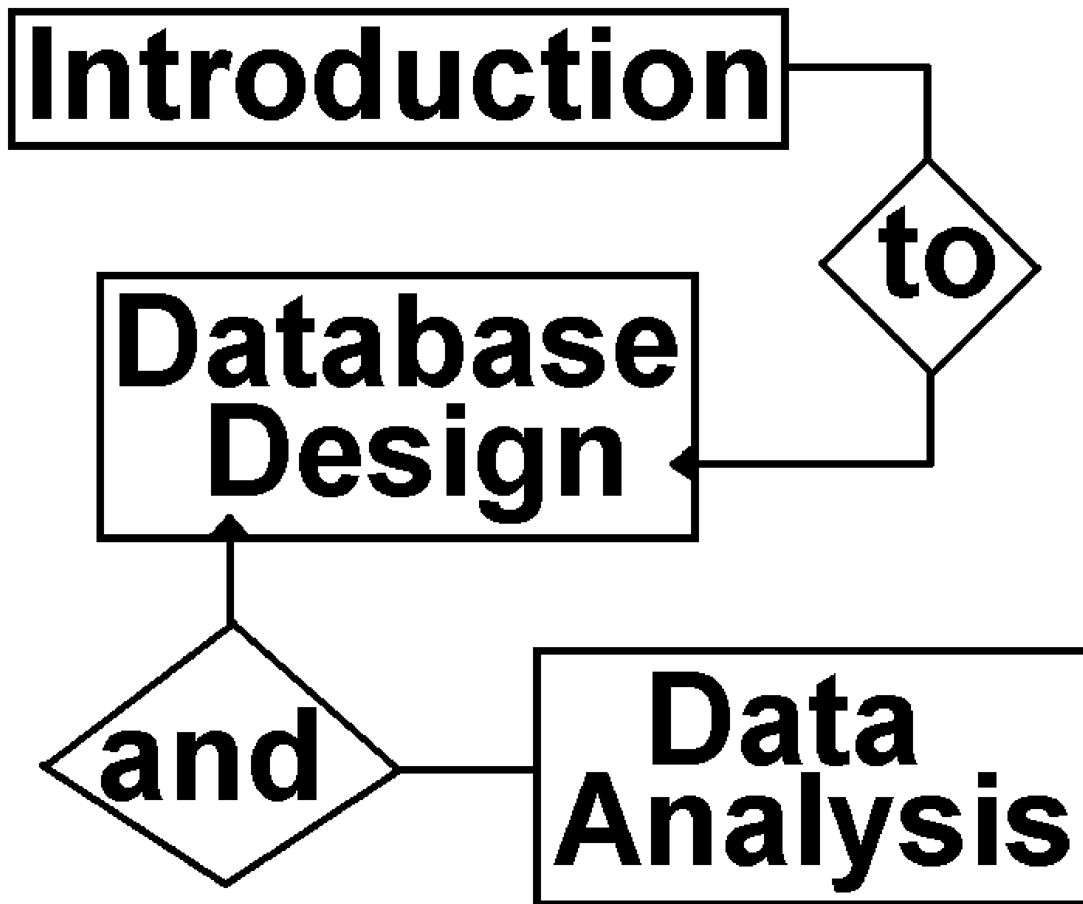


How and why correct planning is essential to the long term operational life of a relational database

Using *Microsoft Access* as the working example

©Rob Davis, M.Sc., M.I.A.P.

Copyright

This document is placed in the public domain, and may be used and distributed freely.

© Rob Davis MSc MIAP
Telford, Shropshire, United Kingdom

07976 379489
email elsham@blueyonder.co.uk
Internet www.robdavis.webhop.org

Printed on Thursday, 11 February 2016

The mythical town of Binster is a tribute to Nigel Roberts

Many thanks to staff and especially the mature students at Babington Community College, Beaumont Leys, Leicester, who during development of this book, were enthusiastic and cheerfully critical guinea-pigs.

Terminology

Database	A set of data items, ordered and stored electronically, for retrieval and analysis.
Table	One logically related set of data items, stored within the database.
Record	One entry from the table; a horizontal section of the data.
Field	One component part of a table; a "column" or vertical section in the list.
Key Field	The field which contains data uniquely identifying any single record.
Posted Key	A field which contains the key field from another table in the database.
One-to-Many / one:many	A relationship whereby one record of a table is looked up by many records in another table.
Decomposition	The process of recognising and creating many-to-many tables.
Many-to-Many / many:many	A relationship whereby many records of a table are looked up by many records in another table.

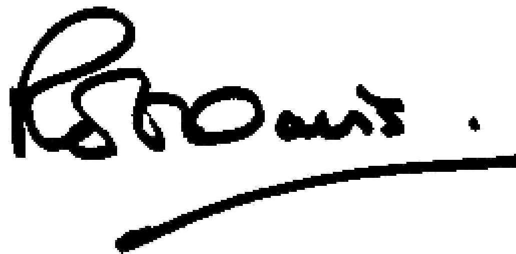
Objectives

The purpose of this book is to demonstrate that the science of relational database design is best approached on paper rather than at the computer keyboard.

Once this book has been read, and the examples put into practice, the reader will be familiar with the pitfalls of database design, and be able to avoid the most common design errors which so beset the beginner. He or she will be able to collect and analyse the various pieces of data which are to be stored, and then design a reliable data model as a conceptual exercise using a standardised methodology.

Furthermore, the book covers identifying design faults in an existing database, showing the reader how to recognise and correct such mistakes, resulting in a reliable, expandable database which will give the results expected of it.

Although the analysis and design process is independent of any particular hardware or software, the practical computer exercises are set in the popular environment of Microsoft Access.



