<http://www.dreamincode.net/forums/topic/179103-relational-database-design-normalization/>

It's common for people new to relational databases to approach the design of the tables like they would a flat, Excel-like spreadsheet: to design the tables as if they were isolated, with all the data needed for a particular functionality crammed into a single table.  
  
This is not how relational databases are meant to be used. Tables in a relational database are linked together, forming a web of sorts, where all the data needed for a single query can be spread over a great many tables, and a change in one table can have an affect on every other table in that database.  
  
It is therefore extremely important that each an every table in the database be properly designed, not only in regards to the data that each table is mean to store, but also in respect to every other table in the "web". If you ever find yourself in need to expand your database, or refactor it in order to fix or improve your existing applications, a proper design can save you an enormous amount of time and effort.  
  
**Key concepts**  
Let me start by naming a few of the most important concepts in relational database design. A basic understanding of these will be required to understand the rest of the article.

* **Primary Key (PK)**  
  A column with a unique value for each row. Although not all database management systems (DBMS) require you to put a PK into each table, from a design perspective a PK is a requirement. No table should be without one.
* **Foreign Key (FK)**  
  These define relationships between tables. When you want a row in one table to be linked to a row in another table, you place a FK column in the child table and use the value of the parent row's PK as the value of the FK field.
* **Composite Key**  
  This is a key that is made up of more than one column. This is typically used when you want to prevent a table from using the same combination of values twice. For example, in a table that lists item prizes for shops, you would only want each shop to have a single price for each item. So, you create a FK for the shop and a FK for the item, and then you create a composite PK out of those two columns. This would cause the DBMS to forcefully restrict entries that would create rows where the combined values of these fields are duplicated. - This type of key is commonly used in N:M relationships. (Explained below... with visual aids.)
* **One-To-One (1:1) relationship**  
  A relationship between two tables, where a single row in one table is linked to a single row in another table.

|  |  |
| --- | --- |
| 1 | +------------+     +----------------+ |
| 2 | | person     |     | person\_contact | |

|  |  |
| --- | --- |
| 3 | +------------+     +----------------+ |
| 4 | | person\_id  |1---1| person\_id      | |

|  |  |
| --- | --- |
| 5 | | first\_name |     | email          | |
| 6 | | last\_name  |     | phone          | |

|  |  |
| --- | --- |
| 7 | +------------+     +----------------+ |

* This type of relationship is practically non-existent in normalized relational designs. They exist mostly to get around limitations in databases like Access, where the number of column was limited, thus creating the need to split tables up. They are also sometimes used to optimize the performance of the database.
* **One-To-Many (1:N) relationship**   
  A relationship between two tables, where multiple rows in a child table can be linked to a single row in a parent table. For example:

|  |  |
| --- | --- |
| 1 | +------------+     +------------+ |
| 2 | | person     |     | country    | |

|  |  |
| --- | --- |
| 3 | +------------+     +------------+ |
| 4 | | person\_id  |  |-1| country\_id | |

|  |  |
| --- | --- |
| 5 | | name       |  |  | name       | |
| 6 | | country\_id |\*-|  +------------+ |

|  |  |
| --- | --- |
| 7 | +------------+ |

* This is in fact the only "real" type of relationship in a relational database. (See the next point for the reasoning behind that assertion.)
* **Many-To-Many (N:M) relationship**  
  A relationship between two tables, where multiple rows in one table can be linked to multiple rows in another table. This type is "artificial" in a a way, because this kind of relationship can not be created directly between tables. To accomplish this type of relationship you need to create a third table; an intermediary table that contains FKs to both parents, linked via a set of 1:N relationships.

|  |  |
| --- | --- |
| 1 | +-----------+     +--------------+     +--------------+ |
| 2 | | shop      |     | prices       |     | product      | |

|  |  |
| --- | --- |
| 3 | +-----------+     +--------------+     +--------------+ |
| 4 | | shop\_id   |1-|  | product\_id   |\*---1| product\_id   | |

|  |  |
| --- | --- |
| 5 | | shop\_name |  |-\*| shop\_id      |     | product\_name | |
| 6 | +-----------+     | price        |     +--------------+ |

|  |  |
| --- | --- |
| 7 | +--------------+ |

**Normalization**  
To help us properly design our tables we have a set of guidelines which, if followed properly, will help reduce the redundancy and chance of data corruption. We call this "Normalization".  
  
