key.** Ahora, cuando tienes una cardinalidad **1:1,** no importa a qué entidad le pongas la referencia de la otra; es indistinto (eliges la que quieras).

**CHECK:** Nos permite definir una regla, la que sea que queramos, que hace un símil con el mundo real; y que nos permite, justamente, hacer reglas de negocio y reglas que obedezcan a lo que nuestro cliente o nuestro caso de uso (proceso) nos está pidiendo. Es como un condicional.

**DEFAULT:** Cuando tú no quieres que el tipo de dato por defecto sea NULL, al no meter un dato dentro del registro de un campo; puedes definir, por medio de DEFAULT, que sea de otro valor (asignado por ti mismo). Sirve especialmente en los casos en que, por ejemplo, digamos, hay registros de inventario que llenar y como bien sabe se llenan con valores numéricos; entonces, podemos hacer que nuestro sistema sobre-entienda que, ante un campo o registro vacío, esto se interprete como que no hay existencia para una determinada mercancía; entonces, en ese caso, es más conveniente que el valor por default sea 0 en vez de NULL. Ej.:

../../../../../../Captura%20de%20pantalla%202022-02-13%20a%20las%2012.00.20%20a.m..png

**INDEX:** Lo que nos va a permitir es hacer búsquedas más rápidas en nuestra tabla de base de datos, esto es muy útil cuando tenemos muchos registros; ahora, el índice tiene una desventaja y es que cuando existe un índice en una columna, cada que añadimos registros, se vuelve más lento porque cada que añades un nuevo registro tienen que volver a indexar (o indizar) toda la info. de la base de datos.

***Normalización (3er elemento del diagrama físico).***

**Lanormalización, como su nombre lo indica, nos ayuda a dejar todo de una forma *normal;* esto obedece a las 12 reglas (o mandamientos) de Codd: nos permiten separar cada componente de la base de datos, de tal forma que, se convierta en una base de datos relacional por excelencia, eso es *normalizar*. El tener una base *normalizada* es como alcanzar el Nirvana.**

Gráficamente, cómo se vería una base de datos (o tabla) ***sin normalizar?***



**La idea es separar la información, por medio de las 12 reglas de Codd, y lograr una normalización de la tabla para una base de datos.**

**Primera forma normal (1FN): sobre los atributos atómicos (sin campos repetidos).**

*Esta vendría siendo la regla núm. 1 de las 12 reglas de Codd; que son, precisamente, las que nos ayuda a separar la información.*

Los atributos atómicos significa que no podemos tener campos repetidos; en nuestro caso, cuando nos referimos a “campos repetidos”, estamos hablando de la siguiente barra azul:



Esta barra azul, si no estoy mal, corresponde a los *atributos* de las entidades; en este caso, como ya se dará cuenta, la entidad es *alumnos*,... sólo que en esta tabla no la muestran (recuerde que las entidades se escriben en plural). Ahora, el atributo que se repite es “Materia”, más allá de que haya “materia\_1” y “materia\_2”, este es nuestro campo repetido y se debe suprimir. Entonces, cuando aplicamos la *primera forma normal* lo que nos queda es una tabla más o menos así (ya incluyendo la mención a la **entidad** y también un atributo clave, el **id):**



Si se da cuenta ya no hay un campo de “Materia” repetido, ya no hay dos columnas para ella; sino que, sólo tenemos “Materia” y ahora tenemos dos (2) renglones por cada alumno. Ahora el espacio se extiende hacia abajo y no hacia un lado; esto, también, es con intención; pues, nos reduce o *atomiza* los datos que tenemos inicialmente.

