**Design databases**

1. **Identifying Entities**

* Dans le développement d’un projet, la première chose à faire est de savoir les acteurs qui vont participer dans ce projet, et plus précisément, les entités qui vont être actif.

**Exemple :**

On va définir les acteurs pour un projet de site web d’un magasin ; on doit recueillir les informations avec laquelle on va travailler.

* Dans un magasin, on vend des produits aux clients
* Le magasin est une localisation
* Vendre est un événement
* Produit sont des objets
* Les clients sont des humains

Toutes ces entités formeront le besoin qui va être inclus dans la base de données.

Mais quels sont les autres informations qui se produisent lors de la vente d’un produit ? Tout ceci doit être englobé afin de compléter notre conception.



Figure : Les entités du système

1. **Identifying Relationships**

* La prochaine étape maintenant est de déterminer la relation entre les différentes entités acteur de ce système. la relation est qu’est ce qu’une entité fait avec une autre.

🡺 Donc faut écrire les scénarios qui se passent entre les différentes entités entre eux.  
🡺 Les cardinalités sont combien dans un coté de la relation apporte a l’autre coté de la relation.

**Exemple :**

Combien de clients dépendent d’une seule vente ? Combien de ventes dépendent d’un seul client ? Combien de ventes dépendent d’un seul magasin ?

On aura comme ceci :

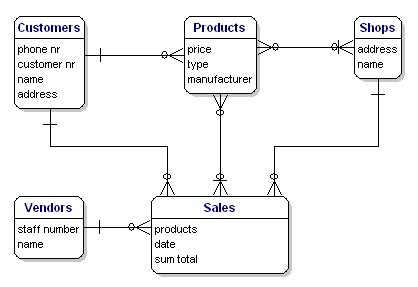
* **Customers --> Sales;** 1 customer can buy something several times
* **Sales --> Customers**; 1 sale is always made by 1 customer at the time
* **Customers --> Products;** 1 customer can buy multiple products
* **Products --> Customers;** 1 product can be purchased by multiple customers
* **Customers --> Shops**; 1 customer can purchase in multiple shops
* **Shops --> Customers**, 1 shop can receive multiple customers
* **Shops --> Products**; in 1 shop there are multiple products
* **Products --> Shops;** 1 product (type) can be sold in multiple shops
* **Shops --> Sales**; in 1 shop multiple sales can me made
* **Sales --> Shops;** 1 sale can only be made in 1 shop at the time
* **Products --> Sales**; 1 product (type) can be purchased in multiple sales
* **Sales --> Products;** 1 sale can exist out of multiple products

**Est ce qu’on a mentionnée toutes les relations ?** Au faite Non ! Car une relation réfère à une règle de gestion, donc tout dépend de ce que l’on souhaite réaliser, on pourra l’insérer dans notre schéma.

**Comment faciliter la schématisation ?** On va mettre les cardinalités par chaque relation  
🡺**Customers --> Sales;** 1 customer can buy something several times  
🡺**Sales --> Customers;** 1 sale is always made by 1 customer at the time

So we’ll get:

* **Customers --> Sales**; --> 1:N
* **Customers --> Products**; --> M:N
* **Customers --> Shops**; --> M:N
* **Sales --> Products**; --> M:N
* **Shops --> Sales**; --> 1:N
* **Shops --> Products**; --> M:N



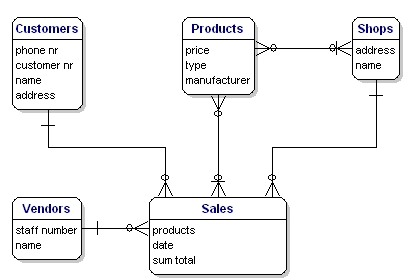
1. **Simplify the diagram**

### Redundant Relationship

Sometimes in our model, we can get a ‘redundant relationship’, There are relationships that are already indicated by other relationship, although not directly.

