akus.net
Diseño Web

(https://disenowebakus.net)

Cada vez que tengamos que crear una tabla que sirva para almacenar datos de una aplicación Web, debemos poner a prueba nuestra capacidad para definir los tipos de datos que con **mayor eficiencia** puedan almacenar cada dato que necesitemos guardar.

Los campos de las tablas MySQL nos dan la posibilidad de elegir entre **3 tipos de contenidos**:

Datos numéricos

- TINYINT
- SMALLINT
- MEDIUMINT
- INT o INTEGER
- BIGINT

Datos para guardar cadenas de caracteres (alfanuméricos)

- CHAR
- VARCHAR
- BINARY
- VARBINARY

TINYBLOB

TINYTEXT

BLOB

TEXT

MEDIUMBLOB

MEDIUMTEXT

LONGBLOB

LONGTEX

ENUM

SET

Datos para almacenar fechas y horas

DATE

DATETIME

TIME

TIMESTAMP

YEAR

Atributos de los campos

NULL

DEFAULT

BINARY

INDEX

PRIMARY KEY

AUTO_INCREMENT

UNIQUE

FULLTEXT

Desde ya que es muy obvio poder distinguir a cuál de los tres grupos corresponderá, por ejemplo, un campo que guarde la "**edad**" de una persona: será un **dato numérico**.

Pero dentro de los distintos tipos de datos numéricos, ¿Cuál séra la mejor opción?

Un número entero, pero, ¿Cuál de los distintos tipos de enteros disponibles?

¿Qué tipo de dato permitirá consumir menor espacio físico de almacenamiento y brindará la posibilidad de almacenar la cantidad de datos que se espera almacenar en ese campo? (dos dígitos, o máximo tres, en el caso de la edad).

Esas son preguntas que sólo podremos responder a partir del conocimiento de los distintos **tipos de datos** que nos permite MySQL. Vamos, entonces, a analizar los usos más apropiados de cada uno de estos tres grandes grupos.

La diferencia entre uno y otro tipo de dato es simplemente el **rango de valores** que puede contener.

Dentro de los datos numéricos, podemos distinguir dos grandes ramas: **enteros** y **decimales**.

Comencemos por conocer las opciones que tenemos para almacenar datos que sean números **enteros** (edades, cantidades, magnitudes sin decimales); poseemos una variedad de opciones:

Tipos de datos	Bytes	Valor mínimo	Valor máximo
TINYINT	1	-128	127
SMALLINT	2	-32768	32767
MEDIUMINT	3	-8388608	8388607
INT o INTEGER	4	-2147483648	2147483647
BIGINT	8	-9223372036854775808	9223372036854775807

Veamos un ejemplo para comprender mejor qué tipo de dato nos conviene elegir para cada campo.

Si necesitamos definir un campo para almacenar la "**edad**" de nuestros usuarios, sería suficiente con asignar a ese campo un tipo de dato **TINYINT**, que permite almacenar como máximo el valor de 127 (es decir, por mas que tenga tres dígitos, no nos dejará almacenar un 999, ni siquiera un 128, solo un número hasta el número **127** inclusive).

Como la edad posible de las personas queda incluida en ese rango, es suficiente un TINYINT.

Ahora bien, si queremos definir el tipo de dato para el campo **id** (identificador) de la tabla de productos de un gran mercado que vende varios miles de artículos diferentes, ya no séra suficiente con un TINYINT, y deberemos conocer con mayor precisión la cantidad de artículos distintos que comercializa (en la actualidad y la cantidad prevista en un futuro próximo, para que nuestro sistema de almacenamiento no quede obsoleto rápidamente).

Suponiendo que **SMALLINT**, ya que nos permitirá numerar hasta algo más de **32,000** artículos (tal como vimos en el cuadro anterior).

En el supuesto de que el campo **id** deba utilizarse para una tabla de clientes de una empresa telefónica con 5 millones de usuarios, ya no nos servirá un SMALLINT, sino que deberíamos utilizar un **MEDIUMINT**.

En el caso de que esa empresa tuviera 200 millones de clientes, deberíamos utilizar un campo de tipo **INT**.