**Segunda forma normal (2FN): cumple 1FN y cada campo de la tabla debe depender de una clave única (y cada fila también).**

Si aplicamos esta regla, nuestra tabla tenderá a separarse o a *partirse en dos.* Si bien es cierto que cada alumno tiene una clave única, la clave se está repitiendo en las filas (última tabla). Juanito es el alumno 1 y Pepito el alumno 2, pero tenemos dos veces *uno* y dos veces *dos:***esto no puede permitirse en una base de datos porque, como se había comentado antes, no identifica de manera única a un renglón o fila.** Entonces, en ese orden tenemos que, con la aplicación de la segunda forma normal, nuestra tabla quedaría así (quedarían):



Tuvo que tratarse al atributo “materia” como una entidad separada “materias” (quizás por ser multivaluable y por ser la que nos obligaba a asistirnos **dos veces** del mismo “alumno\_id”; por lo cual, se le da un tratamiento particular por separado); esto para así lograr por medio del nuevo atributo “materia\_id”, de la nueva entidad “materias”, identificar una clave única para cada fila.

Ahora si se percata, aunque dos tablas estén ahora completamente separadas; siempre deben conservar, al menos, un campo que se presente en ambas; en este caso, sería el atributo: “alumno\_id”,... esto para poder relacionarlas al final.

***Entonces, hasta ahora, no puede haber ningún atributo repetido por cada tabla o entidad (los que se ubican de forma horizontal en la primera fila o registro); ni, tampoco, repetirse el valor de la primera columna; es decir, hay que pasarle una clave única, que suele ser la Primary Key, a los que se ubican de forma vertical en la primera columna de cada tabla... esto, sin importar, que toque dividir tablas (separar) y le toque tratar a un atributo en particular (o a varios) como una entidad independiente separada.***

**Tercera forma normal (3FN): cumple 1FN, cumple 2FN y los campos que NO son clave NO deben tener dependencias.**



Ahora, si te fijas, tenemos los cursos separados; es así porque conceptualmente, y aunque los cursos pudieran ser, sin problema alguno, atributos de un alumno,... son dos (2) cuestiones separadas. Juanito, por ejemplo, no tiene intrínsecamente ligada la maestría; es decir, la maestría no sólo se dicta a Juanito; en consecuencia, los *cursos* sonun atributo multivaluable (que puede recibir varios valores; en este caso, puede ser dictado a varios alumnos). Entonces, como sucede con *un alumno y sus cursos*; un sólo curso, también, podría tener muchos alumnos de ellos. **Aclarado que los “cursos” son un atributo multivaluable, como ya se imagina, podemos separarlos como una entidad aparte; y que, se podría relacionar con la entidad “alumnos” aún, por medio del atributo “curso\_id”.**

*Dato curioso: Dese cuenta que los atributos multivaluables, ahora como entidades separadas, no se relacionan directamente entre sí; sino, que todas se relacionan directamente es con la entidad “alumnos”; es decir, con la entidad principal de nuestra base de datos.*

**Cuarta forma normal (4FN): cumple 1FN, 2FN y 3FN y los campos multivaluados se identifican por una clave única.**



Se tuvo que separar, aún todavía más, la entidad “materias”; esto porque, **si queremos alcanzar un grado de normalización *suprema* tenemos que evitar a toda costa que se repitan los valores textuales, incluso, de los atributos de cada una de las entidades**. Si se da cuenta aún siguen repitiéndose, en la entidad “materias”, y más precisamente en su atributo “materia”, los valores: MySQL (dos veces) y Python (dos veces también); lo cual, toca suprimir; es decir, realizar más “separaciones”.

En la nueva entidad “materias\_por\_alumno” lo que ve definido como el atributo “mpa\_id” es solamente una clave única artificial, de esa nueva entidad, que hemos definido manualmente para poder diferenciar, de forma secuencial, los atributos “materia\_id” y “alumno\_id” de esa misma tabla. Además sería ésta, nuestra última tabla resultante, la que hará la vinculación o *relación* entre la entidad “alumnos” con las *materias* por medio del atributo “alumno\_id”.