In the case of our example, we have a direct relationship between customers and products. But there are also relationships from customers to sales and from sales to products, so indirectly we have a relationship between customers and products through sales.

**The new model will then look like:**

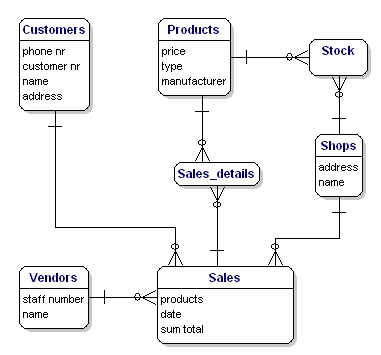


**Solving Many-to-Many relationship**

Many-to-many relationships (M: N) are not directly possible in a database.  Somewhere you need to save which records these are and the solution is to split the relationship up in two one-to-many relationships.

**Products <----> Sales**: Every sale includes more products. The relationship shows the content of the sale. In other words, it gives details about the sale. So the entity is called 'Sales details'. You could also name it 'sold products'.

**Products <----> Shops**: Shows which products are available in which the shops, also known as 'stock'. Our model would now look like this:



1. **Identifying Attributes**

The data elements that you want to save for each entity are called 'attributes'.  
Attributes are all the information that we want to know about an entity.

For example:

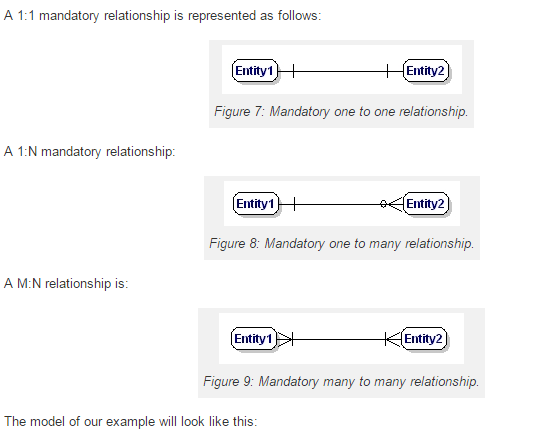
* **About the products that you sell**: you want to know, for example, what the price is, what the name of the manufacturer is, and what the type number is.
* **About the sales**: you know when they happened, in which shop, what products were sold, and the sum total of the sale.

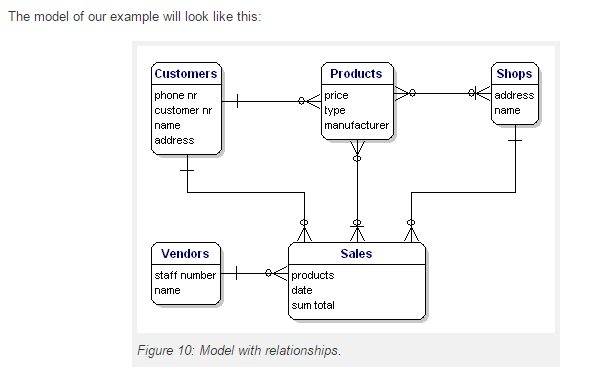
**Derived data:**

Derived data is the data that is derived from another data that we have already saved. In this case of the ‘sum total’ is a classical case of derived data. We know exactly what has been sold and what each product costs. So we can calculate how much the ‘sum total’ is, so it’s not necessary to save the sum total.

So why is it saved here? Well, because it is a sale, and the price of the product can vary over time.   
A product can be priced at 10 Euros today and at 8 Euros next month and for your administration you need to know what it cost at the time of the sale, and the easiest way to do this is to save it here. There are a lot of more elegant ways, but they are too profound for this article.

1. **Presenting Entities and Relationships: Entity Relationship Diagram (ERD)**

****



**Primary key:**

The primary key is one or more data attributes that identify an entity. A key that consist of two or more attributes is called a composite key.

All attribute part of primary key must have a value in every record (which cannot be left empty), and the combination of the values within these attributes must be unique in the table.

