Normalización de Bases de Datos y Técnicas de diseño

Uno de los factores mas importantes en la creación de páginas web dinámicas es el diseño de las Bases de Datos (BD). Si tus tablas no estan correctamente diseñadas, te pueden causar un montón de dolores de cabeza cuando tengas de realizar complicadísimas llamadas SQL en el código PHP para extraer los datos que necesitas. Si conoces como establecer las relaciones entre los datos y la normalización de estos, estarás preparado para comenzar a desarrollar tu aplicación en PHP.

Si trabajas con MySQL o con Oracle, debes conocer los métodos de normalización del diseño de las tablas en tu sistema de BD relacional. Estos métodos pueden ayudarte a hacer tu código PHP mas fácil de comprender, ampliar, y en determinados casos, incluso hacer tu aplicación mas rápida.

Básicamente, las reglas de Normalización están encaminadas a eliminar redundancias e inconsistencias de dependencia en el diseño de las tablas. Más tarde explicaré lo que esto significa mientras vemos los cinco pasos progresivos para normalizar, tienes que tener en cuenta que debes crear una BD funcional y eficiente. Tambien detallaré los tipos de relaciones que tu estructura de datos puede tener.

Digamos que queremos crear una tabla con la información de usuarios, y los datos a guardar son el nombre, la empresa, la dirección de la empresa y algun e-mail, o bien URL si las tienen. En principio comenzarias definiendo la estructura de una tabla como esta:

Formalización CERO

usuarios				
nombre	empresa	direccion_empresa	url1	url2
Joe	ABC	1 Work Lane	abc.com	xyz.com
Jill	XYZ	1 Job Street	abc.com	xyz.com

Diríamos que la anterior tabla esta en nivel de Formalizacion Cero porque ninguna de nuestras reglas de normalización ha sido aplicada. Observa los campos url1 y url2 -- ¿Qué haremos cuando en nuestra aplicación necesitemos una tercera url ? ¿ Quieres tener que añadir otro campo/columna a tu tabla y tener que reprogramar toda la entrada de datos de tu código PHP ? Obviamente no, tu quieres crear un sistema funcional que pueda crecer y adaptarse fácilmente a los nuevos requisitos. Hechemos un vistazo a las reglas del Primer Nivel de Formalización-Normalización, y las aplicaremos a nuestra tabla.

Primer nivel de Formalización/Normalización. (F/N)

1. Eliminar los grupos repetitivos de la tablas individuales.
2. Crear una tabla separada por cada grupo de datos relacionados.
3. Identificar cada grupo de datos relacionados con una clave primaria.

¿ Ves que estamos rompiendo la primera regla cuando repetimos los campos url1 y url2 ? ¿ Y que pasa con la tercera regla, la clave primaria ? La regla tres básicamente significa que tenemos que poner una campo tipo contador autoincrementable para cada registro. De otra forma, ¿ Qué pasaria si tuvieramos dos usuarios llamados Joe y queremos diferenciarlos. Una vez que aplicaramos el primer nivel de F/N nos encontrariamos con la siguiente tabla:

usuarios				
userId	nombre	empresa	direccion_empresa	url
1	Joe	ABC	1 Work Lane	abc.com
1	Joe	ABC	1 Work Lane	xyz.com
2	Jill	XYZ	1 Job Street	abc.com
2	Jill	XYZ	1 Job Street	xyz.com

Ahora diremos que nuestra tabla está en el primer nivel de F/N. Hemos solucionado el problema de la limitación del campo url. Pero sin embargo vemos otros problemas....Cada vez que introducimos un nuevo registro en la **tabla usuarios**, tenemos que duplicar el nombre de la empresa y del usuario. No sólo nuestra BD crecerá muchísimo, sino que será muy facil que la BD se corrompa si escribimos mal alguno de los datos redundantes. Aplicaremos pues el segundo nivel de F/N:

Segundo nivel de F/N

1. Crear tablas separadas para aquellos grupos de datos que se aplican a varios registros.
2. Relacionar estas tablas mediante una clave externa.

Hemos separado el campo url en otra tabla, de forma que podemos añadir más en el futuro si tener que duplicar los demás datos. Tambien vamos a usar nuestra clave primaria para relacionar estos campos:

usuarios			
userId	nombre	empresa	direccion_empresa
1	Joe	ABC	1 Work Lane
2	Jill	XYZ	1 Job Street

urls		
urlId	relUserId	url

1	1	abc.com
2	1	xyz.com
3	2	abc.com
4	2	xyz.com

Vale, hemos creado tablas separadas y la clave primaria en la **tabla usuarios**, userId, esta relacionada ahora con la clave externa en la **tabla urls**, relUserId. Esto esta mejor. ¿ Pero que ocurre cuando queremos añadir otro empleado a la empresa ABC ? ¿ o 200 empleados ? Ahora tenemos el nombre de la empresa y su dirección duplicandose, otra situación que puede inducirnos a introducir errores en nuestros datos. Así que tendrémos que aplicar el tercer nivel de F/N:

Tercer nivel de F/N.

1. Eliminar aquellos campos que no dependan de la clave.

Nuestro nombre de empresa y su dirección no tienen nada que ver con el campo userId, asi que tienen que tener su propio empresaId:

usuarios		
userId	nombre	relEmpresaId
1	Joe	1
2	Jill	2

empresas		
emprId	empresa	direccion_empresa
1	ABC	1 Work Lane
2	XYZ	1 Job Street

urls		
urlId	RelUserId	url
1	1	abc.com
2	1	xyz.com
3	2	abc.com
4	2	xyz.com

Ahora tenemos la clave primaria emprId en la **tabla empresas** relacionada con la clave externa recEmpresaId en la **tabla usuarios**, y podemos añadir 200 usuarios mientras que sólo tenemos que insertar el nombre 'ABC' una vez.