11 February 2016

For Sandy

Contents

Introduction

Part 1 : Information is Power

Why store information anyway?
Get off your horse and write your database
The RDBMS in action
The Reason for Data Analysis
Gathering and Collection
Why all this bother?
Why tables are related; storing data twice?
Beginners' Common Mistakes : Field errors
Beginners' Common Mistakes : Data Redundancy
Self Test Exercise and Solution
Example : Motor Vehicles & Owners
Why change anything?
Design Exercise : DVD Hire Shop, and Solution

Part 2 : Repairing the Damage

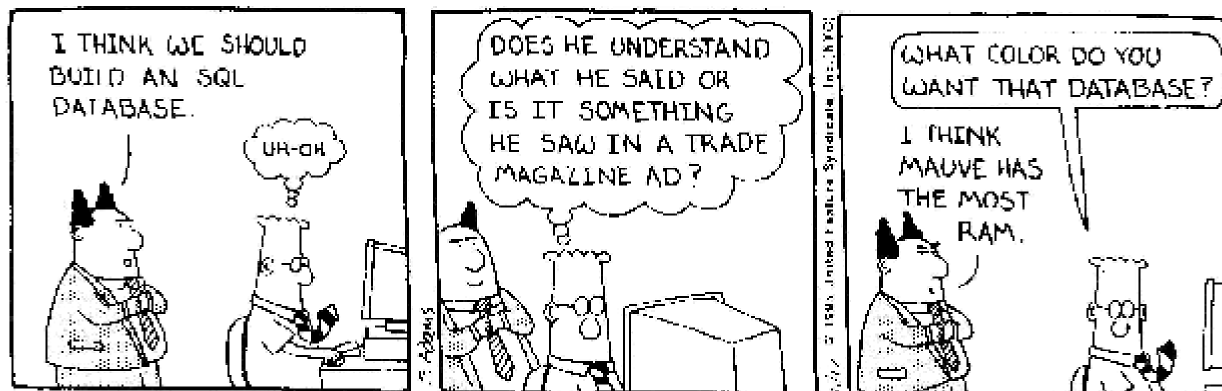
Redundancy example, using real data
Recognising redundancy
Removing the redundant data
Other Types of Redundancy, and Assumptions
Creating The Relationships
One-To-Many
Lookups in action
However... caution!

Part 3 : Data Modelling and Database Design

Avoiding major surgery by advance planning
Entity-Relationship Diagrams
An Entity-Relationship Diagram (ERD) explained
Another example of a many-to-one relationship
Many-to-many relationships
One-To-One Relationships
Null, empty or blank data fields
Skeleton Tables
Checking the ERD and Skeleton Tables
Creating the full data model

Summary

Introduction



The tremendous advances made in the field of computer hardware over the last twenty years have resulted in wide availability of PCs, or personal computers. Increasingly higher performance allied to constantly lowering prices has allowed educational establishments to offer a wide range of courses in computing and Information Technology. Nowadays, schools without computer facilities for students are unknown.

Such steps forward have put a powerful, versatile workhorse within the reach of ordinary families as well as even the smallest of businesses. Schoolchildren, students and the small business can easily buy a PC and use it for word processing, accounts, cash forecasting and similar tasks. The general “usability” and “user friendliness” of the archetypal word processor or off-the-shelf account package is hugely important to the student or businessman. In the field of making the software easy to use, yet offering powerful features, there has been as much advancement as in the hardware on which it runs.

Strength is also weakness

The Relational Database is the most misunderstood application on most hard disks and is rarely, if ever, used effectively by untrained people.

Whilst a low level of training in word processing or spreadsheets can produce impressive results, this is not the case in the world of relational databases. Advances in built-in features and usability have given the common man a tool which is, paradoxically, too powerful for the untrained. ***In short, the modern database’s power is also its weakness***. Whilst the latest up-to-date word processor becoming easier to use is nothing but good for the computing public, the modern relational database is a different story.

Its ease of use lends itself readily to single-table or flat-file storage of information, but this is like buying a Ferrari to go once a week to the corner shop. Only when the ***relational database management system*** (RDBMS) is used to link logically related data together, does it show its true mettle. And there lies the heart of the problem; the database development system is no tool for the beginner to handle, and to use it effectively requires considerable study of data analysis techniques.

Constructing a database with correctly related items is no easy task for the untrained person, as anyone who has tried it will have found out. Misunderstandings about how and why data

is associated results in a relational database which probably works for a short while - and then, just as probably, fails, with a sea of inexplicable misassociations. The aim of this book, therefore, is to show how and why data is associated, and how the traps for the untrained can be, at least, avoided, or at best, eliminated.

Although this book is not allied to any particular database application, the popular Microsoft Access database system will often be used as an example. ***This book is not intended as an Access manual***, and therefore assumes a working knowledge of the Access database system, i.e. starting and exiting the database software, creating tables, queries and forms, saving and retrieving work, etc.

Part 1 : Information Is Power

Why store information anyway?

Information, goes the old saying, is power. The smallest snippet of information can slot into its place and allow the user to see the complete picture; studying a sorted list can easily identify where something is obviously wrong; and analysis of data can show trends and histories. In effect, storing information is the foundation of knowledge, whether it's done on a computer or not. Had mankind never developed writing, no written records would ever have been available, and civilisation would rely entirely on the spoken word, reducing the passing of skills from father to son as reliable as a game of Chinese Whispers.

If we haven't got it, says the data analyst, we can't use it. If we didn't bother to store potentially important information, when the time comes to make decisions on what actions to take with our business or private life, we don't have the foundations on which to base our decision.

Get off your horse and write your database

Deciding what information is important enough to store is a crucial skill, and we will see how a great deal of thought and advance planning is required with what looks like even the simplest of databases. Those who go in *John Wayne* style - i.e. *guns blazing* - and attack the keyboard with little or no advance planning, find out quite soon that they have missed storing key information - and may well have stored information which is not, after all, required.

