Prof. Gabriel F. Arenas

Guía de Estudio

Diseño y normalización de bases de datos relacionales

Diseñando la Base de Datos.

Para diseñar una base de datos se parte de la recolección de atributos o campos que va a tener, y de la definición de sus tipos de dato. La manera más profesional es realizando el análisis de requisitos con todas las personas que van a hacer uso de los datos. Pero por experiencia ya sabes que esto se hace muy a ojo: piden realizar una aplicación y según los requisitos de la aplicación haces el diseño de la BD.

El primer método está más estandarizado, y suele ser más lento pero a cambio *es improbable que el diseño salga mal*. El segundo es *más rápido* porque directamente se piensa en las tablas y sus datos sobre la marcha. Se utiliza principalmente en la metodología de programación conocida como "programación extrema" y en las demás de la familia "desarrollo ágil de software"; y es *más propenso a fallos de diseño*, proporcionalmente inversos al tiempo que se dedique a su definición y valoración (más tiempo, menos probabilidad de fallos).

Normalización de una base de datos.

La normalización *es un método de análisis de BD para conseguir una BD relacional*, que respete la *integridad referencial*, y que no tenga *redundancia de datos*. Se divide en formas normales, y aunque hay un montón y es toda una ciencia, explicaré por encima las 3 primeras ya que *el nivel 3 es suficiente* para la mayoría de casos.

Hay que destacar que la normalización se puede hacer a nivel completo de la BD, o a nivel de tablas o esquemas. La técnica es la misma: analizar el conjunto de campos y en base a eso designar una clave inicial que identifique a un grupo de datos. Por ejemplo si estamos normalizando todo un esquema de facturación podemos partir de los datos del cliente añadiendo la clave del cliente, y según vayamos normalizando nos saldrán todas las tablas y les iremos dando claves primarias nuevas. Si lo que normalizamos es una tabla, el procedimiento es el mismo y ya irán saliendo otras tablas subordinadas si acaso.

La normalización se adoptó porque el viejo estilo de poner los datos en un solo lugar, como un archivo o una tabla de la base de datos, era ineficiente y conducía a errores de lógica cuando se trataba de manipular los datos.

Por lo tanto, el proceso de **normalización de** bases de datos consiste en aplicar una serie de reglas a las relaciones obtenidas tras el paso del modelo entidad-relación al modelo relacional.

Las bases de datos relacionales se normalizan para:

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

Básicamente, en un proceso de conversión de las relaciones entre las entidades, evitando:

- La redundancia de los datos: repetición de datos en un sistema.
- Anomalías de actualización: inconsistencias de los datos como resultado de datos redundantes y actualizaciones parciales.
- Anomalías de borrado: pérdidas no intencionadas de datos debido a que se han borrado otros datos.

- Anomalías de inserción: imposibilidad de adicionar datos en la base de datos debido a la ausencia de otros datos.

En el modelo relacional es frecuente llamar tabla a una relación, aunque para que una tabla sea considerada como una relación tiene que cumplir con algunas restricciones:

- Cada tabla debe tener su nombre único.
- No puede haber dos filas iguales. No se permiten los duplicados.
- Todos los datos en una columna deben ser del mismo tipo.

Las reglas de Codd

Además de la normalización hay unas reglas, las reglas de Codd, que *ayudan a diseñar una BD relacional perfecta* (desde el punto de vista de Codd, claro) que merece la pena estudiarlas pues *son casi de lógica común* y nos harán la vida más fácil en todos los sentidos. La idea de estas reglas surgió porque *la normalización no era suficiente para que una BD fuera relacional, consistente e independiente*.

Hay ocasiones en las que los diseñadores de las BD confeccionan la BD para satisfacer necesidades lógicas y funcionales de una aplicación, por ejemplo almacenando los datos en un formato que luego la aplicación se encarga de transformar. Esto es bastante típico cuando el diseñador es el programador de la aplicación, y lo hace por comodidad o falta de conocimiento.