Asimismo, si quisieramos definir un campo que identifique a cada uno de los seres humanos que habitamos en el planeta, deberemos recurrir a un campo **BIGINT**, ya que el tipo **INT** sólo perimite hasta **2 mil millones de datos diferentes**, lo cual no nos alcanzaría.

> *Vemos, entonces, que hay que considerar siempre cuál será el valor maximo que se almacenará dentro de un campo, antes de elegir el tipo de dato más adecuado.*

Pero no sólo los valores máximos deben considerarse, también debemos tener en cuenta los **valores mínimos** que pudieran almacenarse.

Por ejemplo, para guardar el puntaje de un juego, que pudiera tomar **valores negativos**, o el saldo de una cuenta corriente, o una tabla que incluya valores de temperatura bajo cero, y casos similares.

En el cuadro anterior hemos visto que cada tipo de dato posee valores mínimos negativos simétricos a los valores positivos máximos que podía almacenar.

Ahora bien: existe la posibilidad de **duplicar** el limite de valor máximo positivo de cada tipo de dato, si eliminamos la posibilidad de almacenar valores negativos.

Pensemos en los ejemplos anteriores: la edad no tiene sentido que sea negativa, entonces, si eliminamos la posibilidad de que ese campo almacene valores negativos, duplicaríamos el limite positivo de almacenamiento, y el campo de tipo **TINYINT** que normalmente permitía almacenar valores del **-128** al **127**, ahora dejará almacenar valores desde el **0** hasta el **255**.

Esto puede ser útil para almacenar precios, cantidades de objetos o magnitudes que no puedan ser negativas, etc.

Observemos en la tabla cómo se duplican los valores máximos **sin signo** y luego aprenderemos cómo configurar esta posibilidad de quitar el signo desde **phpMyAdmin**:

Tipo de dato	Bytes	Valor mínimo	Valor máximo
TINYINT	1	0	255
SMALLINT	2	0	65535
MEDIUMINT	3	0	16777215
INT o INTEGER	4	0	4294967295
BIGINT	8	0	18446744073709551615

¿Cómo definimos que un campo no tiene signo? Mediante el modificador **UNSIGNED** que podemos definirle a un campo numérico:

Elegimos en la columna **Atributos** el valor de **UNSIGNED** y este campo ya no podrá contener valores negativos, duplicando su capacidad de almacenamiento (en el caso de ser este tipo entero, pronto veremos que en los tipos de coma flotante, si se lo define como UNSIGNED no altera el valor máximo permitido).

Comentemos al pasar, que es importante que, en el momento de definir un campo en la columna "**Longitud**" escribamos un número coherente con la capacidad de almacenamiento que acabamos de elegir.

Por ejemplo, en un TINYINT para la edad, colocaremos como longitud un tres, y no un número mayor ni menor.

Dejemos los enteros y pasemos ahora a analizar los valores numéricos **con decimales**.

Estos tipos de datos son necesarios para almacenar precios, salarios, importes de cuentas bancarias, etc. que no son enteros.

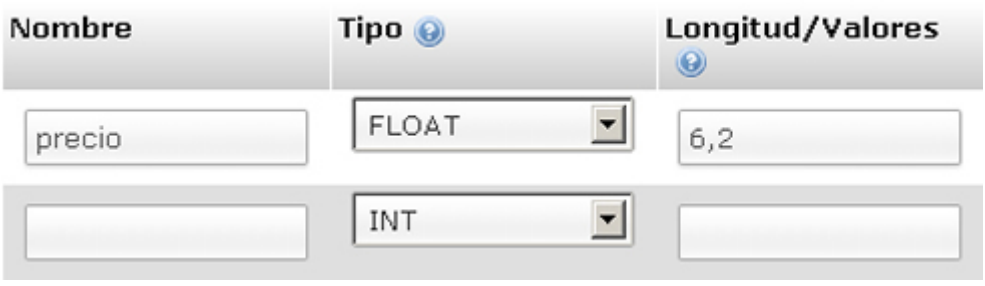
Tenemos que tener en cuenta que si bien estos tipos de datos se llaman "**de coma flotante**", por ser la coma el separador entre la parte entera y la parte decimal, en realidad MySQL los almacena usando un **punto** como separador.

En esta categoría, disponemos de tres tipos de datos: **FLOAT**, **DOUBLE** y **DECIMAL**.