EVERY HOUR OF DESIGN TIME SAVES TEN HOURS OF PROGRAMMING OR DEVELOPMENT TIME

The RDBMS in action

The sheer friendliness of the modern Relational Database Management System (RDBMS) lends itself to allowing the user to create a simple, usable database in just a few minutes. The automatic generation of many of the component parts of the database give the user nothing more than remote control of what the RDBMS is doing “behind the scenes.”

In a conventional programming language such as C, Visual Basic or Delphi, the programmer must write the entire program by hand. This means that:-

- he must be fluent and competent in the programming language itself;
- he must understand what data is actually to be stored.

In a RDBMS, the software itself does away with any need for fluency in any kind of programming language. Whilst some database developers argue that the RDBMS itself is a programming language, it is not, as it allows the user little or no control of how it does what it’s been told to do.

For example, the language programmer is free to develop faster search routines, code for sorting and so on. The author of a database “written” in a RDBMS has no such control and has to accept the built-in ways of doing things. Both, however, retain full control of the user interface, often created - especially in one of the visual programming languages - with very similar tools.

Having said that, many RDBMS have a built-in programming language, often a BASIC-like system which allows the developer to write program code to make the database do something which it can't do itself. This assumes of course that the developer is fluent in the provided language, and it must be made clear that a very wide variety of features and functions are available without recourse to any code programming at all. However, sharp programming skills can make a standard database pick up its skirts and run, performing all sorts of clever tricks it didn’t do before.

With traditional programming languages such as Visual Basic, Delphi, C++ known as ***Third Generation Languages***, RDBMS are often known as ***Fourth Generation Languages***, these being the next logical step in developing software.

Easy and automated as creating the data structure is, the user still needs a thorough understanding of design concepts, or the potential power of the database application cannot be unleashed. At best, it becomes a case of holding the tail of the tiger, and, in the fear of being devoured by the animal, not being in a position to let go.

This is hardly a suitable position for the database developer to be in, and we will look at ways that the monster can be understood and, subsequently, tamed.

The Reason for Data Analysis

The technique of data analysis involves stepping back from the minutiae of the proposed database, and taking an abstract or high level view of the desired product, identifying the information which is to be stored. This information is then grouped into **Entities** which form the cornerstones of the data model, the foundation on which the actual database will be built. Upon implementation in software, the **Entities** will become the **data tables**, and have **fields**. Each data table has a **key field** which uniquely identifies any single data **record**.

It is all too common for programmers or database developers with little or no experience in data analysis to build a database with scant planning. Such hastily built systems will rarely work for very long under operational conditions and are likely to contain significant amount of duplicated or **redundant** information, which is wasteful of system resources and makes the resulting database difficult to maintain and update.

Gathering and Collection

Faced with the prospect of constructing a database, and recognising by now that a considerable number of problems can be eliminated or avoided by some attention to the design, what is the next step in the process?

Ø ***Consult the end users***

Firstly, it is of paramount importance that the end-users of the database be consulted in order to ascertain just what information they require to do their jobs. It is all too easy to speak only with people working at higher levels than the end-users, and subsequently discover that their idea of what information is relevant is different to those who will use the database operationally.

Ø ***What system, if any, already exists?***

Is there an existing system already in place? What is it? Why does it fall short of what is required? Where does it perform well? Only by speaking to the users at all levels will a complete picture be formed of what the fine points of database will be. For example, line or departmental managers will have well formed ideas about what they want the database to tell them. Actual hands-on users will be familiar with what data is required to produce that output.

If there is a card-index system, or other documentary items, ask for samples, with real or dummy data on them. Speak to the operators and users and find out what they would like the system to be able to do, and obtain examples of the data with which they are working.

Ø ***Make a list of the data items to be stored ...***

The primary stage of the design process is to compile a list of data items, which will become the **fields** of the system. This list, at the moment, will be quite random and unstructured; in effect, “throwing all the data fields into a hat” for analysis.

Ø ... and group the resulting items

When all the users have been consulted, tidy up this list and distil the collection of data items into groups. It is these groups which will later form the Data Tables, and be the basis of a diagrammatic representation of the data model.

Ø Don't shoot in the dark!

It is pointless to try and build a database with no clear picture of what the end result will be. The process of analysis is to form, in the database developer's mind, a crystal clear picture of the proposed system. With this, there is a clear target at which to shoot; with no such picture there is nothing towards which to aim.

Ø Examine each group

Once all the data items have been listed and divided into groups, it should become apparent that many of the items, or ***fields***, occur in several of the groups. A good example is a simple list of customer information. Such data will be required in many other groups:-

Sales enquiries
Sales transactions
Delivery notes or drivers' manifests
Invoicing
Customer care
a telephone contact list.

Ø Does data occur in more than one group?

To repeat possibly hundreds or even thousands of customer addresses in all of these groups of data would clearly be time consuming to bring about, wasteful of computer resources (disk space and processing time) and very awkward to maintain. Suppose a customer moves premises? Every single instance of this information must be tracked down and altered. If one instance was overlooked, for example the delivery address details were not changed, even though the invoicing address was changed, drivers would be sent to the old address, incurring time and running costs which should have been avoided.

When data fields are common to several groups, or **entities**, these fields will be stored just once, and the relational database management system will be able to relate the "one" entry of this information to where it is used, or looked up, "many" times. This action, known as a "many-to-one" relationship, forms the core of database design.

Later in this book, the diagrammatic method of planning databases on paper or whiteboard will be explained. For the moment it is sufficient to recognise that the importance of the analysis part of the process, which occurs long before any actual keyboard work is done, should never be underestimated.

Summary of the data collection stage

- **Consult** with all the database users
- List all the data items, or **fields**
- Group these into **entities**
- Identify the **fields** which occur in several **entities**.

Why all this bother?