**Ahora, si te fijas, tenemos la misma información que al principio y, aunque pueda parecer que esto es más rebuscado y tenemos más tablas que al inicio, esto al final nos va a ayudar: 1. A que la base de datos, como computadora que es, entienda las diferencias entre uno y otro; pero, además (lo más importante), también nos va a ayudar a permitir hacer uniones que de inicio, tal vez, no teníamos pensadas. De inicio sólo teníamos dos renglones; pero, en este caso, ya podemos ligar por ejemplo un nuevo alumno a toda esta estructura. Podemos meter a un nuevo alumno que tenga las mismas dos materias o podemos meter una nueva materia y ligarla a Juanito y Pepito: nos da mucho más flexibilidad de hacer estas cosas con una tabla cada vez más *normalizada* y sin datos repetidos*. En resumen, tener una base de datos normalizada significa que cada cosa va en su tabla y éstas se relacionan, solamente, con valores enteros.***

***Aclaración importante:*** Los **Campos** serían cada una de las columnas por separadas; donde, cada una, guarda un conjunto de datos que sí o sí tienen que ser de un mismo tipo; por ejemplo, un campo podría ser el campo de los *alumnos* o el campo de *curso\_id*,... ahora,si se fija, son campos de un mismo registro, ¿y qué es **Registro**? Los registros serían cada una de las filas de una tabla (entidad) y no necesariamente deben ser del mismo tipo de datos; cada una de las celdas por cada fila, me refiero, no necesariamente deben conservar los mismos tipos de datos entre ellas, pero sí deben de relacionarse. En resumen, un registro es una colección de datos iguales o de diferentes tipos que están sí o sí relacionados. Con “relacionados” queremos decir que son datos que por sí solos no tienen sentido, pero que *relacionados* ya toman un sentido (brindan información relativamente completa sobre algo en concreto). Por último, el conjunto de ***campos*** y ***registros*** conforman lo que se llama ***Archivo;*** es decir, la *tabla* que contiene todo el conjunto de datos para una entidad en especifica. Los registros de cada tabla no se repiten, recuérdelo!

Ejemplo. ***Entidad Alumnos:***



Ahora, supongamos que yo tengo otro archivo que me proporciona una información de *materias* y calificaciones **relacionadas** a los *alumnos.* Veamos:



Si nosotros tenemos dos (2) archivos diferentes, pero que la vez están *relacionados;* estamos hablando de algo que se llama: **Base de datos.**

***Un dato:*** Cuando una entidad por su propia naturaleza no posee un atributo único o llave; sino que, al diseñador le toca pasarle una llave artificial a la entidad en cuestión,... hablamos de ***entidades débiles.*** Aquellas entidades como, por ejemplo, *carros* (que ya vienen con una placa propia y única para cada carro en sí) o *personas* (donde cada persona en sí se identifica de manera individual por medio de un DNI único e inigualable) son ***entidades fuertes.***

***Consejo de buena práctica:* Ninguna *tupla* o registro se borra jamás; si no se necesita, se *desactiva* (al igual se les puede activar cada que quiera).** Esto de “activar” o “desactivar” realmente es un juego de palabras, se trata de un *truco* (Realmente no hay un “switch” que nos permita activar o desactivar un registro)*;*es decir, realmente lo que se hace es crear una nueva columna o atributo que se nombrará como “active”, por conveniencia, y se definirá con un tipo de dato **TINYINT(1)** que puede recibir “1” (si el registro continua “activo”, es decir, *continua con normalidad o es lo que se espera*)o que puede recibir “2” (si el registro se “desactiva”, es decir, parcialmente no se le dará ningún tipo de uso a dicho registro). Adicionalmente, es una buena práctica que a este atributo también se le defina con el *Constrain* **NOT NULL** & **DEFAULT 1** (“DEFAULT 1” porque lo que se espera o lo normal es que ningún registro se “desactive”; entonces, si no se le pasa ni 1 ni 0, que la consola entienda automáticamente que el registro sigue “activo”).

En consola este caso se ve algo así: **

Se aconseja que todas las tablas, o entidades, tenga el anterior atributo definido dentro de ellas.

***Un caso especial en entidades o tablas tipo “operacionales”***: Las tablas tipo *Operacionales* tienen que ver directamente con hacer un seguimiento, registro a registro, al caso de uso o razón social de una compañía (para los casos en los que nuestro esquema o base de datos tenga que ver, o involucre, personas jurídicas con alguna finalidad comercial y/o no comercial).

