# 4. Improving the database design



## Database normalisation

You've now seen how to take a data model and represent it in a relational database. To recap:

- each class is represented by a table
- each object becomes a row in the table
- each table has a primary key which is a field or set of fields which uniquely identifies each row
- a relationship between two tables is represented by the foreign key field(s) which refer to the primary key of the related table

However, there may still be problems which can result in data in the database becoming inaccurate or difficult to retrieve. We need to do some further checking of the database design to help prevent such problems.

The process of formally checking and modifying a relational database design is called **normalisation**. Normalisation uses a set of rules to check whether all fields are in the right tables and whether we need to restructure or add tables to the schema. These rules were first proposed by E.F. Codd in about 1970, and have become a key part of relational database design.

As a general rule, a well thought-out data model tends to lead to a pretty well normalised database schema. However, any flaws in the data model design will have been translated in the database schema. Also, there may be problems which arise because of the way we have chosen to represent some features of the data model.

We will look at some of the problems which can arise, and how normalisation can help.

# Problems with un-normalised data

Problems with databases are usually a result of having attributes in the wrong tables. The solution usually involves moving attributes to different tables and creating additional tables.

Look at the following example from an IT consultancy company's database.

> **NOTE**
>
> Of course the GCUTours database schema is the result of a careful design process, so all its attributes *must* be in the right place. Maybe - you can look back later and see if this is true, but for now we'll use other examples.

## Consultants

| consultantID | firstname | lastname |
|---|---|---|
| 1001 | John | Smith |
| 1002 | Amy | Jones |
| 1003 | Jane | Lee |

## Assignments

| consultantID | clientnumber | clientname | contact | hours |
|---|---|---|---|---|
| 1001 | 6 | Acme Ltd | 0141 999 0001 | 20 |
| 1001 | 4 | SuperPrint | 0141 888 4567 | 13 |
| 1002 | 10 | Bestco Stores | 0141 987 6543 | 28 |
| 1002 | 6 | Acme Ltd | 0141 999 0010 | 6 |

There are two tables containing information about consultants and the clients they are assigned do work for. Look at the *Assignments* table. This table has **repeated data** – there are two rows containing the details of the same client (Acme Ltd.). This happens because two consultants have worked for this client. Data which is repeated unnecessarily anywhere in a database is called **redundant** data.

### Inaccurate data

Why is this a problem? One reason is that repeating data increases the chances of **inconsistencies** and **inaccuracies** in the data. Look at the contact number for Acme Ltd – the number is different in the two rows. Which one is correct? There is no way of telling from this data.

It's quite easy to see the problem by looking at the data shown in the figure. However, what if we had only looked at the first three rows? We wouldn't have seen the problem, and might have concluded that the database design was OK. The problem might only have surfaced later on when people actually started using the database.

However, **following the normalisation process** will pick up the problem before the design is signed off and users start complaining.

## Update anomalies

These are problems which arise when you try to add or remove data from a database. Update anomalies can make it impossible to get data into the database, and can cause important data to be lost from the database. Here are two examples of problems with the *Consultants* and *Assignments* tables.

### *Insertion problems*

What should the primary key be for the *Assignments* table? It can't be a single field – you can see from the figure that every field can have duplicated values. The combination of *consultantID* and *clientnumber* will be unique, though, so it should be suitable.

So, what if the company signs up a new client, but hasn't decided which consultant(s) will be assigned to work for them? We could add a new row with the information about the client and simply leave the *consultantID* field left empty. Unfortunately, databases **do not allow rows to be inserted with a null value in a primary key field**. Therefore we have no way of recording information about a client until a consultant is assigned to them.

### *Deletion problems*

What if Amy Jones is no longer going to work for SuperPrint? We remove the relevant row from the *Assignments* table. Whoops! We have just deleted all the information we have in the database about SuperPrint. If we want to assign another consultant to work for SuperPrint then we'll have to re-enter the company details.

Both of these problems can be avoided with a small change to the design. We'll look at the improved version, and then we'll go on to look in detail at how normalisation would deal with this and other situations.

## A better design

All of these problems arise because some of the fields in the Assignments table should really not be in that table. What we need is an **additional table** to store information about clients. The redesigned tables are shown below.

**Consultants**

| consultantID | firstname | lastname |
|---|---|---|
| 1001 | John | Smith |
| 1002 | Amy | Jones |
| 1003 | Jane | Lee |

**Assignments**

| consultantID | clientnumber | hours |
|---|---|---|
| 1001 | 6 | 20 |
| 1001 | 4 | 13 |
| 1002 | 10 | 28 |
| 1002 | 6 | 6 |

**Clients**

| clientnumber | clientname | contact |
|---|---|---|
| 6 | Acme Ltd | 0141 999 0001 |
| 4 | SuperPrint | 0141 888 4567 |
| 10 | Bestco Stores | 0141 987 6543 |

Now:

- each piece on information about the client is recorded only once.