La estructura con la que podemos declarar un campo **FLOAT** implica definir dos valores: la **longitud total** (incluyendo los decimales y la coma), y cuántos de estos dígitos son la **parte decimal**. Por ejemplo:

FLOAT (6.2)

Esta definición permitirá almacenar como minimo el valor -999.99 y como máximo 999.99 (el signo menos no cuenta, pero el punto decimal sí, por eso son seis digitos en total, y de ellos dos son los decimales).



La cantidad de decimales (el segundo número entre los paréntesis) debe estar entre **0** y **24**, ya que ése es el rango de precisión simple.

En cambio, en el tipo de dato **DOUBLE**, al ser de doble precisión, sólo permite que la cantidad de decimales se defina entre 25 y 53.

Debido a que los cálculos entre campos en MySQL se realizan con doble precisión (la utilizada por DOUBLE) usar **FLOAT**, que es de simple precisión, puede traer problemas de redondeo y pérdida de los decimales restantes.

Por último, **DECIMAL** es ideal para almacenar valores monetarios, donde se requiera menor longitud, pero la "máxima exactitud" (sin redondeos).

Este tipo de dato le asigna un ancho fijo a la cifra que almacenará.

El máximo de dígitos totales para este tipo de dato es de **64**, de los cuales 30 es el número de decimales máximo permitido. Más que suficientes para almacenar precios, salarios y monedas.

El formato en el que se definen en el phpMyAdmin es idéntico para los tres: primero la longitud total, luego, una coma y, por ultimo, la cantidad de decimales.

Para almacenar datos alfanuméricos (cadenas de caracteres) en MySQL poseemos los siguientes tipos de datos:

- CHAR
- VARCHAR
- BINARY
- VARBINARY
- TINYBLOB
- TINYTEXT
- BLOB
- TEXT
- MEDIUMBLOB
- MEDIUMTEXT
- LONGBLOB
- LONGTEX
- ENUM
- SET

Veremos ahora cuáles son sus caracteristicas y cuáles son las ventajas de usar uno u otro, según qué datos necesitemos almacenar.

Comencemos por el tipo de dato alfanumérico mas simple: **CHAR** (character, o caracter).

Este tipo de dato permite almacenar textos breves, de hasta **255 caracteres de longitud como máximo** en caracteres que le definamos, aunque no lo utilicemos.

Por ejemplo, si definieramos un campo "nombre" de 14 caracteres como CHAR, reservará (y consumirá en disco) este espacio.

1	2	3	4	5	6	7	8	9	10	11	12	13	14
J	u	a	n		P	e	r	e	z				
C	a	r	l	o	s		G	a	r	c	i	a	
J	o	s	e		R	a	m	i	r	e	z		
L	u	i	s		F	e	r	n	a	n	d	e	z
P	e	p	e		L	o	p	e	z				

Por lo tanto, no es eficiente cuando la longitud del dato que se almacenará en un campo es desconocida *a priori* (tipicamente, datos ingresados por el usuario en un formulario, como su nombre, domicilio, etc.)

¿En qué casos usarlo, entonces? Cuando el contenido de ese campo será completado por nosotros, programadores, al agregarse un registro y, por lo tanto, estamos seguros de que la longitud **siempre será la misma.**

Pensemos en un formulario con botones de radio para elegir el "sexo"; independientemente de lo que muestren las etiquetas visibles para el usuario, podríamos almacenar un solo caracter **M** o **F** (masculino o femenino) y, en consecuencia, el ancho del campo CHAR podría ser de un digito, y sería suficiente. Lo mismo sucede con códigos que identifiquen provincias, países, estados civiles, etc.

Completariamente, el tipo de dato **VARCHAR** (*character varying*, o caracteres variables) es útil cuando la longitud del dato es **desconocida**, cuando depende de la información que el usuario escribe en campos o áreas de texto de un formulario.

La longitud máxima permitida era de **255 caracteres** hasta MySQL 5.0.3. pero desde esta versión cambio a un **máximo de 65.535 caracteres**.