Cuando una tabla consiste en evaluar las *tareas u operaciones* de una persona jurídica, una empresa por ejemplo, hay que determinar si la tarea ejecutada, cada una de ellas registradas fila por fila, ha sido terminada en sí misma o requiere esperar la confirmación/ejecución de una tarea secundaria que se correlaciona con la *primera*.

Más claro. Cuando hay *oficios, tareas u operaciones* que dependen de sub-operaciones para dar por fin terminado el oficio, la tarea u operación en cuestión… hay que dejarlo claro! **Y se deja claro en este tipo de entidades o tablas *operacionales* por medio de un atributo**. Incluso aunque la tarea sí haya sido finalizada por su propia acción o por esa única ejecución en sí, hay que dejarlo claro. A este atributo, por convención, se le conoce como: ***“finished”.*** Con el atributo “***finished***” se deja constanciade si una tarea, en su conjunto, ha finiquitado o, por el contrario, depende de una sub-tarea para ser finiquitada.

Por ejemplo, supongamos que tenemos una tabla o entidad de tipo *operacionales,* dentro de una base de datos, para una empresa X, que me evalúa tres tareas: Si la mercancía es vendida, prestada o devuelta. Si la mercancía es *vendida*, esto es una tarea auto-suficiente en sí misma, es decir, la empresa ya hizo la venta y no queda condicionada a *esperar algo de vuelta* para dar por fin a la tarea. Sin embargo, en los casos en que la empresa hace un *préstamo*, si bien el préstamo ya está ejecutado, la tarea en su conjunto no está finalizada; pues, la compañía aún espera recibir devuelta en una *devolución* el préstamo que concedió. Entonces, el *préstamo* depende directamente de una *devolución* para dar, por fin, terminada toda la tarea comercial en su conjunto.

**Dicho esto, cada una de estas acciones, casos de uso, se registran bajo la lógica anterior. El hecho de establecer si una tarea fue finalizada o no es algo que se deja evidenciado, registro a registro, cada que se ejecute la tarea en cuestión; es decir, cada vez que una tabla “operacional” haga un registro. Tenga claro esto!**

El tratamiento del atributo ***finished***es muy similar al del atributo ***active***; pues, básicamente consiste en pasarle un veredicto final, de dos posibles, al registro del atributo en cuestión: decir si la operación, en su conjunto, *finalizo o No;* y eso lo podemos hacer justamente con el tipo de dato **TINYINT(1)** que puede tomar sólo dos valores posibles, en este caso: **1 ó 0. TINYINT(1)** se complementa también con **NOT NULL** para no dejar nunca el registro vacío. **TINYINT(1)** significaría que la operación fue finalizada & **TINYINT(0)** significaría que No!

**Ej.: **

Recuerde, tenemos que la ejecución de una operación principal puede estar finalizada en sí mismo o esperar que una sub-operación, si bien correlacionada, independiente la finalice; entonces, para ese último caso, tenemos que la ejecución total de una tarea principal es dependiente de una sub-operación, secundaria, que es quien determina si la tarea principal es finalizada o no. *Retomando el ejemplo anterior*, cuando se ejecuta un préstamo es realmente la devolución, si se hace, la que termina toda la tarea en su conjunto para la empresa X, tenga esto cuenta. **La finalización de la tarea del préstamo, en su conjunto, depende de si la devolución es ejecutada o no.**

***Diagrama físico: normalizando Platziblog.***

***Recordemos, antes que nada, el diagrama ER de Platziblog.***



***Ahora sí, Diagrama físico: donde se incluye el tipo de datos para cada variable y también su restricción, en caso la tenga.***