- we can add a new client in the Clients table without assigning a consultant right away
- we can delete an assignment without deleting information about the client.

**THINK ABOUT IT**

What would the relationships between these tables be, and how would these relationships be implemented? What kind of relationship is there between consultants and clients?

In this example, the problem has arisen because the design of the data model was flawed. The purpose of the *Assignment* entity simply wasn't clear enough. Had we identified the need for a *Client* entity when designing the data model, and then the database problems would not have arisen.

# Functional dependencies

Normalisation is based on the idea of a **functional dependency**. The following statement is an example of a functional dependency in the *Consultants* table you have seen above:

> *If we know the value of a consultant's consultantID wecan tell you the value of his or her last name*
>
> We can write this more formally:
>
> *consultantID functionally determines lastname*
>
> or as symbols:
>
> *consultantID* → *lastname*

So, if we know a consultant's ID is 1001, can we say for certain what his last name is? Yes – it's Smith.

Does it work the other way round? If we know a consultant's last name is Jones, can we say for certain what her ID is? It looks like it from the data shown. However, the ID is unique and the last name is not, so we could add another row later for a consultant called Bob Jones, with ID 1006, for example.

Then, given the last name Jones, we can't say for sure what the corresponding ID is as it could be either 1002 or 1006. So, *lastname* does *not* functionally determine *consultantID*.

**THINK ABOUT IT**

Are the following statements true or false:

*firstname* → *lastname* (Consultants table)

*consultantID* → *firstname* (Consultants table)

*clientnumber* → *contact* (Clients table)

*consultantID* → *clientnumber* (Assignments table)

On what is *hours* functionally dependent in the Assignments table?

## Functional dependencies and keys

You saw the term **primary key** earlier in the module. Keys are closely related to functional dependencies as follows:

» *The key fields of a table should functionally determine all the other fields in the table.*

So, as we have seen, *lastname* does not functionally determine *consultantID*, so it **cannot be a key field**.

However, *consultantID* does functionally determine *lastname*, and also *firstname* (we can write this as *consultantID* → *firstname*, *lastname*), so it **is a key**.

## Primary keys

The terms **key** and **primary key** seem to be pretty much the same thing. There is an important difference, though. Remember that a key can contain more than one field, so what about the combination of *consultantID* and *lastname*? If we know that the ID is 1001 and the last name is Smith, can we say for sure what the first name is? Yes we can – we can write this functional dependency as:

(*consultantID, lastname*) → *firstname*

So the combination of these two fields is a key. However, we **don't actually need** *lastname* in order to know *firstname* – the **ID is sufficient** as it is itself a key. Therefore, this isn't a primary key – a primary key must have no unnecessary fields. The rule is:

» *A primary key has no subset of its fields that is also a key.*

This is actually quite important. Remember that we represent relationships using **foreign keys** which must match primary keys. If we defined (*consultantID*, *lastname*) as the primary key of *Consultants*, then there would need to be an additional *lastname* field in *Assignments* to allow a foreign key to be defined, as shown in the figure below. This is an example of redundant data.

### Assignments

| consultantID | lastname | clientnumber | hours |
|---|---|---|---|
| 1001 | Smith | 6 | 20 |
| 1001 | Smith | 4 | 13 |
| 1002 | Jones | 10 | 28 |
| 1002 | Jones | 6 | 6 |

## Normalisation and normal forms

Now that we know all about functional dependencies and primary keys, we are ready to do some normalisation. There are several levels of normalisation, called **normal forms**. We proceed through the forms, refining the tables and addressing additional problems each time.

## First normal form (or 1NF)

First normal form ensures that we are not trying to cram several pieces of data into a single field. A fancy way of saying this is that the data in a table should be **atomic**.

The following example shows a table which is not in 1NF. We are storing information about the skills of our IT consultants. Each row has several pieces of data in the *skills* field.

### Consultants

| consultantID | firstname | lastname | skills |
|---|---|---|---|
| 1001 | John | Smith | databases, Java, UML |
| 1002 | Amy | Jones | networks, web design |
| 1003 | Jane | Lee | Windows, Linux, databases |

**Why is this bad?** One reason is that it is difficult to find all the consultants with a particular skill with a table like this.

What if we make a separate field for each skill, like this?