There are several steps involved in normalizing a database. The steps are referred to as "Normal Forms" (abbreviated: NF). There are at least seven NF ranging from 1NF to 6NF. Each NF requires that the NF before it has also been satisfied. The spot between 3NF and 4NF is reserved for the BCNF (Boyce-Codd normal form), which was developed later as a slightly stronger version of the 3NF, to address certain shortcomings.  
  
Tables that have reached the 3NF are generally considered "normalized". Specifically aiming for a higher level is unusual, but a table that is designed to be in 3NF is very likely also in the 5NF.  
  
The first three Normal Forms aren't complex at all. They can take some getting used to, but after working with them for a while you may well find yourself developing 3NF compliant tables without even thinking about it.  
  
  
**The First Normal Form (1NF)**  
The first normal form is both the simplest and the most important of the three steps. It simply requires that **tables must not contain repeating groups of data.** This means that if you need to store multiple, identical pieces of data for a single entry (row) then you can not serialize them into a single field, or create multiple identical columns.  
  
Consider the following example. It's a simple list of persons, where each person is listed with his/her name and phone numbers.

|  |  |  |
| --- | --- | --- |
| 1 | +----+------+--------------------+ | |
| 2 | | id | name | phone              | |

|  |  |
| --- | --- |
| 3 | +----+------+--------------------+ |
| 4 | |  1 | Joe  | 588-5522,789-85522 | |

|  |  |
| --- | --- |
| 5 | |  2 | Anna | 589-4567,987-12354 | |
| 6 | +----+------+--------------------+ |

Note that both entries store multiple phone numbers in a single field, separated by a comma. There are two major problems with this approach:

* Your database management system (DBMS) regards each field as a single value, so it can not differentiate between the individual phone numbers. From your DBMS perspective, the phone field is just a normal string containing a single value. - This complicates the use of that data, especially when you need to do things like search for or extract a single value, or filter a data-set based on a single value in the field.
* It also means you need to manually manage the data; to write the code that separates the values when they are retrieved and constructs the string when they are inserted. This makes it FAR more likely that your data will become corrupt or incompatible between applications. The DBMS has NO control over how the data is stored. It simply considers it a single string, and stores it as such. The internal structure and integrity of the data is completely up to you.

To remedy the situation, you would need to separate the data into individual fields. Your first instinct might be to simply create multiple "phone" columns, like so:

|  |  |  |
| --- | --- | --- |
| 1 | +----+------+----------+-----------+ | |
| 2 | | id | name | phone1   | phone2    | |

|  |  |
| --- | --- |
| 3 | +----+------+----------+-----------+ |
| 4 | |  1 | Joe  | 588-5522 | 789-85522 | |

|  |  |
| --- | --- |
| 5 | |  2 | Anna | 589-4567 | 987-12354 | |
| 6 | +----+------+----------+-----------+ |

But this is NOT an acceptable solution. It does solve both of the problems I listed above, but it creates a new problem.  
  
Namely that now we have restricted each person to two phone numbers and ONLY two phone numbers. What if a person needs to store three numbers? Using this table, the only way would be to add a third "phone" column, which would also add a third number to ALL other persons in the table. (What if we need to store a hundred phone numbers? Or a thousand?)  
  
What we want to do is allow each person to supply as many phones as that person needs. To allow for that, we need to extract the phone numbers from that table altogether and put it into a new table dedicated to listing phone numbers. The design for that may look something like this:

|  |  |
| --- | --- |
| 1 | +-----------+     +--------------+ |
| 2 | | person    |     | phone\_number | |

|  |  |
| --- | --- |
| 3 | +-----------+     +--------------+ |
| 4 | | person\_id |1-|  | phone\_id     | |

|  |  |
| --- | --- |
| 5 | | name      |  |-\*| person\_id    | |
| 6 | +-----------+     | number       | |

|  |  |
| --- | --- |
| 7 | +--------------+ |

There each row in the phone\_number table contains a column with the ID of a person. This column identifies the person who's number this is. A relationship like this is referred to as a One-To-Many (1:N) relationship, because each row in the parent table (person) can be linked to multiple rows in the child table (phone\_number), but not the other way around.  
  