Nuestras tablas de usuarios y urls pueden crecer todo lo que quieran sin duplicación ni corrupción de datos. La mayoria de los desarrolladores dicen que el tercer nivel de F/N es suficiente, que nuestro esquema de datos puede manejar facilmente los datos obtenidos de una cualquier empresa en su totalidad, y en la mayoria de los casos esto será cierto.

Pero hechemos un vistazo a nuestro campo urls - ¿ Ves duplicación de datos ? Esto es perfectamente aceptable si la entrada de datos de este campo es solicitada al usuario en nuestra apliación para que teclee libremente su url, y por lo tanto es sólo una coincidencia que Joe y Jill teclearon la misma url. ¿ Pero que pasa si en lugar de entrada libre de texto usáramos un menú desplegable con 20 o incluso más urls predefinidas ? Entonces tendríamos que llevar nuestro diseño de BD al siguiente nivel de F/N, el cuarto, muchos desarrolladores lo pasan por alto porque depende mucho de un tipo muy específico de relación, la relación 'varios-con-varios', la cual aún no hemos encontrado en nuestra aplicación.

Relaciones entre los Datos

Antes de definir el cuarto nivel de F/N, veremos tres tipos de relaciones entre los datos: uno-a-uno, uno-con-varios y varios-con-varios. Mira la **tabla usuarios** en el Primer Nivel de F/N del ejemplo de arriba. Por un momento imaginámos que ponemos el campo url en una tabla separada, y cada vez que introducimos un registro en la **tabla usuarios** tambien introducimos una sola fila en la **tabla urls**. Entonces tendríamos una relacion uno-a-uno: cada fila en la tabla usuarios tendría exactamente una fila correspondiente en la **tabla urls**. Para los propósitos de nuestra aplicación no sería útil la normalización.

Ahora mira las tablas en el ejemplo del Segundo Nivel de F/N. Nuestras tablas permiten a un sólo usuario tener asociadas varias urls. Esta es una relación uno-con-varios, el tipo de relación más común, y hasta que se nos presentó el dilema del Tercer Nivel de F/N. la única clase de relación que necesitamos.

La relación varios-con-varios, sin embargo, es ligeramente más compleja. Observa en nuestro ejemplo del Tercer Nivel de F/N que tenemos a un usuario relacionado con varias urls. Como dijímos, vamos a cambiar la estructura para permitir que varios usuarios esten relacionados con varias urls y así tendremos una relación varios-con-varios. Veamos como quedarían nuestras tablas antes de seguir con este planteamiento:

usuarios		
userId	nombre	relEmpresaId
1	Joe	1
2	Jill	2

empresas		
emprId	empresa	direccion_empresa
1	ABC	1 Work Lane

2	XYZ	1 Job Street

urls	
urlId	url
1	abc.com
2	xyz.com

url_relations		
relationId	relatedUrlId	relatedUserId
1	1	1
2	1	2
3	2	1
4	2	2

Para disminuir la duplicación de los datos (este proceso nos llevará al Cuarto Nivel de F/N), hemos creado una tabla que sólo tiene claves externas y primarias url_relations. Hemos sido capaces de remover la entradas duplicadas en la tabla urls creando la tabla url_relations. Ahora podemos expresar fielmente la relación que ambos Joe and Jill tienen entre cada uno de ellos, y entre ambos, las urls. Así que veamos exáctamente que es lo que el Cuarto Nivel de F/N. supone:

Cuarto Nivel de F/N.

1. En las relaciones varios-con-varios, entidades independientes no pueden ser almacenadas en la misma tabla.

Ya que sólo se aplica a las relaciones varios-con-varios, la mayoria de los desarrolladores pueden ignorar esta regla de forma correcta. Pero es muy útil en ciertas situaciones, tal como esta. Hemos optimizado nuestra **tabla urls** eliminado duplicados y hemos puesto las relaciones en su propia tabla.

Os voy a poner un ejemplo prático, ahora podemos seleccionar todas las urls de Joe realizando la siguiente instrucción SQL:

SELECT nombre, url FROM usuarios, urls, url_relations WHERE url_relations.relatedUserId = 1 AND usuarios.userId = 1 AND urls.urlId = url_relations.relatedUrlId

Y si queremos recorrer todas las urls de cada uno de los usuarios, hariamos algo así:

SELECT nombre, url FROM usuarios, urls, url_relations WHERE usuarios.userId = url_relations.relatedUserId AND urls.urlId = url_relations.relatedUrlId

Quinto Nivel de F/N.

Existe otro nivel de normalización que se aplica a veces, pero es de hecho algo esotérico y en la mayoria de los casos no es necesario para obtener la mejor funcionalidad de nuestra estructura de datos o aplicación. Su principio sugiere:

1. La tabla original debe ser reconstruida desde las tablas resultantes en las cuales a sido troceada.

Los beneficios de aplicar esta regla aseguran que no has creado ninguna columna extraña en tus tablas y que la estructura de las tablas que has creado sea del tamaño justo que tiene que ser. Es una buena práctica aplicar este regla, pero a no ser que estes tratando con una extensa estructura de datos probablemente no la necesitarás.