In the future, after the database has gone into service, it is inevitable that additions to it will be required. In what form such alterations and additions will occur is rarely possible to determine in advance, so the analysis technique so far concentrates on building a rock solid data model which will stand unlimited expansion.

As the database grows, adding extra **data tables** is greatly simplified if the initial model is absolutely sound. Faults and shortcomings in the design have a dreadful habit of raising their ugly heads later in the database's operational life. Given that the analysis and modelling is correct, future expansion is readily achieved. It can be argued that a well designed database, when under expansion, requires little surgery; a poorly designed one, when being enlarged, requires considerable alterations.

In the next sections, common mistakes in design by beginners, and the methods of recognising data **redundancy**, and its elimination, will be covered.

Why tables are related; storing data twice

Only the most simple database will have one table of data. In any single-table database, it will probably be apparent that the contents of some fields will be duplicated. Even in a simple address-book database, fields such as ***Town*** or ***County*** are likely to contain many identical entries. This is hardly likely to pose problems for the hobbyist, but in an industrial environment, where a data table will store thousands of records of data, the endless repetition of data is both inefficient and difficult to maintain.

Even a county name will be entered many times, and if the database is for a small business which deals mainly with customers and suppliers in its own local geographical area, the “home” county name will recur very frequently. What would happen if, as happens from time to time, county boundaries are shifted, or a county name is changed? There could be huge numbers of changes to be made to the data as a result.

Thinking of a table's contents displayed in a columnar layout, looking down the information held in any **field** is likely to reveal duplication. Such duplicated data is termed **redundant.**

In an ideal database, no data item is stored twice. If redundancy occurs, the redundant data should be extracted, the duplications removed, and the resulting data formed into a new, separate table. This process is called **normalisation.** (There are some exceptions to the “don’t store it twice” rule; these fall mainly into the region of common sense, and will be covered as they occur.)

Let's take the aforementioned address-book, where thousands of data records exist, and the county name of ***Binstershire*** recurs very frequently. Storing this county name occupies at least – count them – 12 bytes of disk space. As the actual field or field called "County" is almost certainly bigger than this, in order to allow for any possible longer country names, it should be clear even to the budding data analyst that great scope for normalisation exists.

The normalisation process will create a new table of data called ***Counties***, in which each county name will occur just once. Each record of the data will contain the actual county name, plus an extra identifying field – the key field – which makes it unique. This identifying field will probably be of the auto-numbering type, which requires, using Access as the example, just 4 bytes to store.

Returning to the original data, the contents of each county name field is altered to contain the corresponding identifying field from the ***Counties Table***, and the field type is then changed to numeric rather than textual.

The normalised data in the ***Counties*** table is then **related** to the original table, which can perform the process of looking up the related fields and returning with the required content. This has a very significant positive effect on the efficiency of the database.

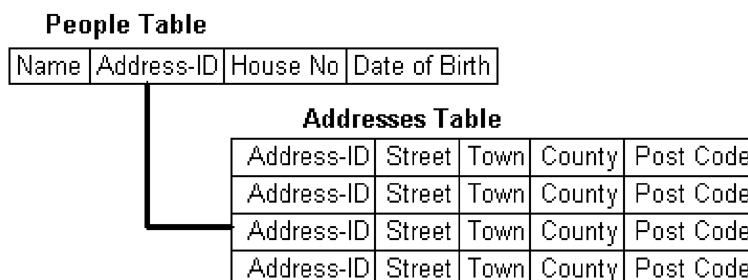
The gains in disk space can be very large. Taking an example database of 10,000 records, each of which before normalisation has a County field of 20 characters, this represents 100,000 x 25 = 2,500,000 bytes, i.e. a quarter of a megabyte, just to store the counties information. To store the normalised data will occupy 25 x the number of unique county names, let's say there are 30 county names; the ***Counties*** table will then be 25 x 30 bytes, plus 4 x 30 bytes for the identifying fields; that makes 870 bytes. Reducing the original table's County field from 20 to 4 bytes, which is the size required to store the corresponding key fields from the ***Counties*** table, saves (100,000 x 25) – (100,000 x 4) = 2,100,000 bytes.

The maths are not quite as clear cut as this because there is a small amount of increase in the file size when the extra table is created, and database puts into effect the various relationships. However, the disk space gains alone are massive, and the speed and efficiency of the database increases as less data has to be processed.

However, disk space these days is cheap, and this factor alone is not the chief motivating issue behind normalisation. The prime reason is to build a readily maintainable, expandable database which is efficient in both operation as well as disk space.

With related tables, a field or data field in one table of data holds nothing more than a pointer to the related data in the other table.

Diagrammatically, like this:-



The ***People Table*** looks up the full address details from the corresponding entry in the ***Addresses Table***. There may be many instances of People with the same address; but

each address will exist only once in the database, and be looked up by many records of data in the ***People Table***. To put this another way, the data record from the ***Addresses Table*** is "plugged in" to the related data record of the ***People Table***. Thus, any ***<u>one</u>*** record of data in the ***Addresses Table*** can occur ***<u>many</u>*** times in the ***People Table***.

The concept of a "one-to-many" or 1:M relationship is the fundamental design aspect which should be firmly understood at this stage. Many-to-one has the same meaning.

Beginners' Common Mistakes : Field errors

To illustrate the importance of the planning and design stage of a database, let's take an example of a local Council's database concerning Council Tax. Design errors which take the council by surprise once the database is operational will have major consequences:-

- Ø tax may go uncollected, reducing council revenue;
- Ø payments may be made but lost in the system;
- Ø bills may be sent to the wrong people;
- Ø householders may pay more (or less) than the correct amount;
- Ø tax dodgers may go unnoticed, or be untraceable.