The person\_id column in the phone\_number table is what we call a Foreign Key. It is an indication that the value in the column is meant to reference the value of another table. In many cases the DBMS will enforce this link and reject phone\_number entries that do not provide a person\_id that exists in the person table.  
  
  
**The Second Normal Form (2NF)**  
This requires that **no field should only be partially dependent on any candidate key in the table**. This does not only include the PK, but any fields combinations that would uniquely identify a row.  
  
Consider this design:

|  |  |
| --- | --- |
| 01 | +---------------+ |
| 02 | | prices        | |

|  |  |
| --- | --- |
| 03 | +---------------+ |
| 04 | | price\_id (PK) | |

|  |  |
| --- | --- |
| 05 | | product       | |
| 06 | | shop          | |

|  |  |
| --- | --- |
| 07 | | unit\_price    | |
| 08 | | qty           | |

|  |  |
| --- | --- |
| 09 | | shop\_address  | |
| 10 | | unit\_weight   | |

|  |  |
| --- | --- |
| 11 | +---------------+ |

The *price\_id* column is the PK there, but because the combined values of the *product* and *shop* columns could also act as a composite PK, together they are considered a "candidate key".  
  
Lets look at a few example entries into that table.

|  |  |
| --- | --- |
| 1 | +----------+---------+--------+---------+------+--------------+------------+ |
| 2 | | price\_id | product | shop   | u\_price | qty  | shop\_address |unit\_weight | |

|  |  |
| --- | --- |
| 3 | +----------+---------+--------+---------+------+--------------+------------+ |
| 4 | |        1 | Beer    | Bob's  |    9.50 | 12.0 | Main Road 5  |      15.00 | |

|  |  |
| --- | --- |
| 5 | |        2 | Pepper  | Bob's  |   19.50 |  2.5 | Side Road 10 |       2.00 | |
| 6 | |        3 | Beer    | Jill's |    3.50 |  6.0 | Main Steet 1 |       1.50 | |

|  |  |
| --- | --- |
| 7 | |        4 | Pepper  | Jill's |    8.50 | 30.0 | Main Road 1  |      20.00 | |
| 8 | |        5 | Salt    | Jill's |   27.50 | 3.14 | Main Road 10 |     250.00 | |

|  |  |
| --- | --- |
| 9 | +---------+---------+--------+---------+------+--------------+-------------+ |

The problem becomes apparent when we examine the values of the *shop\_address* and *unit\_weight* fields against the above mentioned candidate key. The *shop\_address* values should be identical in all rows with the same *shop* value, and the*unit\_weight* should be identical for the same *product* values. However, this design allows us to specify different shop addresses and unit weights for the same shops and products, which in reality makes no sense.  
  
So, to turn this into a 2NF design, you move these fields out of the table into their own tables, creating FKs to link the new tables to the main *sale* table.

|  |  |
| --- | --- |
| 1 | +-----------+     +--------------+     +--------------+ |
| 2 | | shop      |     | prices       |     | product      | |

|  |  |
| --- | --- |
| 3 | +-----------+     +--------------+     +--------------+ |
| 4 | | name (PK) |1-|  | price\_id     |  |-1| name (PK)    | |

|  |  |
| --- | --- |
| 5 | | address   |  |  | product      |\*-|  | weight       | |
| 6 | +-----------+  |-\*| shop         |     +--------------+ |

|  |  |
| --- | --- |
| 7 | | unit\_price   | |
| 8 | | qty          | |

|  |  |
| --- | --- |
| 9 | +--------------+ |

This ensures that all *prices* entries will be linked to a proper*address* and *weight* values.  
  
**The Third Normal Form (3NF)**  
The last of the forms needed for a database to be considered normalized. It requires that **columns should depend only upon the primary key of the table.** Basically what that means is that any column that is not solely dependent on the primary key of this table, or only partially, should be moved out of the table.  
  
