Database Normalization

We want to create a table of user information, and we want to store each users' Name, Company, Company Address, and some personal urls. You might start by defining a table structure like this:

Zero Form

Users				
name	company	company_address	url1	url2
Joe	ABC	1 Work Lane	abc.com	xyz.com
Jill	XYZ	1 Job Street	abc.com	xyz.com

This table is in Zero Form because none of rules of normalization have been applied yet. Notice the url1 and url2 fields -- what do we do when the application needs to ask for a third url? Do you want to keep adding columns to your table and hard-coding that form input field into your code? Obviously not, you would want to create a functional system that could grow with new development requirements. Let's look at the rules for the First Normal Form, and then apply them to this table.

First Normal Form

1. Eliminate repeating groups in individual tables.
2. Create a separate table for each set of related data.
3. Identify each set of related data with a primary key.

Notice how we're breaking that first rule by repeating the url1 and url2 fields? And what about Rule Three, primary keys? Rule Three basically means we want to put some form of unique, auto-incrementing integer value into every one of our records. Otherwise, what would happen if we had two users named Joe and we wanted to tell them apart? When we apply the rules of the First Normal Form we come up with the following table:

Users				
userId	name	company	company_address	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

Now the table is said to be in the First Normal Form. We've solved the problem of url field limitation, but look to the problem we created. Every time we input a new record into the **users table**, we've got to duplicate all that company and user name data. Not only will our database grow much larger than we'd ever want it to, but we could easily begin corrupting our data by misspelling some of that redundant information. Let's apply the rules of Second Normal Form:

Second Normal Form

1. Create separate tables for sets of values that apply to multiple records.
2. Relate these tables with a foreign key.

We break the url values into a separate table so we can add more in the future without having to duplicate data. We'll also want to use our primary key value to relate these fields:

users			
userId	name	company	company_address
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

Ok, we've created separate tables and the primary key in the **users table**, userId, is now related to the foreign key in the **urls table**, relUserId. We're in much better shape. But what happens when we want to add another employee of company ABC? Or 200 employees? Now we've got company names and addresses duplicating themselves all over the place, a situation just rife for introducing errors into our data. So we'll want to look at applying the Third Normal Form:

Third Normal Form

1. Eliminate fields that do not depend on the key.

   Our Company Name and Address have nothing to do with the User Id, so they should have their own Company Id:

Users		
userId	name	relCompId
1	Joe	1
2	Jill	2

companies		
compId	company	company_address
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

Now we've got the primary key compId in the **companies table** related to the foreign key in the **users table** called relCompId, and we can add 200 users while still only inserting the name "ABC" once. Our users and urls tables can grow as large as they want without unnecessary duplication or corruption of data. Most developers will say the Third Normal Form is far enough, and our data schema could easily handle the load of an entire enterprise, and in most cases they would be correct.