***Ahora las relaciones del Diagrama físico, pero con el uso de conectores y las respectivas foreign keys.***



***Dato:* Los atributos no pueden llamarse igual en ningún caso**, ni siquiera cuando estamos exportando una llave primaria de una entidad a otra y ésta última la recibe como llave foránea; por ejemplo, si su llave primaria se llama: “Clave\_C”, podría considerar que su llave foránea se llame: “Clave\_C1”. Ahora bien, las tablas que exportan su llave primaria y, además, conservan una relación de 1:M con respecto a otra tabla, precisamente las que reciben como llave foránea la llave importada de la primera tabla en mención, son las llamadas: ***tablas independientes.*** Luego, las tablas que reciben justamente una llave importada, llave foránea, frente a una relación M:1 que tengan con respecto a otra, se les llama: ***tablas dependientes.*** Ahora, dentro de un manejador de bases de datos, o en cualquier escenario realmente, lo correcto y lógico sería crear primero las tablas independientes que las dependientes, esto por una razón: **Hay que crear primero las tablas independientes para poder luego exportar sus llaves primarias a las tablas dependientes. En cada tabla dependiente que pretenda ser creada ya deben existir las tablas de las llaves foráneas que asume o pretende asumir, ojo.**

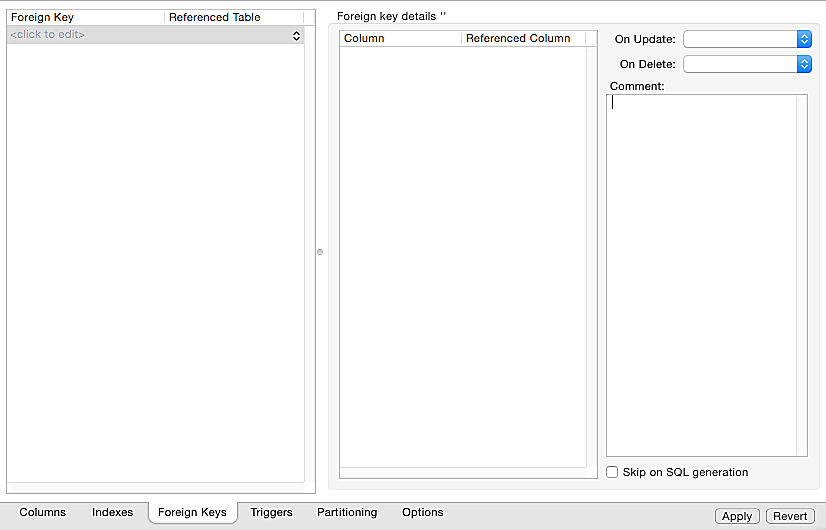
**Un adelanto: Cómo se configuran las llaves foráneas dentro de MySQL?**

Cuando uno está creando llaves foráneas para una tabla, dichos campos o atributos deben conservar las mismas características que su análogo en llave primaria. Ahora, en principio los atributos que se identifican como *foráneos* se configuran así luego de la creación de la tabla en cuestión, antes no; es decir, para indicar qué atributos o campos son foráneos, dichos atributos o campos ya deben estar creados antes dentro de la tabla. Posteriormente, por medio de una pestaña que se llama *foreign keys,* se establece qué columna es foránea. Veamos.



Sobre esta tabla, que ya creamos , es que vamos a determinar qué columnas o atributos son de tipo *foreign key;* Usted ya los conoce: **usuario\_id & categoria\_id.** Entonces, posteriormente, se irá a la pestaña ***foreign keys***.

, estando ahí verá lo siguiente:

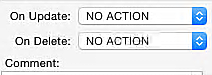


En **Foreign key,** debe pasar idealmente un nombre con el que logre relacionar la llave foránea de la tabla actual con la otra tabla que le exportó su llave primaria. Puede probar. Por ejemplo, tomemos nuestro caso: si quisiéramos configurar una llave foránea para el atributo **usuario\_id** de nuestra tabla actual **“posts”** con relación a la tabla “**usuarios”** en la columna **Foreign key** podría pasar el nombre: **posts\_usuarios.**

Ahora, en **Referenced table,** se debe indicar justamente la tabla que nos exportó su llave primaria; es decir, en nuestro caso actual, se debe seleccionar a la tabla **usuarios,** que se vería como: **‘platziblog’.‘usuarios’** *(hacen mención a* **‘platziblog’** *para dejar claro que es la tabla* **.‘usuarios’** *de la base de datos* **‘platziblog’***).*