Este tipo de dato tiene la partícularidad de que cada registro puede tener una **longitud diferente**, que dependerá de su contenido; si en su registro el campo "nombre" (supongamos que hubiera sido definido con un ancho máximo de 20 caracteres) contiene solamente el texto: "Pepe", consumirá sólo cinco caracteres, cuatro para las cuatro letras, y uno más que indicará cuántas letras se utilizaron.

Si luego, en otro registro, se ingresa un nombre de 15 caracteres, consumirá 16 caracteres (siempre uno más que la longitud del texto, mientras la longitud no supere los 255 caracteres; si no los supera, serán dos los bytes necesarios para indicar la longitud).

Por lo tanto, será más eficiente para almacenar registros cuyos valores tengan **longitudes variables**, ya que si bien "gasta" uno o dos caracteres por registro para declarar la longitud, esto le permite ahorrar muchos otros caracteres que no serían utilizados.

En cambio, en el caso de datos de longitud siempre constante, sería un desperdicio gastar un caracter por registro para almacenar la longitud, y por eso convendría utilizar CHAR en esos casos.

1	2	3	4	5	6	7	8	9	10	11	12	13	14
J	u	a	n		P	e	r	e	z				
C	a	r	l	o	s		G	a	r	c	i	a	
J	o	s	e		R	a	m	i	r	e	z		
l	u	i	s		F	e	r	n	a	n	d	e	z
P	e	p	e		L	o	p	e	z				

Estos dos tipos de datos son identicos a CHAR y VARCHAR, respectivamente, salvo que almacenan *bytes* en lugar de caracteres, una diferencia muy sutil para un nivel básico a intermedio de MySQL.

Antes de la versión 5.0.3. de MySQL, este campo era el utilizado "por excelencia" para descripciones de productos, coméntarios, textos de noticia, y cualquier otro **texto largo**.

Pero, a parir de la posibilidad de utilizar VARCHAR para longitudes de hasta 65.535 caracteres, es de esperar que se utilice cada vez menos este tipo de campo.

La principal **desventaja de TEXT** es que no puede indexarse facilmente (a diferencia de VARCHAR).

Tampoco se le puede asignar un valor **predeterminado** a un campo TEXT (un valor por omisión que se complete automaticamente si no se ha proporcionado un valor al insertar un registro).

Sólo deberíamos utilizarlo para textos realmente muy largos, como los que mencionamos al comienzo de este parrafo.

Es un campo que guarda información en **formato binario** y se utiliza cuando desde PHP se almacena en la base de datos el contenido de un archivo binario (típicamente, una **imagen** o un archivo comprimido **ZIP**) leyéndolo byte a byte, y se requiere almacenar todo su contenido para luego reconstruir el archivo y servidor al navegador otra vez, sin necesidad de almacenar la imagen o el ZIP en un disco, sino que sus bytes quedan guardados en un campo de una tabla de la base de datos.

El tamaño máximo que almacena es de **65.535 bytes**.

De todos modos, y como lo hemos mencionado en este ejemplo, respecto al tipo de dato para una imagen, usualmente no se guarda "la imagen" (sus bytes, el contenido del archivo) en la base de datos porque, un sitio grande, se vuelve muy pesada y lenta la base de datos, **sino que almacena sólo la URL que lleva hasta la imagen**.

De esa forma, para mostrar la imagen simplemente se lee ese **campo URL** y se completa una etiqueta **img** con esa URL, y esto es suficiente para que el navegador muestre la imagen. Entonces, con un **VARCHAR** alcanza para almacenar la URL de una imagen.

El campo BLOB es para almacenar directamente "la imagen" (o un archivo comprimido, o cualquier otro archivo binario), **no su ruta**.

Similares al BLOB, sólo cambia la longitud máxima:

- **TINYBLOB** es de **255 bytes**
- **MEDIUMBLOB** es de **16.777.215 bytes**, y
- **LONGBLOB** es de **4 Gb** (o lo máximo que permita manipular el sistema operativo).

Su nombre es la abreviatura de "**enumeración**". Este campo nos permite establecer cuáles serán los valores posibles que se le podrán insertar.

Es decir, crearemos una lista de **valores permitidos**, y no se autorizará el ingreso de ningún valor fuera de la lista, y se permitirá elegir **solo uno de estos datos** como valor del campo.

Los valores deben estar separados por comas y envueltos entre comillas simples.