La moraleja es que *una BD debe ser independiente de la aplicación*, y si lo piensas bien es mejor así. Según las reglas de Codd *la BD tiene que ser completamente operativa desde su lenguaje de consultas* (típicamente SQL), y *las restricciones en los datos deben ser propiedad de la BD* (no vale controlar la entrada desde la aplicación). Con esto conseguiremos que mediante el SQL no se puedan realizar operaciones que hagan que la aplicación no funcione (introduciendo datos en un formato inesperado para la aplicación, por ejemplo), y entre otras cosas, que si tenemos que realizar informes puntuales o sacar listados los podremos hacer desde un simple cliente y sin tener que parsear nada ni realizar consultas sobre consultas.

Definición de la clave

Antes de proceder a la normalización de la tabla lo primero que debemos de definir es una clave, esta clave deberá contener un valor único para cada registro (no podrán existir dos valores iguales en toda la tabla) y podrá estar formado por un único campo o por un grupo de campos.

Claves primarias: se llama clave primaria a un campo, o a una combinación de campos, que identifica en forma única a cada registro. Por ejemplo, para una tabla de clientes se podría usar, como clave primaria, una de las siguientes opciones:

- La cédula de identidad del cliente.
- La combinación de nombre y apellido. Esto es dudoso, ya que un dos clientes pueden tener el mismo nombre y el mismo apellido.
- Un código de cliente, asignado por la empresa.

Una **clave primaria** es aquella columna (pueden ser también dos columnas o más) que identifica únicamente a esa fila. La clave primaria es un identificador que va a ser único para cada fila. Se acostumbra poner la clave primaria como la primera columna de la tabla pero esto no tiene que ser necesario, si no es más una conveniencia. Muchas veces la clave primaria es autonumérica.

En una tabla puede que tengamos más de una clave, en tal caso se puede escoger una para ser la clave primaria, las demás claves son las **claves candidatas**. Además es la posible clave primaria.

Una **clave foránea** es aquella columna que existiendo como dependiente en una tabla, es a su vez clave primaria en otra tabla.

Una **clave alternativa** es aquella clave candidata que no ha sido seleccionada como clave primaria, pero que también puede identificar de forma única a una fila dentro de una tabla. Ejemplo: Si en una tabla clientes definimos el número de documento (id_cliente) como clave primaria, el número de seguro social de ese cliente podría ser una clave alternativa. En este caso no se usó como clave primaria porque es posible que no se conozca ese dato en todos los clientes.

Una **clave compuesta** es una clave que está compuesta por más de una columna.

Grados de normalización

Existen básicamente tres niveles de normalización: Primera Forma Normal (1NF), Segunda Forma Normal (2NF) y Tercera Forma Normal (3NF). Cada una de estas formas tiene sus propias reglas. Cuando una base de datos se conforma a un nivel, se considera normalizada a esa forma de normalización. No siempre es una buena idea tener una base de datos conformada en el nivel más alto de normalización, puede llevar a un nivel de complejidad que pudiera ser evitado si estuviera en un nivel más bajo de normalización.

En general, las primeras tres formas normales son suficientes para cubrir las necesidades de la mayoría mayoría de las bases de datos. El creador de estas 3 primeras formas normales (o reglas) fue *Edgar F. Codd.*

En la tabla siguiente se describe brevemente en qué consiste cada una de las reglas, y posteriormente se explican con más detalle.

Regla	Descripción
Primera Forma Normal (1FN)	Incluye la eliminación de todos los grupos repetidos.
Segunda Forma Normal (2FN)	Asegura que todas las columnas que no son llave sean completamente dependientes de la llave primaria (PK).
Tercera Forma Normal (3FN)	Elimina cualquier dependencia transitiva. Una dependencia transitiva es aquella en la cual las columnas que no son llave son dependientes de otras columnas que tampoco son llave.

Conceptos usados en la normalización

- **Dependencia Funcional.** Es la relación que existe entre dos atributos. Ejemplo:

Dado un valor de X existe un valor de Y entonces Y es funcionalmente dependiente de Y.