Posteriormente, en *,...* para***Column*** se elige la columna o atributo que queremos que sea ***Foreign Key*** de nuestra tabla ***posts;*** en este primer caso sería ***“usuarios\_id”*** y en ***Referenced Column*** se seleccionará automáticamente la columna o atributo que conserve la restricción ***PK (primary key)*** de la tabla de la que importamos su llave primaria; es decir, sería el atributo ***id*** de la tabla ***usuarios*** (tiene sentido que sea así, no de otra manera). Eso para la configuración de nuestra primera ***FK.*** Siga esta misma lógica y dinámica para configurar su segunda ***Foreign Key*** para el atributo **“*categoria\_id”.***

Más adelante explicaremos esta última parte:



En todo caso, en ***On Update,*** podemos configurar qué queremos que pase con nuestra tabla ***posts*** en caso que hayan cambios en la ***PK*** de ***usuarios.* No action** significa, justamente, “no hacer nada” ante cambios en la **PK** de laotra tabla; es decir, por ejemplo en este caso, ante cambios en la **PK** de la tabla ***usuarios,*** que eso no afecte en absoluto sobre nuestro análogo en ***posts (en el FK).*** Ahora, cuando definimos **No action** en ***On Delete;*** significa que, en caso que sea eliminada la **PK** de la tabla ***usuarios,*** no queremos que sea eliminado su análogo, la **FK,** en nuestra tabla ***posts.***

*Veamos este otro valor:*



Ahora, ademas de ***No Action*** tenemos el valor: ***Cascade.*** Con ***Cascade,*** para ambos escenarios, ***On Update & On Delete,*** queremos decir que se *emule* todolo que le pase a la ***PK***, por ejemplo en este caso, al atributo ***id*** de la tabla ***usuarios*** en su análogo en la tabla ***posts;*** ya sea en el caso en que se elimine dicha **PK** *(****id*** *de* ***usuarios****):*  o en el caso en que se modifique solamente dicha **PK** en cuestión ***(id*** *de* ***usuarios):***

Cómo se vería esto, ya en Lenguage SQL, si definimos que ***On Update*** sea ***Cascade*** y ***On Delete*** sea ***No Action?***



**Un adelanto: Tratamiento a la relación “muchos a muchos” (también dentro de MySQL).**

***En el tema de los conectores, muchos a muchos (N:N) es un caso especial; vamos a profundizar este último caso en la relación de las entidades: Posts & Etiquetas****,... pues un post puede tener varias etiquetas pero una etiqueta también puede estar en varios post.* Resulta que la cardinalidad *muchos a muchos* es, en efecto, un caso especial porque,... realmente no se sabe cuál podría ser la entidad que exportaría su *primary key* a otra entidad que la recibiría como *foreign key*. Dicho lo anterior, este tipo de eventualidades o, más bien, *cardinalidades* requieren un tipo de tratamiento más sofisticado y especial (casi que se da la sensación, también, de que se trata de forma aislada). Veamos esto en nuestro mismo caso de estudio: ***relación posts-etiquetas.***

En los casos *muchos a muchos*, los cuales deben ser *eliminados,* se rompe la relación entre las dos entidades (la relación de *muchos a muchos* justamente); esto para, inmediatamente luego, poner una tabla intermedia entre ellas. Esa tabla resultante, generalmente, recibe el nombre de ***tabla pivote*** y se encarga de mostrarnos cuál es la relación entre ambas entidades (a mayor profundidad).