For example in **the Sales details** entity we could use the combination of the PK's of the sales and products entities as the PK of Sales details. In this way we enforce that the *same product* (type) can only be used *once in the same sale*. Multiple items of the same product type in a sale must be indicated by *the quantity*.

**Foreign key:**

The foreign key in an entity is the reference to the primary key of another entity. In the ERD that attribute will be assigned with ‘FK’ behind its name.

The foreign key of an entity can also be part of the primary key; in that case the attribute will be indicated with ‘PF’ behind its name. This is usually the case with the link-entities, because you usually link two instances only once together (with 1sale only 1 product type is sold 1 time).

Please note that the attribute 'products' is no longer necessary in 'Sales', because 'sold products' is now included in the link-table

 In the link-table another field was added, 'quantity', that indicates how many products were sold.  
The quantity field was also added in the stock-table, to indicate how many products are still in store.

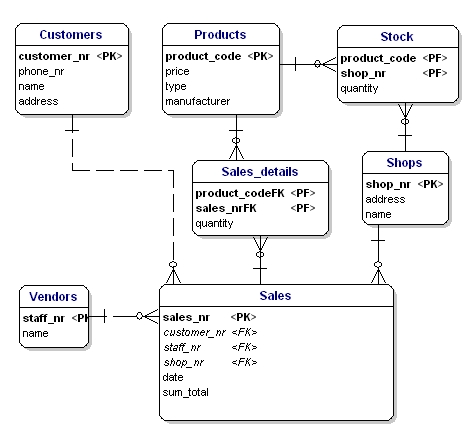


Figure 2 : ERD Diagram - PK and FK.

1. **Defining The Attribute’s Data Type**

There are a lot of different data types. A few are standardized, but many databases have their own data types that all have their own advantages. The standard data types that every database knows, and are most-used, are: CHAR, VARCHAR, TEXT, FLOAT, DOUBLE, and INT.

**Text:**

* **CHAR** (length): includes text (characters, numbers, punctuations...). CHAR has as characteristic that it always saves a fixed amount of positions. If you define a CHAR(10) you can save up to ten positions maximum, but if you only use two positions the database will still save 10 positions. The remaining eight positions will be filled by spaces.
* **VARCHAR** (length): includes text (characters, numbers, punctuation...). VARCHAR is the same as CHAR; the difference is that VARCHAR only takes as much space as necessary.
* **TEXT**: can contain large amounts of text. Depending on the type of database this can add up to gigabytes.

**Numbers:**

* INT - contains a positive or negative whole number. A lot of databases have variations of the INT, such as TINYINT, SMALLINT, MEDIUMINT, BIGINT, INT2, INT4, and INT8. These variations differ from the INT only in the size of the figure that fits into it. A regular INT is 4 bytes (INT4) and fits figures from -2147483647 to +2147483646, or if you define it as UNSIGNED from 0 to 4294967296. The INT8, or BIGINT, can get even bigger in size, from 0 to 18446744073709551616, but takes up to 8 bytes of diskspace, even if there is just a small number in it.
* FLOAT, DOUBLE - The same idea as INT, but can also store floating point numbers. . Do note that this does not always work perfectly. For instance in MySQL calculating with these floating point numbers is not perfect, (1/3)\*3 will result with MySQL's floats in 0.9999999, not 1.

**Other types:**

* BLOB - for binary data such as files.INET - for IP addresses. Also useable for netmasks.