EMPLEADO

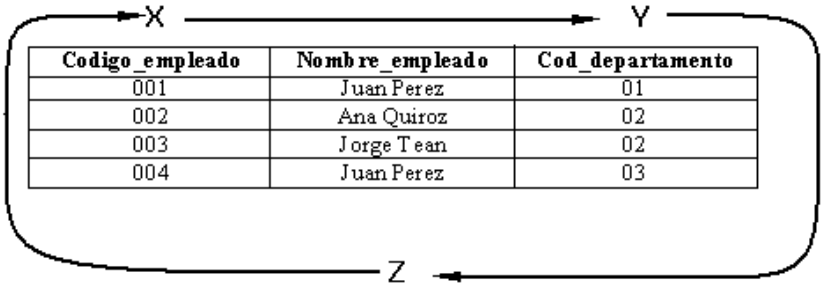
Cod_empleado	Nombre
001	Juan Perez
002	Ana Quiroz
X → Y	

- **Claves o llaves.-** Es el atributo que le da la diferencia a cada tabla este atributo hace que no tengamos tuplas o filas repetidas.

Cod_cliente	Nombre_cliente
001	Juan Perez
002	Ana Quiroz
003	Ana Quiroz
004	Juan Perez
005	José Lopez

- **Dependencia transitoria.-** Es la dependencia que esta encadenada.

X Y Z = Dado un valor de "X" existe un valor de "Y" y dado un valor de "Y" existe un valor de "Z" entonces se dice que "z" es transitivamente dependiente de "X".



Codigo_empleado	Nombre_empleado	Cod_departamento
001	Juan Perez	01
002	Ana Quiroz	02
003	Jorge Tean	02
004	Juan Perez	03

Cod_departamento	Nombre_departamento
01	Juan Perez
02	Ana Quiroz
03	Ana Quiroz

Normalizando la BD: primera forma normal (1FN)

Una tabla está en Primera Forma Normal si:

- Eliminar la repetición de grupos (redundancia de datos)
- Crear una tabla diferente para cada conjunto de datos relacionados.
- Todos los atributos son atómicos. Un atributo es atómico si los elementos del dominio son indivisibles, mínimos.
- La tabla contiene una clave primaria unica.
- La clave primaria no contiene atributos nulos.
- No debe de existir variación en el número de columnas.
- Los Campos no clave deben identificarse por la clave.
- Debe Existir una independencia del orden tanto de las filas como de las columnas, es decir, si los datos cambian de orden no deben cambiar sus significados

Una tabla no puede tener múltiples valores en cada columna. Los datos son atómicos. (Si a cada valor de X le pertenece un valor de Y y viceversa). Esta forma normal elimina los valores repetidos dentro de una BD

- No se permiten vectores de campos en una columna.

Un ejemplo de esto es cuando en un campo de texto metemos varios valores del *mismo dominio*, como por ejemplo tres números de teléfono, o dos direcciones e-mail. Lo típico en estos casos es separar los datos por comas, espacios u otro carácter y después procesarlo mediante la aplicación. Para evitar esto *hay que definir una nueva tabla* que tendrá el identificador de la tabla de la que parte y el campo multivaluado, haciendo juntos de *clave única compuesta* (se puede definir otra incremental si se desea, pero el conjunto de los otros dos campos *tiene que ser único*). Además en esta tabla se puede agregar campos que ayuden a describir el tipo de registro.

Ejemplo

Incorrecto

Clientes

IDCliente	Nombre	Telefono
45	Francisco	444444444
275	Miguel	555555555,666666666

Correcto

Clientes

IDCliente	Nombre
45	Francisco
275	Miguel

Telefonos_cliente

IDCliente	Telefono
45	444444444
275	555555555
275	666666666

- No se permiten grupos repetidos en varias columnas

  Esto es una variante de lo anterior: separamos los campos de un mismo dominio en varias columnas, haciendo un grupo difícilmente procesable a la hora de consultarlo. En el ejemplo anterior sería tener el campo telefono1, telefono2… y así. Es evidente que este fallo del diseño es incluso peor que el anterior pues *habrá muchos campos nulos*, y en caso de necesitar más tendríamos que *redimensionar la tabla* con un nuevo campo (telefono3). Pero la solución es sencilla: la misma que en el anterior caso.