El máximo de valores diferentes es de **65.535**.

Lo que se almacenará no es la cadena de caracteres en sí, sino el número de índice de su posición dentro de la enumeración.

Por ejemplo, si al crear la tabla definimos un campo de esta manera:

En este ejemplo, la categoría será **excluyente**, no podremos elegir más de una, y todo docente (uno por registro) deberá tener una categoría asignada.

Su nombre significa "**conjunto**". De la misma manera que ENUM, debemos especificar una lista, pero de hasta **64** opciones solamente.

La carga de esos valores es idéntica a la de ENUM, una lista de valores entre comillas simples, separados por comas. Pero, a diferencia de ENUM, sí podemos llegar a dejarlo vacío, sin elegir ninguna opción de las posibles.

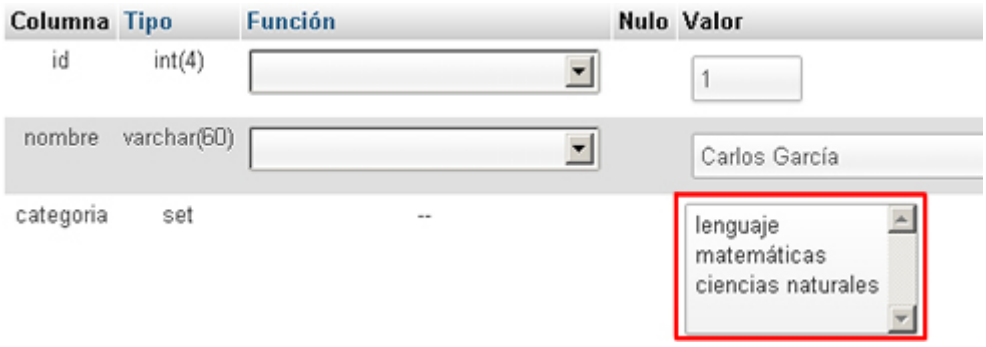
Y también podemos elegir como valor del campo más de uno de los valores de la lista.

Por ejemplo, darnos a elegir una serie de temas (típicamente con casillas de verificación que permiten selección múltiple) y luego almacenamos en un solo campo todas las opciones elegidas.

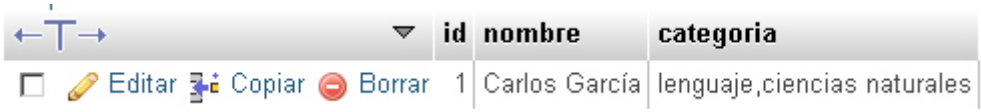
Un detalle importante es que cada valor dentro de la cadena de caracteres no puede contener comas, ya que es la coma el separador entre un valor y otro.

Unas vez definido este tipo de dato, podemos cargar valores **múltiples** para ese campo dentro de un mismo registro, pulsando " **Control**" mientras hacemos clic en cada opción.

Eso significará, en este ejemplo de una tabla de alumnos, que ese alumno está cursando ambas materias seleccionadas.



(imagenes/materias/programacion-en-internet-2/asignando-valores-multiples.jpg)
Si una vez agregado algún registro pulsamos en "**Examinar**" para ver el contenido de la tabla, veremos que este registro, que en un campo contenía un valor múltiple, incluye lo siguiente:



Con esto damos por finalizado el grupo de tipos de datos alfanuméricos. Pasemos ahora al último grupo, el de fechas y horas.

En MySQL, poseemos varias opciones para almacenar datos referidos a fechas y horas.

Veamos las diferencias entre uno y otro, y sus usos principales, así podemos elegir el tipo de dato apropiado en cada caso.

El tipo de dato DATE nos permite almacenar fechas en el formato: **AAAA-MM-DD** (los cuatro primeros dígitos para el **año**, los dos siguientes para el **mes**, y los ultimos dos para el **dia**).

Atención:
En los países de habla hispana estamos acostumbrados a ordenar las fechas en Día, Mes y Año, pero para MySQL es exactamente al revés.

Tengamos en cuenta que esto nos obligará a realizar algunas maniobras de reordenamiento utilizando funciones de manejo de caracteres.

