Normalization

One of the most important factors in software development is database definition. If your tables are not set up properly, it can cause you a lot of headaches down the road when you extract the data you want. By understanding data relationships and the normalization of data, you will be better prepared to begin developing your applications.

Whether you work with MS-Access, Foxpro, MS-SQL Server, mySQL or Oracle, you should know the methods of normalizing the table schema in your relational database system. They can help make your code easier to understand, easier to expand upon, and in some cases, actually speed up your application.

Normalization...

Basically, the Rules of Normalization are enforced by eliminating redundancy and inconsistent dependency in your table designs. Here we will explain what that means by examining the five progressive steps to normalization you should be aware of in order to create a functional and efficient database.

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

Normalization...

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

Normalization...

We would say this table is in Zero Form because none of our rules of normalization have been applied yet. Notice the url1 and url2 fields -- what do we do when our 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 application 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.

Normalization...

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:

Normalization...

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

Normalization...

Now our table is said to be in the First Normal Form. We've solved the problem of url field limitation, but look at the headache we've now caused ourselves. 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:

Normalization...

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:

Normalization...

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

Normalization...

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:

Normalization...

Third Normal Form

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

Normalization...

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

Normalization...

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.

Normalization...

But look at our url fields - do you notice the duplication of data? This is perfectly acceptable if we are not pre-defining these fields. If the input page which our users are filling out to input this data allows a free-form text input there's nothing we can do about this, and it's just a coincidence that Joe and Jill both input the same bookmarks. But what if it's a drop-down menu which we know only allows those two urls, or maybe 20 or even more. We can take our database schema to the next level, the Fourth Form, one which many developers overlook because it depends on a very specific type of relationship, the many-to-many relationship, which we have not yet encountered in our application.

Normalization...

Data Relationships

Before we define the Fourth Normal Form, let's look at the three basic data relationships: one-to-one, one-to-many, and many-to-many. Look at the users table in the First Normal Form example above. For a moment let's imagine we put the url fields in a separate table, and every time we input one record into the users table we would input one row into the urls table. We would then have a one-to-one relationship: each row in the users table would have exactly one corresponding row in the urls table. For the purposes of our application this would neither be useful nor normalized.

Normalization...

Now look at the tables in the Second Normal Form example. Our tables allow one user to have many urls associated with his user record. This is a one-to-many relationship, the most common type, and until we reached the dilemma presented in the Third Normal Form, the only kind we needed.

The many-to-many relationship, however, is slightly more complex. Notice in our Third Normal Form example we have one user related to many urls. As mentioned, we want to change that structure to allow many users to be related to many urls, and thus we want a many-to-many relationship. Let's take a look at what that would do to our table structure before we discuss it:

Normalization…

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

Normalization...

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

Normalization...

In order to decrease the duplication of data (and in the process bring ourselves to the Fourth Form of Normalization), we've created a table full of nothing but primary and foriegn keysin url_relations. We've been able to remove the duplicate entries in the urls table by creating the url_relations table. We can now accurately express the relationship that both Joe and Jill are related to each one of , and both of, the urls. So let's see exactly what the Fourth Form Of Normalization entails:

Normalization...

Fourth Normal Form

1. In a many-to-many relationship, independent entities can not be stored in the same table.

Since it only applies to the many-to-many relationship, most developers can rightfully ignore this rule. But it does come in handy in certain situations, such as this one. We've successfully streamlined our urls table to remove duplicate entries and moved the relationships into their own table.

Just to give you a practical example, now we can select all of Joe's urls by performing the following SQL call:

Normalization...

```
SELECT name, url FROM users, urls, url_relations WHERE
    url_relations.relatedUserId = 1 AND users.userId = 1 AND urls.urlId
    = url_relations.relatedUrlId
```

And if we wanted to loop through everybody's User and Url information, we'd do something like this:

```
SELECT name, url FROM users, urls, url_relations WHERE
    users.userId = url_relations.relatedUserId AND urls.urlId =
    url_relations.relatedUrlId
```

Normalization...

Fifth Normal Form

There is one more form of normalization which is sometimes applied, but it is indeed very esoteric and is in most cases probably not required to get the most functionality out of your data structure or application. It's tenet suggests:

1. The original table must be reconstructed from the tables into which it has been broken down.

The benefit of applying this rule ensures you have not created any extraneous columns in your tables, and that all of the table structures you have created are only as large as they need to be. It's good practice to apply this rule, but unless you're dealing with a very large data schema you probably won't need it.