Ejemplo

Incorrecto

Clientes

IDCliente	Nombre	Telefono	Telefono2	Telefono3
45	Francisco	444444444	NULL	NULL
275	Miguel	555555555	666666666	NULL

Correcto

Clientes **telefonos_cliente**

IDCliente	Nombre
45	Francisco
275	Miguel

IDCliente	Telefono
45	444444444
275	555555555
275	666666666

- No se permiten campos nulos
  Esta regla es algo discutible, pero tiene su lógica. Para empezar, si un campo va a tener valores nulos, ¿qué proporción de registros tendrán ese campo con valor nulo? En mi opinión esta regla nos ayuda a separar unas entidades de otras, porque si una cantidad de registros tienen unos atributos que otros no, ¿no será que pertenecen a *otra clase*? Por ejemplo, si en una tabla de productos definimos los campos talla, kilates y potencia se ve que los productos tendrán clases diversas y entonces habrá que crear *una entidad para cada clase* (ropas, joya y eléctricos, por ejemplo) construyendo lo que se llama una *generalización*.

Ejemplo

Incorrecto

productos

IDProducto	Nombre	Talla	Kilates	Potencia
147	Blusa fashion	44	NULL	NULL
155	Broche duquesa	NULL	24	NULL
221	Subwoofer extreme	NULL	NULL	1500

Correcto

Productos **ropas**

IDProducto	Nombre
147	Blusa fashion
155	Broche duquesa
221	Subwoofer extreme
IDProducto	**Talla**
147	44

Joyas **electricos**

IDProducto	Kilates
155	24
IDProducto	**Potencia**
221	1500

Normalizando la BD: segunda forma normal (2FN)

Segunda forma normal se refiere a las relaciones y **dependencias funcionales** entre atributos no-claves.

Una entidad que cumplas Segunda forma normal tiene que tener las siguientes características:

- La entidad debe estar en primera forma normal.
- Que todos los atributos no claves sean dependientes totalmente de la clave primaria

Indicando los dos puntos de una forma diferente, eliminar los campos que son independientes de la clave principal.

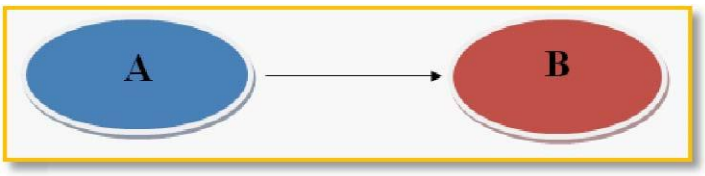
- Crear una nueva tabla para separar la parte parcialmente dependientes de la clave principal y sus dependientes campos.

Dependencia funcional:
Es una conexión entre uno o más atributos, es decir, es una relación entre 2 atributos de una tabla. Por ejemplos si conocemos el valor de **Fecha De Nacimiento** podemos conocer el valor de **Edad**.
Las dependencias funcionales del sistema se escriben utilizando una flecha, de la siguiente manera:

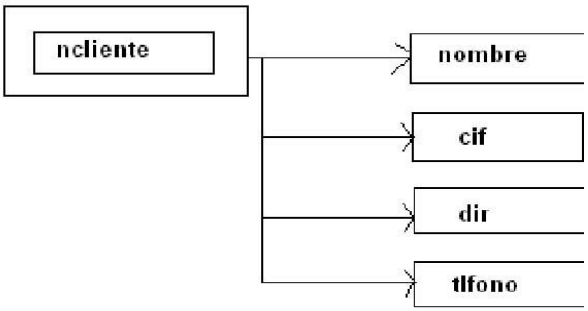
FechaDeNacimiento → **Edad**

Aquí a **FechaDeNacimiento** se le conoce como un determinante. Se puede leer de dos formas **FechaaDeNacimiennto** determina **Edad** o **Edad** es funcionalmente dependiente de

FechaDeNacimiento. De la normalización (lógica) a la implementación (física o real) puede ser sugerible tener éstas dependencias funcionales para lograr la eficiencia en las tablas. Figura 4.