Si bien al leer un campo **DATE** siempre nos entrega los datos separados por guiones, al momento de **insetar** un dato nos permite hacerlo tanto en formato de número continuo (por ejemplo, 201512319, como utilizando cualquier caracter separador (2015-12-31 o cualquier otro caracter que separe los tres grupos).

El rango de fechas que permite manejar desde el **1000-01-01** hasta el **9999-12-31**.

Es decir, que no nos será útil si trabajamos con una línea de tiempo que se remote antes del año 1000, (¿alguna aplicación relacionada con la historia?), pero si nos resultara útil para datos de un pasado cercano y un futuro muy largo por delante, ya que llega casi hasta el año 10.000.

Un campo definido como **DATETIME** nos permitirá almacenar información acerca de un instante de tiempo, pero no sólo la fecha sino también su horario, en el formato:

AAAA-MM-DD HH:MM:SS

Siendo la parte de la fecha de un rango similar al del tipo DATE (desde el **1000-01-01 00:00:00** al **9999-12-31 23:59:59**), y la parte del horario, de **00:00:00** a **23:59:59**.

Este tipo de cambio permite almacenar horas, minutos y segundos, en el formato **HH:MM:SS**, y su rango permitido va desde **-839:59:59** hasta **839:59:59** (unos 35 días hacia atrás y hacia adelante de la fecha actual). Esto lo hace

ideal para calcular tiempos trancurridos entre dos momentos cercanos.

Un campo que tenga definido el tipo de dato **TIMESTAMP** sirve para almacenar una fecha y un horario, de manera similar a DATETIME, pero su formato y rango de valores serán diferentes.

El fomato de un campo TIMESTAMP puede variar entre tres opciones:

- AAAA-MM-DD HH:MM:SS
- AAAA-MM-DD
- AA-MM-DD

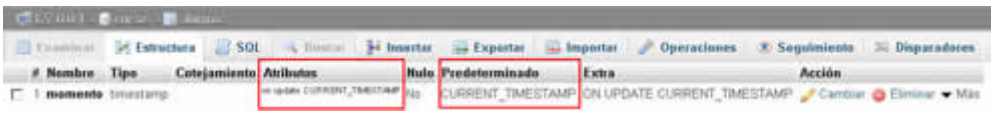
Es decir, la longitud posible puede ser de **14**, **8** o **6** dígitos, según qué información proporcionemos.

El rango de fechas que maneja este campo va desde el **1970-01-01** hasta el año **2037**.

Además, posee la particularidad de que podemos definir que su valor **se mantenga actualizado** automáticamente,cada vez que se **inserte** o que se **actualice** un registro.

De esa manera, conservaremos siempre en ese campo la fecha y hora de la última actualización de ese dato, que es ideal para llevar el control sin necesidad de programar nada.

Para definir esto desde el phpMyAdmin, deberemos seleccionar en Atributos la opción "**on update**" **CURRENT_TIMESTAMP**, y como valor predeterminado **CURRENT_TIMESTAMP**:



(imagenes/materias/programacion-en-internet-2/campos-actualizacion-automatica.jpg)

Campo cuyo valor se actualizará automáticamente al insertar o modificar un registro

En caso de definir un campo como **YEAR**, podremos almacenar un año, tanto utilizando dos como cuatro dígitos.

En caso de hacerlo en **dos dígitos**, el rango posible se extenderá **desde 70 hasta 99** (del 70 hasta el 99 se entenderá que corresponden al rango de años entre 1970 y 1999, y del 00 al 69 se entenderá que se refiere a los años 2000 a 2069); en caso de proporcionar los **cuatro digitos**, el rango posible se ampliará, yendo **desde 1901 hasta 2155**.

Una posibilidad extra, ajena a MySQL pero relativa a las fechas y horarios, es generar un valor de *timestamp* con la función **time** de PHP (repito, no estamos hablando de MySQL, no nos confundamos a causa de tantos nombres similares).

A ese valor, lo podemos almacenar en un campo **INT de 10 dígitos**.

De esa forma, será muy simple ordenar los valores de ese campo (supongamos que es la fecha de una noticia) y luego podremos mostrar la fecha transformando ese valor de **timestamp** en algo legible mediante funciones de manejo de fecha propias de PHP.