**Consultants**

| consultantID | firstname | lastname | skill1 | skill2 | skill3 |
|---|---|---|---|---|---|
| 1001 | John | Smith | databases | Java | UML |
| 1002 | Amy | Jones | networks | web design | |
| 1003 | Jane | Lee | Windows | Linux | databases |

Well, each field only contains one piece of information now. However, this is *not* a good solution. What if Jane Lee learns to do web design in addition to her other three skills? We would have to add a new field to the table to accommodate her all-round brilliance. We are still keeping more than one *skill* value in each row, even if the values are actually in separate fields. In fact, to be properly atomic, a table can't have multiple fields with the same kind of data, so this solution is still not atomic.

Here's a general rule for checking for 1NF:

» *A table is not in **first normal form** if it contains data which is not atomic – that is, it keeps multiple values for a piece of information.*

Normalisation gives us rules – it also gives us ways to fix tables which don't obey the rules. The fix for a table not in 1NF is:

» *Remove the multivalued information from the table. Create a new table with that information and the primary key of the original table.*

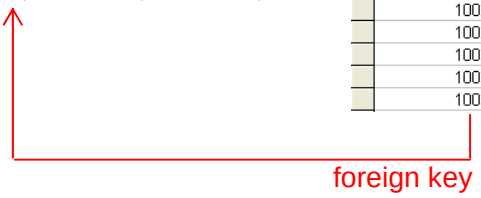
This means we should now have two tables:

**Consultants**

| consultantID | firstname | lastname |
|---|---|---|
| 1001 | John | Smith |
| 1002 | Amy | Jones |
| 1003 | Jane | Lee |

**ConsultantSkills**

| consultantID | skill |
|---|---|
| 1001 | databases |
| 1001 | Java |
| 1001 | UML |
| 1002 | networks |
| 1002 | web design |
| 1003 | Windows |
| 1003 | Linux |
| 1003 | databases |




The foreign key of *ConsultantSkills* will be *consultantID*, and will refer to the primary key of *Consultants*. This is a one-to-many relationship.

# Second Normal Form (2NF)

Getting to first normal form is a good start, but there can still be plenty of problems lurking. To see an example, let's now add some more information about the skills of the consultants. Clients will be charged at different hourly rates for a consultant applying each skill. The *ConsultantSkills* table could gain an extra field:

## ConsultantSkills

| consultantID | skill | hourlyrate |
|---|---|---|
| 1001 | databases | £31.00 |
| 1001 | Java | £35.00 |
| 1001 | UML | £35.00 |
| 1002 | networks | £26.00 |
| 1002 | web design | £26.00 |
| 1003 | Windows | £22.00 |
| 1003 | Linux | £32.00 |
| 1003 | databases | £31.00 |

This table is in 1NF as the data is atomic. Before we can decide if it is in second normal form we need to know what the primary key is.

It can't be *consultantID* as the values are not unique. Neither are the values in *skill*. However, the combination (*consultantID*, *skill*) must be unique (for each consultant, the table stores each skill only once).

This is much like the example we looked at earlier, the *Assignments* table on page 2. As in that example, this table is prone to update anomalies and inaccurate data. We solved the problem for that example – now we'll see how the definition of 2NF leads to a similar solution here.

» *A table is in **second normal form** if it is in first normal form AND we need ALL the fields in the key to determine to values of the non-key fields.*

Why is this table not in 2NF? Well, the value of *hourlyrate* is functionally dependent **only** on the value of *skill*, which is only **part of the primary key**.

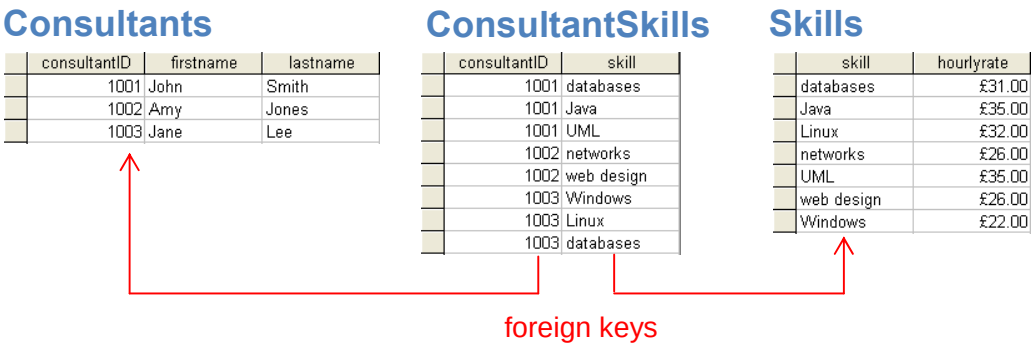
*skill* → *hourlyrate*

*hourlyrate* does not depend on *consultantID* – database work is charged at £31 no matter who is doing the work. So, we don't need all the fields in the key to determine the value of the non-key field, *hourlyrate*.