Ejemplo:



Una tabla está en segunda forma normal siempre que esté en primera forma normal y todos sus atributos (campos) *dependan totalmente de la clave candidate sin ser parte de ella*. Viene a ser que, si un campo de la tabla no depende totalmente de una clave única (que pueden ser compuestas), debe sacarse fuera con la parte de la clave principal de la que es dependiente.

Otro ejemplo {DNI, ID_PROYECTO} ⟶HORAS_TRABAJO (con el DNI de un empleado y el ID de un proyecto sabemos cuántas horas de trabajo por semana trabaja un empleado en dicho proyecto) es completamente dependiente dado que ni DNI ⟶HORAS_TRABAJO ni ID_PROYECTO ⟶HORAS_TRABAJO mantienen la dependencia. Sin embargo {DNI, ID_PROYECTO} ⟶NOMBRE_EMPLEADO es parcialmente dependiente dado que DNI ⟶NOMBRE_EMPLEADO mantiene la dependencia.

Ejemplo

Incorrecto

lineas_pedido

IDCliente	IDProducto	Cantidad	Nombre_producto
29	42	1	Zapatillas deportivas de tenis
46	9	5	Balón reglamentario de baloncesto
204	42	1	Zapatillas deportivas de tenis
144	10	1	Zapatillas deportivas de rugby

Correcto

lineas_pedido

IDCliente	IDProducto	Cantidad
29	42	1
46	9	5
204	42	1
144	10	1

productos

IDProducto	Nombre_producto
9	Balón reglamentario de baloncesto
10	Zapatillas deportivas de rugby
42	Zapatillas deportivas de tenis

Como vemos en la tabla "lineas_pedido" del ejemplo incorrecto, la única clave candidata es IDCliente + IDProducto, ya que *en conjunto son únicas en la tabla* (podríamos tener un IDLinea_pedido único también, pero aún así esos dos campos segurían siendo una clave candidata). El campo Cantidad es dependiente de la clave candidata, pues el cliente ha pedido de ese producto una cantidad determinada de artículos, pero el nombre en cambio *es dependiente sólo del producto, no del cliente*. Si dejaramos esa tabla como está, tendríamos por una parte una *redundancia de datos innecesaria* pues el nombre del producto lo podemos sacar uniendo la tabla de productos, y además podrían darse *inconsistencias de datos* si cambiamos el nombre del producto en un registro… ¿cuál sería el nombre real del producto 42 si en varios registros tiene un nombre distinto?

Conclusiones

Por lo tanto los pasos para aplicar la segunda forma normal son muy sencillos: *encontrar las claves candidatas* (compuestas), que identifican de manera única el registro; comprobar que los campos que no forman parte de la clave candidata y no son parte de ella (en el ejemplo de antes ni IDCliente ni IDProducto deben ser analizados) *dependen totalmente de la clave candidata*. Para el segundo paso puede ayudar preguntarse lo siguiente:

¿puedo saber el valor del campo X sabiendo el valor del campo Y (siendo Y parte de la clave candidata y X no siendo parte de ella)? Pero como todo lo relacionado con el análisis esto requiere un mínimo de agudeza, pues puede que casualmente el valor de un campo se repita para una parte de la clave (por casualidad todos los que compran unas pelotas de tenis lo hacen en cantidades de 5) pero sabemos que no es dependiente de ella.

Por último, aclarar que hay ocasiones en las que *el análisis no tiene que ser tan cerrado*, ya que a veces las apariencias engañan. Un ejemplo de ello es una tabla de facturas que tiene el nombre, dirección, NIF, y demás datos del cliente: a simple vista esos datos *están duplicados y dependen del cliente y no de la factura*, pero resulta que esos datos deben permanecer ahí pues fiscalmente debemos saber a qué datos se emitió una factura; esos datos *son realmente dependientes de la factura, no del cliente*. Si no los incluyéramos en la tabla de facturas, al modificar el registro del cliente en la tabla de clientes no sabríamos a qué datos fiscales se emitió la factura. Así que una vez más, hay que utilizar un poco de ingenio y *no aplicar normas como una máquina y sin pensar*.