Ya hemos visto los diferentes tipos de datos que es posible utilizar al definir un campo en una tabla, pero estos tipos de datos pueden poseer ciertos modificadores o "**atributos**" que se pueden especificar al crear el campo, y que nos brindan la posibilidad de controlar con mayor exactitud qué se podrá almacenar en ese campo, cómo lo almacenaremos y otros detalles.

Aunque algunos de estos atributos ya los hemos utilizado intuitivamente al pasar en algunos de los ejemplos anteriores, a continuación vamos a analizar más en detalle.

Algunas veces tendremos la necesidad de tener que agregar registros sin que los valores de todos sus campos sean completados, es decir, dejando algunos campos vacíos (al menos provisoriamente).

Por ejemplo, en un sistema de comercio electrónico, podría ser que el precio, o la descripción completa de un producto, o la cantidad de unidades en depósito, o la imagen del producto, no estén disponibles en el momento en que, como programadores, comencemos a trabajar con la base de datos.

Todos esos campos, nos conviene que sean definidos como **NULL** (nulos), para que podamos ir agregando registros con los datos básicos de los productos (su nombre, código, etc.) aunque todavía la gente del área comercial no haya definido el precio, ni el área de *marketing* haya terminado las **descripciones**, ni los diseñadores hayan subido las fotos (es típica esta división de tareas en empresas grandes, y hay que tenerla presente, porque afecta la declaración de campos de nuestras tablas).

Si definimos esos campos que no son imprescindibles de llenar de entrada como NULL (simplemente marcando la **casilla de verificación** a la altura de la columna NULL, en el phpMyAdmin), el campo queda preparado para que, si no es que proporcionado un valor, quede vacío pero igual nos permita completar la inserción de un registro completo.

Por omisión, si no marcamos ninguna casilla, todos los campos son **NOT NULL**, es decir, **es obligatorio ingresar algún valor en cada campo** para poder cargar un nuevo registro en la tabla.

Muchas veces necesitamos agilizar la carga de datos mediante un **valor por defecto** *(default)*.

Por ejemplo, pensemos en un sistema de pedidos, donde, al llegar el pedido a la base de datos, su estado sea "recibido", sin necesidad de que el sistema envíe ningún valor, sólo por agregar el registro, ese registro debería contener en el campo "estado" el valor de "recibido".

Este es un típico caso de valor predeterminado o por *default*.

En phpMyAdmin, podemos especificar que un campo tenga un valor predeterminado de tres maneras posibles:

Escribiendo nosotros a mano el valor (como en el caso de "recibido") en cuyo caso debemos elegir la primera de las opciones de la columna "Predeterminado", la que dice "Como fuera definido:", y debemos escribir el valor en el campo de texto existente justo debajo de ese menú:

(imagenes/materias/programacion-en-internet-2/valor-predeterminado.jpg)

1. Podemos definir que el valor por omisión sea NULL, es decir, que si no se proporciona un valor, quede el valor NULL como valor de ese campo.
2. Y, por último, podemos definir para un campo de tipo TIMESTAMP que se inserte como valor por defecto el valor actual de TIMESTAMP *(Current_Timestamp)*, algo que hemos visto en detalle al referirnos a este tipo de dato.

En los dos casos, debemos dejar vacío el cuadro de texto inferior.

Es importante señalar que no se puede dar un valor predeterminado a un campo de tipo TEXT ni BLOB (y todas sus variantes).

Definir un campo de texto CHAR o VARCHAR como BINARY (binario) sólo afecta en el **ordenamiento** de los datos: en vez de ser indiferentes a mayúsculas y minúsculas, un campo **BINARY** se ordenará teniendo en cuenta esta diferencia, por lo cual, a igualdad de letra, primero aparecerán los datos que contengan esa letra en mayúsculas.

Se define desde el phpMyAdmin eligiendo BINARY en el menú Atributos:

El objetivo de crear un índice es mantener ordenados los registros por aquellos campos que sean frecuentemente utilizados en búsquedas, para así agilizar los tiempos de respuesta.

> *Un índice no es más que una tabla "paralela", que guarda los mismos datos que la tabla original, pero en vez de estar ordenados por orden de inserción o por la clave primaria de la tabla, en el índice se* ***ordenan por el campo*** *que elegimos indexar.*

Por ejemplo, si hubiera un buscador por título de noticia, en la tabla de noticias le asignaríamos un índice al campo "título", y las noticias estarían ordenadas de la "a" a la "z" por su título.