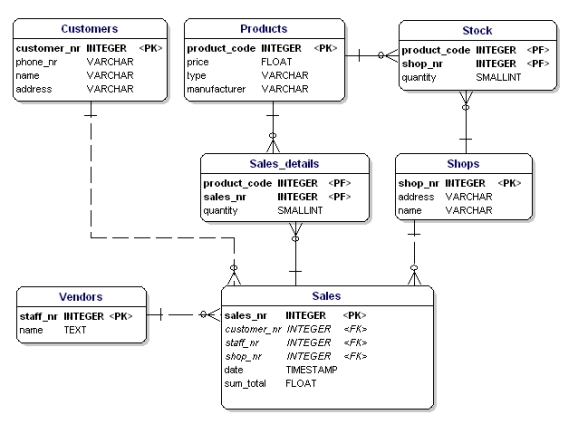


Figure 3 : Data model displaying data types

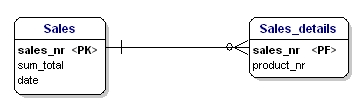
1. **Normalization**

**The first form**

The first form of normalization states that there may be no repeating groups of columns in an entity. We could have created an entity 'sales' with attributes for each of the products that were bought. This would look like this:

****

What is wrong about this is that now only 3 products can be sold. If you would have to sell 4 products, than you would have to start a second sale or adjust your data model by adding 'product4' attributes. Both solutions are unwanted. In these cases you should always create a new entity that you link to the old one via a one-to-many relationship

****

**The second form**

The second form of normalization states that all attributes of an entity should be fully dependent on the whole primary key. This means that each attribute of an entity can only be identified through the whole primary key. Suppose we had the date in the **Sales\_details** entity:

****

Figure 4 : Not according to 2th form

This entity is not according the second normalization form, because in order to be able to look up the date of a sale, I do not have to know what is sold (*productnr*), the only thing I need to know is the sales number. This was solved by splitting up the tables into the sales and the **Sales\_details** table:

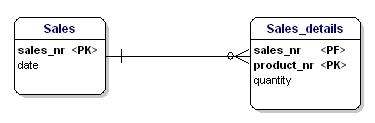
****

Figure 5 : accord to 2th form

Now each attribute of the entities is dependent on the whole PK of the entity. The date is dependent on the sales number, and the quantity is dependent on the sales number and the sold product.

### Normalization, the Third Form

The third form of normalization states that all attributes need to be directly dependent on the primary key, and not on other attributes. This seems to be what the second form of normalization states, but in the second form is actually stated the opposite. In the second form of normalization you point out attributes through the PK, in the third form of normalization every attribute needs to be dependent on the PK, and nothing else.

****

Figure 6 : Not according to 3th form

In this case the price of a loose product is dependent on the ordering number, and the ordering number is dependent on the product number and the sales number. This is not according to the third form of normalization. Again, splitting up the tables solves this.

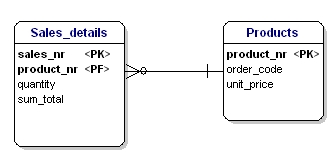
****

Figure 7 : According to 3th form

### Normalization, More Forms

There are more normalization forms than the three forms mentioned above, but those are not of great interest for the average user. These other forms are highly specialized for certain applications. If you stick to the design rules and the normalization mentioned in this article, you will create a design that works great for most applications.

### Normalized Data Model

If you apply the normalization rules, you will find that the 'manufacturer' in de product table should also be a separate table:

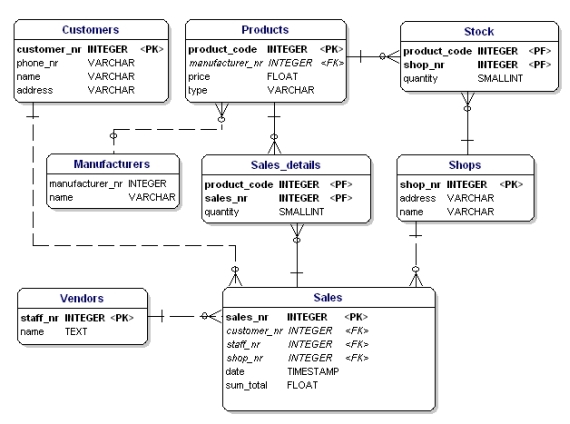


Figure 8 : Data model according to normalization