Normalizando la BD: tercera forma normal (3FN)

Tercera forma normal se refiere a las relaciones y **dependencia funcional transitiva** entre los atributos no-clave.
Para que una entidad esté en tercera forma normal deben cumplirse dos condiciones:

- Que la entidad esté en segunda forma normal.
- Que todos los atributos no claves son independientes del resto de los atributos no clave.

Consiste en eliminar la dependencia transitiva que queda en una segunda forma normal, en pocas palabras una relación esta en tercera forma normal si está en segunda forma normal y no existen dependencias transitivas entre los atributos, nos referimos a dependencias transitivas cuando existe más de una forma de llegar a referencias a un atributo de una relación.

Para definir formalmente la 3FN necesitamos definir **dependencia funcional transitiva.**

Sean *X, Y, Z* tres atributos (o grupos de atributos) de la misma entidad. Si *Y* depende funcionalmente de *X* y *Z* de *Y*, pero *X* no depende funcionalmente de *Y*, se dice entonces que *Z* depende transitivamente de *X*. Simbólicamente sería:

$X \longrightarrow Y \longrightarrow Z$ entonces $X \longrightarrow Z$

FechaDeNacimiento $\longrightarrow$ *Edad*

Edad $\longrightarrow$ *Conducir*

FechaDeNacimiento $\longrightarrow$ *Edad* $\longrightarrow$ *Conducir*

Entonces tenemos que *FechaDeNacimiento* determina a *Edad* y la *Edad* determina a *Conducir*, indirectamente podemos saber a través de *FechaDeNacimiento* a *Conducir* (En muchos países, una persona necesita ser mayor de cierta edad para poder conducir un automóvil, por eso se utiliza este ejemplo).

Ahora bien, para estar más claro con la definición tenemos que una relación está en tercera forma normal si,-……

Una tabla está en tercera forma normal siempre que esté en segunda forma normal (y por consiguiente en primera) y todos sus campos no primarios (campos que no forman parte de una clave candidata) *dependen únicamente de la clave candidata*. Suena como la segunda forma normal, pero es muy distinta: *ningún campo que no sea parte de la clave candidata puede depender de otro campo que no sea la clave candidata*.

Ejemplo

Incorrecto

carga_diaria

IDServidor	Fecha	IDServicio	Nombre_servicio	Carga
21	2009-01-14	1	Oracle	100
21	2009-01-15	9	MySQL	100
21	2009-01-16	22	Apache	85
34	2009-01-14	3	PostgreSQL	74
34	2009-01-15	22	Apache	58
34	2009-01-16	22	Apache	67
66	2009-01-14	9	MySQL	98
66	2009-01-15	22	Apache	94
66	2009-01-16	1	Oracle 10g	84

Correcto

carga_diaria **servicios**

IDServidor	Fecha	IDServicio	Carga
21	2009-01-14	1	100
21	2009-01-15	9	100
21	2009-01-16	22	85
34	2009-01-14	3	74
34	2009-01-15	22	58
34	2009-01-16	22	67

66	2009-01-14	9	98
66	2009-01-15	22	94
66	2009-01-16	1	84

IDServicio	**Nombre_servicio**
1	Oracle
9	MySQL
22	Apache
3	PostgreSQL
22	Apache
22	Apache
9	MySQL
22	Apache
1	Oracle 10g

Imaginad que una tabla se encarga de registrar el primer servicio que más carga los servidores cada día. Del ejemplo incorrecto deducimos que el IDServidor y la Fecha son la clave candidata, pues *identifican de manera única los registros*. Analizando vemos que el IDServicio, que no es un campo primario, depende únicamente de la clave candidata, y que la carga también. Pero resulta que el Nombre_servicio depende de esa clave candidata pero también depende del IDServicio, pues *con el IDServicio podemos averiguar qué Nombre_servicio tiene el registro*. Para solucionar esto sacamos el campo Nombre_servicio de la tabla, y *nos llevamos el IDServicio* para que sea la clave principal pues *es el campo del que depende*

.