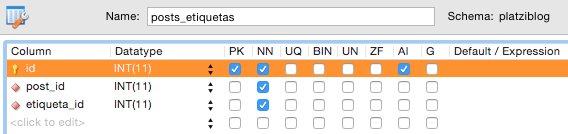
Esta tabla intermedia, *pivote,* logra esa nueva relación de una manera más precisa por medio de la conjugación de las *keys* de ambas entidades (clave compuesta)*.* Las entidades del extremo solamente conservarían su propia *primary key,* pero ninguna exporta la suya propia hacia su contraparte; sin embargo, cada una de las entidades, le exporta su llave primaria a la tabla intermedia,... quedando como resultado que la tabla intermedia, por lo anterior, reciba dos llaves foráneas. Es así como funciona. Caso aparte, es una buena práctica que esa tabla intermedia reciba como nombre un calificativo que referencie a las dos (2) entidades en discusión. Es decir, debe verse algo más o menos así (tomando nuestro caso):



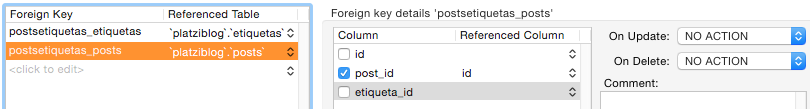
Se unen los nombres de ambas entidades de cada extremo para dar con la configuración de nuestra nueva tabla intermedia. Cuando hablamos de *claves compuestas* es que no sólo hay un id propio, no hay un sólo atributo con *id*, a la entidad en cuestión; sino, que tiene más de uno (1). En este caso, la tabla *posts\_etiquetas* tiene dos claves id que la definen (dos *foreign keys*): ***post\_id & etiqueta\_id;*** esto, al traerse las llaves primarias de cada una de las entidades al extremo; por lo que acá se introducen también, pero como llaves foráneas, las llaves propias a las entidades *posts & etiquetas* (dos *foreign keys*).

Ahora, crear estas tablas intermedias propiamente en MySQL no requieren de mucha novedad; sólo es crear la tabla, tal como lo dicta nuestro diagrama **posts\_etiquetas (con su propio *id* que hará de llave primaria -falto señalarlo)**, y luego configurar dos llaves foráneas para la misma tabla en cuestión (ésta es realmente la novedad, que en tablas *pivote* sí o sí se configurarán siempre dos llaves foráneas; pues, importará las llaves primarias de cada una de las tablas a sus dos extremos).

La tabla *pivote* ***posts\_etiquetas*** creada en MySQL:

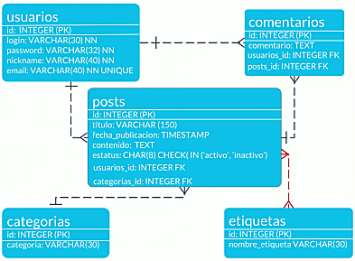


Configuración de sus ***foreign keys,*** luego de creada, en MySQL:



**Ahora, algo muy interesante que te gustará mucho...**

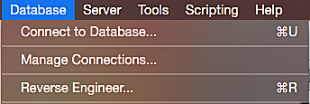
**Recuerdas nuestro diagrama ER de la base de datos *platziblog,* con sus tablas y atributos,elaborada manualmente por nosotros? Esta:**



**Pues bien, esta misma se puede representar visualmente dentro de nuestro manejador de base de datos MySQL (Workbench).**

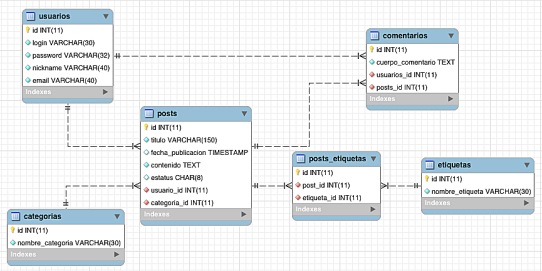
La tecnología que logra la representación gráfica del diagrama del modelo entidad-relación, de nuestra base de datos, se llama: ***Reverse Engineer*** *(Ingeniería Inversa).*

Para ubicarla nos debemos situar sobre la pestaña, o menú del Workbench, que se llama *Database*:



**Honestamente, para hacerlo más fácil, debe darle *Sí a todo.***

***El resultado, el siguiente:***



**Visualmente esto orienta mucho en los casos que, por ejemplo, la base de datos ya está estructurada y eres “nuevo en el trabajo”.**

***Elaborado por Roberto Velasquez en base al Curso de Platzi: Fundamentos de Bases de Datos.***