La búsqueda se realizará primero en el índice, encontrando rápidamente el título buscado, ya que están todos ordenados y, como resultado, el índice le devolverá al programa MySQL el identificador (id) del registro en la tabla

original, para que MySQL vaya directamente a buscar ese registro con el resto de los datos, sin perder tiempo.

Todo esto, por supuesto, de manera completamente invisible para nosotros.

Para indicar que queremos crear un índice ordenado por un campo determinado, al campo que se indexará debemos especificarle, dentro del atributo **Índice**, del menú de selección que aparece a su derecha, la opción **Index:**

Como podemos apreciar, dentro del menú **Índice** que aparecen otras opciones: **Primary**, **Unique** y **Fulltext**. Veamos en qué consiste cada una de estas variantes.

Siempre, en toda la tabla, uno de los campos (por convención, el primero, y también por convención usualmente llamado **id** –por "identificador"-), debe ser de definido como **clave primario** o **Primary Key**.

Esto impedirá que se le inserten valores repetidos y que se deje nulo su valor.

Habitualmente, se especifica que el campo elegido para clave primaria sea numérico, de tipo **entero** (en cualquiera de sus variantes, según la cantidad de elementos que se identificarán) y se le asigna otro atributo típico, que es **Auto_Increment**, es decir, que no nos preocupamos por darle valor a ese campo: al agregar un registro, MySQL se ocupa de **incrementar en uno el valor de la clave primaria** del último registro agregado, y se lo asigna al nuevo registro.

Este campo no suele tener ninguna relación con el contenido de la tabla, su objetivo es simplemente **identificar** cada registro de forma única, irrepetible.

Podemos definir un sólo campo como clave primaria, o dos o más campos combinados.

En caso de haber definido dos o más campos para que juntos formen el valor único de una clave primaria, diremos que se trata de una clave primaria "**combinada**" o "**compuesta**".

Si especificamos que el valor de un campo sea **Unique**, estaremos obligando a que su valor no pueda repetirse en más de un registro, pero no por eso el campo se considerará clave primaria de cada registro.

Esto es útil para un campo que guarde, por ejemplo, número de documentos de identidad, la casilla de correo electrónico usada para identificar el acceso de un usuario, un nombre de usuario, o cualquier otro dato que no debamos permitir que se repita.

Los intentos por agregar un nuevo registro que contenga un valor ya existente en ese campo, serán rechazados.

Si en un campo de tipo TEXT creamos un índice de tipo **FULLTEXT**, MySQL examinará el contenido de este campo **palabra por palabra**, almacenando cada palabra en una celda de una matriz, permitiendo hacer búsquedas de palabras contenidas dentro del campo, y no ya una simple búsqueda de coincidencia total del valor del campo, que son mucho más rápidas pero no sirven en el caso de búsqueda dentro de, por ejemplo, el cuerpo de una noticia, donde el usuario desea encontrar noticias que mencionan determinada palabra.

Se ignoran las palabras de menos de cuatro caracteres y palabras comunes como artículos, etc. que se consideran irrelevantes para una búsqueda, así como también se iganoran diferencias entre mayúsculas y miniscúlas.

Además, si la palabra buscada se encuentra en más de 50% de los registros, no devolverá resultados, ya que se la considera irrelevante por "exceso" de aparición (deberemos refinar la búsqueda en este caso).

Con esto, damos por terminada esta revisión bastante exhaustiva.

Ya hemos aprendido a crear una base de datos y una tabla, definiendo con total precisión sus campos valiéndolos de tipos de datos y atributos, por lo tanto, estamos en condiciones de comenzar a programar todo lo necesario para que **nuestras páginas PHP se conecten con una base de datos y puedan enviarle o le soliciten datos (llevando-datos-de-la-base-mysql-a-las-paginas-php.php)**.

Referencia en Formato APA:

Delgado, Hugo. (2017).
Tipos de datos en MySQL para una base de datos SQL.
Recuperado 25 de abril, 2020, de
https://disenowebakus.net/tipos-de-datos-mysql.php