There will be local inhabitants' names and addresses, property details, tax band details, amounts paid, etc. being handled by the database. In a situation like this, the untrained database developer will build a table of data like this, where one record of data refers to a single person who is eligible to pay tax:-

Tax Payers Table

Field name	Data Type	Size	Examples
Title	Text	5	Mr; Mrs; Miss
Name	Text	50	John Smith, Mary Jones
Address	Text	250	99 Main Street, Binster, BI1 1RT
Age	Integer	2	45; 27; 70

There will be massive data redundancy and several other problems if the table is built like this. Taking the other problems first:-

The question of which field will uniquely identify any single record has not been addressed in this initial data model. Such an identifying field, called a ***Key Field***, must exist in order to allow this table of data to be related to another – if not now, then in six month's time. Building in this sort of expandability is essential.

Finding the ideal data item to serve as the identifying "tag" is quite absolutely vital. Here, the technique is to look at samples of the existing data, and to speak to those who are familiar with it:-

- Ø the payment clerks
- Ø enquiries and "front desk" staff
- Ø tax adjustment and assessment officers
- Ø accountants

Is there an existing data field which might be used as the key field? Very often there is, and speaking to the proposed end-users of the new database will often turn up a suitable key field. Such has the important advantage of being known and recognised by the staff already.

In the Council Tax concept, what fields or fields are potential key fields? Name perhaps, until another person with the same name occurs; as the key field must be unique, and there are likely to be many people with identical names, then Name clearly won't do the job.

A combination of name and age? Again, this will work until another person with the same name and the same age crops up. This may never happen, but if it does, the database will fail, and what we are trying to bring about is a rock solid database design, where this problem is eliminated.

Going back to the principle of speaking to the end-users, and in a case such as this, there will certainly be an existing method of identifying each person on such a system. There will be a reference number of some sort. This should be used unless there is an overwhelming reason why not, because (a) the database end-users will be familiar with it; and (b) later on it may be necessary to link this database with others, which use the existing method of identifying tax payers.

Therefore the first correction to the proposed table design will be the ***Key Field.*** In order to make it clear during the design process that this is the key field, a hash symbol, #, is appended to the end of it.

The # symbol is also used to signify that a data field is either: (a) held in the table in hand but looked up by another table; or (b) held in a separate table and looked up by the table in hand.

Let's have a closer look at each of the proposed data fields:-

TaxPayer# is going to be the key field for this table, and thus has the hash symbol appended to it. Remember that this is for humans to pick it out easily, later on.

Title looks as if it will fit the bill; 5 characters is fine for the most common titles such as *Mr* or *Mrs*, even *Rev'd* will fit without any problems. Suddenly, along comes *Squadron Leader* Jones. True, the title can be truncated a little to *Sqn Ldr* or *Sq Ldr* or even *S/Ldr*. Really, however, the width of the title field is insufficient. This problem could have been eliminated by looking at examples of what data was required to go in this field, and adjusting the width or size accordingly.

Some people even become annoyed if their titles are not given in full.

Name presents a multitude of problems. Beginners often create one field for this, and then find themselves unable to sort the data by surname. True, the name could be entered as *Smith, John* but the problem is better addressed by separating the fields into two, Forename(s) and Surname. This is a very common mistake by beginners and one of the most easily avoided. Imagine the problems of creating a database where the Name is one field, having the end-users type in 100,000 names, and then having to add a Forename field. The poor suffering end-users will not be desperately pleased to be asked to either re-enter the data, or edit out the forenames and initials into a separate field. The database developer isn't going to get asked to the staff barbeque, except maybe as a candidate for being roasted.

The next issue is ***how wide should these fields be***? How many characters are to be allowed to store their data? Again, the problem can be largely eliminated by examination of the data before the table is built. It's pointless to allocate a Surname field of just 25 characters when there are likely to be hyphenated or double-barrelled – even triple-barrelled – surnames. There is no magic answer, except examination of

the data, and speaking to the end users about the kind of names they encounter during the course of their work.

Age is yet another common error for the beginner. It is far more useful to store the date of birth, from which the age can be calculated. If the age itself is stored, the data will need regular updating or housekeeping, to keep the ages current as the years pass. Store this data item as a ***Date of Birth*** and the problem is eliminated.

It is also extremely useful to store a ***Post Code*** in a separate field to the address, for purposes of sorting. This, however, brings in the question of ***Data Redundancy.***

Beginners' Common Mistakes : Data Redundancy

At this stage we have a collection of fields or fields, drawn from some sample data, and we have what looks like the makings of a potentially useful data table.

Now let's turn to the question of what information is likely to be repeated in a database of this nature. The most obvious example is the address, as taking a typical household with two parents and two over-18s, the address where they live will exist in the database identically four times. A block of flats, with many individuals effectively living at the same street address, is another example. The technique of the data analyst is to recognise redundancies such as this and eliminate them, before fingers stray anywhere near a keyboard.

Taking all the points discussed in the previous two sections, the improved data tables would look like this:-

Tax Payers Table

Field name	Data Type	Size	Examples
TaxPayer#	Long Integer	4	8161524
Title	Text	10	Mr; Mrs; Miss; Sqdn Ldr
Forename(s)	Text	30	John Kevin; Frederick Richard
Surname	Text	35	Thompson; Revell-Carter
House Details	Text	15	103; The Limes, Flat 19
Address#	Long Integer	4	(looked up in the Addresses Table)
DoB	Date/Time	4	05-Feb-1954

Addresses Table

Field name	Data Type	Size	Examples
Address#	Long Integer	4	probably an auto-incrementing number
Street	Text	30	Main Street; Beggars Lane
District	Text	30	Ashley; Gaumont Park
Town	Text	30	Binster; Casterbridge
County	Text	30	Binstershire; Rutshire
Post Code	Text	9	BI8 9YR; PC1 9RD

In the above examples, the actual details of each address are stored just once in the ***Addresses Table***. Each record of data in the ***Tax Payers Table*** has a field which contains the individual details for each household, such as the house number or name, as well as a field which is related to a corresponding entry in the ***Addresses Table***, and looks up the data therein.