Lets look at two examples:  
  
**First**, if we add a "country" column to the "persons" table we created for the 1NF.

|  |  |
| --- | --- |
| 1 | +-----------+------+----------+ |
| 2 | | person\_id | name | country  | |

|  |  |
| --- | --- |
| 3 | +-----------+------+----------+ |
| 4 | |         1 | Joe  | France   | |

|  |  |
| --- | --- |
| 5 | |         2 | Anna | England  | |
| 6 | +-----------+------+----------+ |

This looks fine at first glance, but if you look closer, the name of the country does not belong in this table.  
  
Consider what happens if you are asked to list ALL possible countries. (E.g. for a drop-down box on a website.) The problem with that is, because the countries are dependent on a person to exist, the design will only allow us to list countries of people that exist in the database. No country that does not have a representative in the person table can exist in the database.  
  
The solution for this is to move the country out of the table, into it's own table, and add a Foreign Key to the person table that references the country the person belongs to:

|  |  |
| --- | --- |
| 1 | +------------+     +------------+ |
| 2 | | person     |     | country    | |

|  |  |
| --- | --- |
| 3 | +------------+     +------------+ |
| 4 | | person\_id  |  |-1| country\_id | |

|  |  |
| --- | --- |
| 5 | | name       |  |  | name       | |
| 6 | | country\_id |\*-|  +------------+ |

|  |  |
| --- | --- |
| 7 | +------------+ |

Now you can list all the countries in the country table, and just link the persons to their respective countries.  
  
  
**Second**, to address another common 3NF conformance issue, if we wanted to add more than just the country name for each person. If we were to add the city and address info as well, it might looks something like:

|  |  |
| --- | --- |
| 1 | +-----------+------+----------+--------+------------------+ |
| 2 | | person\_id | name | country  | city   | address          | |

|  |  |
| --- | --- |
| 3 | +-----------+------+----------+--------+------------------+ |
| 4 | |         1 | Joe  | France   | Paris  | Eiffel blwd. 101 | |

|  |  |
| --- | --- |
| 5 | |         2 | Anna | England  | London | English road 302 | |
| 6 | +-----------+------+----------+--------+------------------+ |

This is basically the same problem we had with the previous example, but we have added two more columns, neither of which belong in this table either. So, lets try to apply the same solution as last time, but this time move all three location items into a location table, and link the location to the person:

|  |  |
| --- | --- |
| 1 | +-------------+     +-------------+ |
| 2 | | person      |     | location    | |

|  |  |
| --- | --- |
| 3 | +-------------+     +-------------+ |
| 4 | | person\_id   |  |-1| location\_id | |

|  |  |
| --- | --- |
| 5 | | name        |  |  | country     | |
| 6 | | location\_id |\*-|  | city        | |

|  |  |
| --- | --- |
| 7 | +-------------+     | address     | |
| 8 | +-------------+ |

This is better, but can you spot the problem?  
  
As you remember, to reach 3NF no field can depend on anything but the PK of the table. But if you look at the location table, notice that "address" depends not only on the PK, but on the "city" field as well. (If the city field changes, so must the address field). The same applies to the city field; it depends on the country field.  
  
This means that, in order to reach 3NF, you need to move country, city and address all into their own tables, each linked to each other. The address table should be linked to the person, the city table linked to the address table, and finally the country table linked to the city table.

|  |  |
| --- | --- |
| 01 | +-------------+     +------------+ |
| 02 | | person      |     | address    | |

|  |  |
| --- | --- |
| 03 | +-------------+     +------------+ |
| 04 | | person\_id   |  |-1| address\_id | |

|  |  |
| --- | --- |
| 05 | | location\_id |\*-|  | city\_id    |\*-| |
| 06 | | name        |     | name       |  | |

|  |  |
| --- | --- |
| 07 | +-------------+     +------------+  | |
| 08 | | |

|  |  |
| --- | --- |
| 09 | +------------+     +------------+   | |
| 10 | | country    |     | city       |   | |

|  |  |
| --- | --- |
| 11 | +------------+     +------------+   | |
| 12 | | country\_id |1-|  | city\_id    |1--| |

|  |  |
| --- | --- |
| 13 | | name       |  |-\*| country\_id | |
| 14 | +------------+     | name       | |

|  |  |
| --- | --- |
| 15 | +------------+ |

Now this design is in 3NF, and could be considered "normalized".