Again, normalisation gives us a way to fix this:

» *Remove the non-key fields that are not dependent on the whole of the primary key. Create another table with those fields and the part of the primary key they do depend on.*

The result of applying this fix is shown in the next figure. The *Consultants* table is shown here also to illustrate the relationships between the full set of tables.




*ConsultantSkills* will now have another foreign key, *skill*, which will refer to the primary key of *Skills*. This is another one-to-many relationship.

**THINK ABOUT IT**

We could have chosen to use an ID field as the primary key for *Skills*. What fields would *ConsultantSkills* then have?

## Many-to-many relationships

Careful design of the data model would probably have led to the same set of tables. This situation would probably have been modelled in the design process as a **many-to-many** relationship between *Consultant* and *Skill*, and our solution is a typical representation in the database of this type relationship. Since a relational database cannot handle a many-to-many relationship between two tables, the additional table *ConsultantSkills* is required to make this work.

Remember the rule for representing a many-to-many relationship:

» *Where two entities have a many-to-many relationship, this is represented in the database with an additional table which has a many-to-one relationship with each of the two tables representing the entities.*

# Third normal form (3NF)

Tables in second normal form can still cause problems, as shown in the following example. Let's say that the company has two offices, in Glasgow and Edinburgh, and each consultant is based in one office. We'll try to store this information in the *Consultants* table:

## Consultants

| consultantID | firstname | lastname | office | address | phone |
|---|---|---|---|---|---|
| 1001 | John | Smith | Glasgow | Cowcaddens Road | 0141 123 4567 |
| 1002 | Amy | Jones | Edinburgh | George Street | 0131 987 6543 |
| 1003 | Jane | Lee | Glasgow | Cowcadens Road | 0141 123 4567 |

This table is certainly in first normal form.

It's also in second normal form, as the primary key is the single field *consultantID*, so no field can possibly depend only on part of the key.

However, there's still a problem with repeated data. All the details for an office are repeated for each consultant who is based at that office. Remember that repeating data unnecessarily can lead to inaccuracies in the data – for example, look at the spelling of "Cowcaddens" in the first and third rows.

The problem arises because the values of *address* and *phone* are **dependent on the values of more than one field**. For example, given the *consultantID* is 1001, you know that the *address* is "Cowcaddens Road". But, given the *office* is Glasgow, you also know that the *address* is "Cowcaddens Road". We can write these dependencies as:

*consultantID* → *address, phone*
*office* → *address, phone*

Going to third normal form will help. Here's the definition:

» *A table is in **third normal form** if it is in second normal form AND no non-key fields depend on any fields that are not the primary key.*

The way to fix a table which is not in 3NF is:

» *Remove the non-key fields that are dependent on a field (or fields) that is not the primary key. Create another table with those fields and the field(s) that they do depend on.*

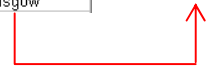
Applying this fix here gives us:

## Consultants

| consultantID | firstname | lastname | office |
|---|---|---|---|
| 1001 | John | Smith | Glasgow |
| 1002 | Amy | Jones | Edinburgh |
| 1003 | Jane | Lee | Glasgow |

## Offices

| office | address | phone |
|---|---|---|
| Glasgow | Cowcaddens Road | 0141 123 4567 |
| Edinburgh | George Street | 0131 987 6543 |



Now

- The values of address and phone for each office are stored once only
- It's not possible to enter an invalid value for office in *Consultants* because each value must match a value in the *Offices* table.

## Summing up the first three normal forms

This quote is taken from Clare Churcher's book on Beginning Database Design:

*A table is based on:*

- *the key;*
- *the whole key;*
- *and nothing but the key (so help me Codd)*

## Higher normal forms

For most cases, normalising to third normal form will take care of the likely problems. There are two more normal forms, **4NF** and **5NF**, which deal with more subtle problems. There is also **Boyce-Codd normal form**, which privdes a single statement which approximately encapsulates the first three normal forms. These are beyond the scope of this module – you will learn about these later in your course.

## When to use normalisation

Accurately identifying entities and their relationships and designing a data model tends to lead to a database schema which is pretty well normalised. Most of the examples of problems are based on tables which were badly designed on purpose, and which wouldn't have come about with a good data model.

However, sometimes databases are created by people without knowledge of data modelling. There are *many* examples of databases in real world use which do not even come close to first normal form.

**So why normalise?** There are basically two reasons to do so:

> *To check the database schema and highlight any flaws in the data model design or the way it has been represented*

or

> *To fix a database which has already been created without the aid of a suitable data modelling process*