This has huge benefits. Much computer disk space is saved, and processing of the data is speeded up. Most significantly, a change to a record of data in the ***Addresses Table*** is immediately reflected in any records which look it up from the ***Tax Payers Table.***

Suppose that an address has been mistyped as *Bunster* instead of *Binster*. This may never come to light – except that a tax payer may never receive a bill, or perhaps never be sent one! Database operators might spot the mistake, but they might not… and to check every one would be impossible. If the data is related in the above example, a change to the ***Addresses*** data will instantly be recognised by the ***Tax Payers*** data.

The result looks like this:-

TaxPayer#	Title	Forename	Surname	House Details	Address#	DoB
0123456	Mr	John James	Smith	12	45688	5/2/54
0123457	Mrs	June Joan	Smith	12	45688	19/2/53
0123458	Miss	Elizabeth Jane	Smith	12	45688	4/6/76
0123459	Mr	Richard Peter	Smith	12	45688	19/7/77
0123460	Rev'd	Peter Louis	Yates	The Rectory	45689	1/12/44
0123461	Mrs	Sarah Rachel	Yates	The Rectory	45689	17/9/46

Address#	Street	District	Town	County	Post Code
45668	Prescott Lane	Merlon Park	Binster	Binstershire	BI1 1WE
45689	Dawes Mews	Uptown	Binster	Binstershire	BI2 3QW

An astute eye will see that there is more scope for removing duplicated, or redundant, data. Towns, Counties and Post codes will be repeated. In a correctly built database, the next step would be to create the appropriate tables to do exactly this.

This goes to show that the first level of database design will usually produce further opportunities for improving the data modelling, and the process of refinement will frequently take several cycles. It will nearly always be necessary to refer back to the end users several times, in order to clarify the picture of the database in the developer's mind.

Well informed end-users will also be able to make recommendations for what data they would like to store, and attention should always be paid to such comments. Remember; these are the people who will actually use the new system, and great pains should be taken to give them what they actually need, rather than what the database developer thinks they need.

Making It Fly

The benefits of a rock solid design will become apparent when, for the next stage in the Council Tax scenario, we are asked to bolt on a method of recording payments received. First of all let's look at the *untrained* database developer's solution:-

Payments Received Table

Field name	Data Type	Size	Examples
Surname	Text	40	Smith, Stanford-Tuck
Forename(s)	Text	20	Arthur, Doreen Mary
Address	Text	250	Full address including post code
Date Paid In	Date/Time	4	28-July-1999
Amount Paid	Currency	4	£123.45

What redundancy occurs here? Bearing in mind that most Council Tax payments occur over a 10 month period, i.e. for every household there will be 10 annual payments. A simple calculation of the above table reveals that about 320 bytes of data will be required to store each payment; assuming 250,000 properties, each with 10 payments, makes a disk capacity of (250,000 x 10 x 320) 800 megabytes per year, and that's on the conservative side. A

more labour saving method, which will reduce this somewhat liberal usage of disk space, would be to use the existing Tax Payers table to be related to each payment, and the resulting alliance performing the necessary lookups:-

Payments Received Table

Field name	Data Type	Size	Examples
TaxPayer#	Long Integer	4	[Looked up in Tax Payer Table]
Date Paid In	Date/Time	4	28-July-1999
Amount Paid	Currency	4	£123.45

This gives huge advantages:-

- Ø A reduction in record or record size from 320 to 12 bytes
- Ø Storing a year's data requires just 30 megabytes
- Ø The database can readily look up addresses and post codes automatically
- Ø Data entry is enormously easier, as automated tools can be used to give point-and-shoot "combo box" type lookups. This almost eliminates typing errors, as operators are selecting or clicking on a drop-down list of names rather than typing in details themselves.

Recognising the potential of repeated or redundant data is a key skill for the analyst and database developer. Well planned systems are easily expanded; poorly planned systems require major surgery when add-on features and extra data tables need to be added.

Self Test Exercise : Adding Council Tax Bands

It is essential in this example Council Tax database to store information concerning in which payment band each property occurs, and what the annual payment figure is for each Band. Examples of Payment Bands are:-

Band A	£300 per annum
Band B	£400 per annum
Band C	£500 per annum
Band D	£600 per annum

Each property must have a Payment Band associated with it.

- Ø Is it necessary to store the Payment Band details in the ***Addresses Table***?
- Ø If this was done, what problems would result?
- Ø How can the data for Payment Bands be correctly normalised and related to the ***Addresses Table***?

Don’t turn the page until you have attempted to solve these problems yourself, on paper.

Payment Bands : the solution

Storing the details of Payment Bands as part of the Addresses Table will result in considerable data redundancy. Also, when the annual amount payable for each Band changes, as it will almost every year, it will be a tedious and error process to track down and change every single instance over many thousands of records of data held in the ***Addresses Table.***

The correct solution is as follows. A new ***Payment Band Table*** should be added to the conceptual design, along these lines:-

Payment Band Table

Field name	**Data Type**	**Size**	**Examples**
Payment Band#	Text	1	A, B, C, D
Annual Cost	Currency		£300, £400, £500

This table stores details of each Band only once. Remember that the # symbol is simply to aid us humans to pick out the key field quickly when we need to.

Then, the *Addresses Table* is adjusted to include an extra field which will hold the key field from the ***Payment Band Table***:-

Addresses Table

Field name	**Data Type**	**Size**	**Examples**
Address#	Long Integer	4	probably an auto-incrementing number
Street	Text	30	Main Street; Beggars Lane
District	Text	30	Ashley; Gaumont Park
Town	Text	30	Binster; Casterbridge
County	Text	30	Binstershire; Rutshire
Post Code	Text	9	BI8 9YR; PC1 9RD
Payment Band#	Text	1	Looked up in the ***Payment Bands Table***

This inclusion of the key field from another table, for lookup purposes, in called a ***posted*** key field, sometimes also referred to as a ***foreign key***.