Y con este ejemplo vemos qué fácil es librarnos de las inconsistencias de no cumplir la tercera forma normal, y de la redundancia de datos. Si no hubieramos normalizado tendríamos que en un registro el IDServicio 22 es Apache y nadie nos asegura que en otro el IDServicio 22 también lo sea pues puede haberse modificado el campo Nombre_servicio. Y si resulta que la tabla fuese un histórico de 500 servidores durante 1000 días, tendríamos 500 mil registros con *un campo innecesario que estaría duplicado* muchísimas veces.

Diseñando la BD sobre la marcha

Si en vuestro desempeño habitual del trabajo os encontráis con que no podéis aplicar, de una manera formal y detallada, la normalización a la hora de diseñar BD, no os alarméis pues le pasa a mucha gente. Lo que puede ocurrir es que nos quede una BD no relacional, y eso es siempre negativo, pero a base de experiencia iréis adquiriendo *una soltura y capacidad analítica automáticas*.

La normalización, al basarse en reglas lógicas, se puede memorizar muy fácilmente y al final *forma parte del instinto del diseñador*: no necesitaréis bolígrafo y papel para ver que una tabla no está normalizada. De hecho cuando sepáis que datos necesita la aplicación pensaréis directamente en las tablas que saldrán.

Primero, documentarse

Lo más importante para diseñar la BD sobre la marcha es *tener la mente amplia, conocer las bases de la normalización, y dejarse aconsejar por los expertos*. Todo esto de las BD relacionales no es nuevo y hay muchos gurús (Codd, Edgar Frank Codd, es el padre de todos) que os pueden ayudar a entender qué características debe cumplir una BD para ser relacional. De modo que si no sabéis del tema, lo mejor es que os *olvidéis de las malas enseñanazas* que tengáis imbuidas: una BD con muchas tablas *no está mal diseñada* (una con pocas es más probable que sí lo esté); llevarse el código del cliente a todas las tablas (lo necesiten o no) *no es la forma de tener una BD relacional*; guardar los datos serializados en un campo *no te hace la vida más fácil*; tener un campo que hace referencia a una tabla unas veces y a otra en otras ocasiones no es un buen diseño (*un campo referencial sólo puede referenciar a una tabla*, así que usad la generalización en esas situaciones).

Errores habituales

Un error habitual a la hora de usar este método de diseño es tener un campo referencial que admite valores nulos o vacíos (típico clave_referencia 0 cuando no referencia a nada). Si la BD es relacional, *un campo referencial tiene que apuntar a otro registro en la BD*. A veces tenemos un campo IDPadre que hace referencia a un mismo registro de la tabla, o vale 0 cuando el registro es padre en sí… pero *lo correcto en una relación reflexiva* (una tabla relacionada consigo misma) que da lugar a otro tabla que contiene el IDHijo y el IDPadre; consultando esa tabla podemos saber si un registro es hijo (tiene entradas con su ID en IDHijo) o padre (su ID está en algún IDPadre) y sacar todos los hijos de un padre.

Consejos

Otro buen consejo, que no está limitado a este método de diseñar, es *tener en cuenta el tamaño de las tablas* en cuanto a longitud de fila (en bytes). De hecho recuerdo que me hablaron de una regla de diseño de BD que decía que una tabla no debía contener más de X campos… y bueno, siempre es discutible pero es más óptimo que la longitud de la fila no sea muy larga, *sobre todo en las tablas que se consultan con frecuencia*. Es una práctica muy recomendada que si tenemos campos grandes, tipo TEXT o BLOB, saquemos esos datos a otra tabla para que las búsquedas y JOIN generales que no necesiten ese campo sean *más rápidas*. Hay que tener en cuenta que el sistema de gestión de BD (SGBD) en ocasiones no puede optimizar las consultas y necesita *escanear por completo la tabla*, o tiene que volcarla a la memoria física o virtual.

En definitiva: recordad que una BD relacional *no puede ser dependiente de la aplicación*, sino al revés. Así que olvidaros de diseñarla pensando qué es lo mejor para la aplicación, y pensad *qué es lo correcto para que sea más óptima y sencilla de manejar* independientemente del cliente que la maneje (aplicación, consultas directas, herramientas de informe…).