It should be apparent, from this need to change the design of a table, that there is a constant repetitive or iterative approach to database designing, with changes being made to the paper-based design, as the system and its requirements grow.

This is a good example of why databases are not written “at the keyboard”. Mistakes are expensive in terms of time and effort, and once data tables are in service, radical changes to them cause an exponential number of changes to the other components of the database, which draw data from them.

Example : Motor Vehicles & Owners

Taking for example a database of ***Vehicles Owned***, as might be used by the DVLA at Swansea, any owner may own several cars; a national car hire company would own thousands of cars. To store details about the owner in a single table of vehicles would waste vast amounts of computer storage space and processing time; and if the car hire company changes address, every instance of their name in the ***Vehicles Table*** would need to be found and updated.

It is more efficient to store the Owner details once per owner in an ***Owners Table***, and include in the ***Vehicles Table***'s list of fields the key field from the ***Owners Table***. Thus, the software can read from disk the desired data from the ***Owners Table***, and display it, without having to store it more than once. This is a termed a ***lookup***.

If the data record in the ***Owners Table*** is updated, all lookups will immediately reflect the change; only one amendment is required.

Is it efficient to store the name of the vehicle's manufacturer, and the address, contact details, phone numbers, etc, (Ford, Vauxhall, Fiat, Renault) millions of times? Certainly not - such ***redundant*** data will be ***normalised*** into separate tables and ***related*** to other tables.

Why change anything?

If the data table has redundant data, which later changes, every instance of the old data must be found and changed. Think about the previous example of the database of motor vehicles and their owners, where an owner may own many cars (i.e. a car hire company). If ***Binster Car Hire***, who own 250 vehicles, now find themselves bought out by a new company who decides to rename the company to ***Binster Super Cars***.

Every instance of ***Binster Car Hire*** in the ***Owner*** field will need to be found and changed - a time consuming and error prone practice. However, if the data has been properly normalised, and the ***Owners*** kept in a separate table which is linked into the ***Vehicles Table***, all that is required to make this change of name is to amend the single entry for ***Binster Car Hire*** in the ***Owners Table***, and change it to ***Binster Super Cars.*** Now, whenever the ***Vehicles Table*** looks up the owner of a vehicle, the database returns the new name.

The same principle can also be applied to the car's manufacturer. Certainly it would be useful to store the manufacturer's full name and address, telephone details, and other pertinent information. However, to store such for every instance of a Ford or Renault in the vehicles data would be astronomically inefficient. There would be millions of Fords and Renaults in the data table; clearly the storage of the full set of manufacturer's details for every single record of vehicle data would occupy huge tracts of hard disk space and be very difficult to update if a manufacturer changed address, or amended its telephone number. *Every single instance* of that manufacturer's details would need to be tracked down and amended – a massive and error prone task.

It would be far more efficient to store the vehicles information in one table, and the manufacturers' details in a separate table. Diagrammatically, such a data model looks like this:-

BEFORE (one table)

A123ABC	FORD
B234BCD	FORD
C345CDE	FIAT
D456DEF	RENAULT
E567EFG	FORD
F678FGH	RENAULT
G789GHI	FIAT
H890HIJ	FORD

AFTER (two tables)

A123ABC	1
B234BCD	1
C345CDE	2
D456DEF	3
E567EFG	1
F678FGH	3
G789GHI	2
H890HIJ	1

1	FORD
2	FIAT
3	RENAULT

The other advantages are a great reduction in the physical size of the database, as only one instance of each previously duplicated field will be stored, and an improvement in performance.

One helpful way to think of this process is

the Three R's of Database Design

***R*edundant** (identify duplications)

***R*emove** (take these out and create separate tables)

***R*elate** (create lookup relationships)

Design Exercise and Self-Test

You are the data analyst engaged by the Binster DVD Library which rents out DVD films to members of the public, who have to join the DVD Club. Members pay a small annual fee and are then charged a rental for any DVDs they take out.

The DVD shop has the following requirements:-

1 Maintain a members' list and details of annual fees
2 Maintain a DVD library list
3 Track any rentals made by members - overdue list etc
4 Allow searches by title, starring artist, subject
5 Track all DVD usage, most popular rentals, etc

This is first best tackled from the most obvious angle - the problem of storing members' details. What are the likely component parts of this data?

Title
Forename
Surname
Address
Post Code
Telephone
Joining Date
Membership Expiry Date

These are the obvious data fields which spring to mind. Would it be a good idea to store a member's date of birth? What could be gained from doing this? Would the management find it useful to be able to see, from the list of DVD hirings, what type of film (i.e. adventure, horror, western etc) is most popular and unpopular with various age groups of members? Only discussion with the end-users of the database will resolve this kind of question, and it should be borne in mind that what the management wants is often different to what the staff in the shop want.

From the data model of the Members, it is clear that some redundancy will occur. As most DVD shops serve a small local community, there will be duplication of street name which probably would not occur with a large corporate database. It would be sensible in the DVD shop data model to store street names, together with their districts, in a separate table to the Members:-

Members Table

Field name	Data Type	Examples
Member#	Autonumber	
Title	Text / 8	Mr, Sqn Ldr, Miss
Forename	Text / 20	John, Alison, Mike
Surname	Text / 30	Williamson, Lloyd-Jones
Address#	Number/Long Integer	[looked up in Addresses Table]
DoB	Date/Time	05-Feb-1954
Join Date	Date/Time	01-Jan-2000
Expiry Date	Date/Time	01-Jan-2001

Addresses Table

Field Name	Data Type	Examples
Address#	Autonumber	
Address	Memo	12 The Green, Binster
Post Code	Text / 8	BI7 8YU

It should be seen from this that each address exists only ***once*** in the ***Addresses Table***, and is referenced or looked up by its corresponding entry in the ***Members Table***.

One aspect of the DVD shop data problem is the actual DVD films themselves, and here a number of problems are revealed. An untrained database developer will probably plan such a table this way:-

DVD Films Table

Field Name	Data Type	Examples
Title	Text / 30 ***(key field)***	Star Wars, Schindler's List
Male Star	Text / 30	Mark Hamill, John Wayne
Female Star	Text / 30	Dervla Kirwan, Whoopie Goldberg
Category	Text / 15	Adventure, Western, Adult
Director	Text / 30	Steven Spielberg, George Lucas
Rental Fee	Currency	£2.50, £3.00

On the face of it, this looks fine; however there are some really tricky problems which a structure of this sort cannot address. For example:-

- There is provision for only one male and one female star; major supporting actors are not identified. Also, there will be great redundancy in actors' and directors' names. Certainly extra fields could be added to incorporate supporting actors, but this will lead to even more duplication of names.
- A film may fall into more than one category.
- Rental fees will change regularly.
- Most DVD rental shops carry more than one copy of popular films. If several copies of a film are stored in the above table, how will each one be differentiated? And, there will be duplication of entries; Title, Stars etc will recur.

The classic way to recognise data redundancy is to mock up a table of this sort and to enter some dummy data, then sort it by each field in turn. Repetitions are readily spotted this way.

It would be beneficial to budding database developers to sit down for some hours with the above problem and work out a solution, following the design rules and concepts covered in

the preceding section of this book. A solution is given overleaf, but readers should try not to refer to it until what looks like a working paper-based solution has been composed.

DVD Shop - A Solution

At this stage, we are concerned with the ***Design***, not the ***Implementation***.

Members Table
Data about people who belong to the DVD club

Field name	Data Type	Examples
Member#	Autonumber	
Title	Text / 8	Mr, Sqn Ldr, Miss
Forename	Text / 20	John, Alison, Mike
Surname	Text / 30	Williamson, Lloyd-Jones
Address#	Number/Long Integer	[looked up in Addresses Table]
DoB	Date/Time	05-Feb-1954
Join Date	Date/Time	02-Jan-2010
Expiry Date	Date/Time	01-Jan-2011

Addresses Table
Address data for members

Field Name	Data Type	Examples
Address#	Autonumber	
Address	Memo	12 The Green, Binster
Post Code	Text / 8	BI7 8YU

Actors Table
Names of actors and actresses who appear in DVD films

Field Name	Data Type	Examples
Actor#	Autonumber	
Actor Forename	Text / 30	John, Mark, Demi, Jennifer
Actor Surname	Text / 30	Wayne, Hamill, Moore

Directors Table
Data about Directors who made DVD films

Field Name	Data Type	Examples
Director#	Autonumber	
Director Forename	Text / 30	George, Steven
Director Surname	Text / 30	Lucas, Spielberg

DVD Titles Table
Data about a single DVD film title

Field Name	Data Type	Examples
DVD#	Autonumber	
Title	Text / 75	Star Wars, Dune
Category#	Number / Long Integer	[looked up in Category Table]
Hire Group#	Number / Long Integer	[looked up in Hire Groups Table]

Categories Table
Data about film categories

Field Name	Data Type	Examples
Category#	Autonumber	
Category	Text / 30	Western, Cartoon, Adult

DVD Stock Table
Data about each copy of a DVD film stored on the shelf.

Field Name	Data Type	Examples
Stock#	Autonumber	
DVD#	Number / Long Integer	[looked up in DVD Titles Table]
Status#	Number / Long Integer	[looked up in the Hire Status Table]

Hire Status Table
Data about the hire categories of DVDs which are on the shelf

Field Name	Data Type	Examples
Hire Status#	Autonumber	
Hire Status	Text / 10	Available, Hired, Damaged

Hire Group Table
Data about different hire and fee charging bands for a DVD title

Field Name	Data Type	Examples
Hire Group#	Autonumber	
Group	Text / 1	A, B, C
Hire Fee	Currency	£2.00, £3.50

Hirings Table
Data about DVDs from the shelf which have been hired

Field Name	Data Type	Examples
Hiring#	Autonumber	
Stock#	Number / Long Integer	[looked up in DVD Stock Table]
Member#	Number / Long Integer	[looked up in Members Table]
Date Out	Date/Time	[today]
Date Returned	Date/Time	

This looks like a huge increase on the number of data tables originally expected - but don't worry - this kind of expansion is quite normal.

Questions on the design

- Why have two tables been planned for DVDs when the data might be incorporated into one? Because many instances of a title will be held, so that whilst a popular film's details will only be stored once, it will have many copies on the shelves. Each copy is treated separately where hirings are concerned.

Readers who have been able to draw up a list similar to the above, and who have correctly answered the questions, have correctly grasped the essentials of database design.

Readers who have identified shortcomings such as:-

- Ø no provision for relating actors and directors with their films
- Ø no provision for having a film in more than one category

-:can award themselves a Gold Star. These shortcomings will be addressed in the next section, because they require a rather deeper understanding of data relationships, and how to relate data not on a one-to-many basis, but many-to-many.

Part 2 : Repairing The Damage

Armed with the knowledge and techniques gained so far, most budding data analysts and database designers will now want to go away and look at an existing database which suffers from some of the problems covered in the preceding section. So we will not immediately look at the techniques of theoretical database design.

It is likely that there is an existing database which has been poorly designed, and which can be used as an example for corrective action. So, rather than cover at this stage the diagrammatic representation of how databases are designed, this section deals with recognising and removing data redundancy from an *existing* database.

It will now be necessary to use the sample data disk which accompanied this book. Before using the disk, make a backup copy of it in case of disaster. Start your computer and a copy of Microsoft Access 97, then open the database file called ***Car Hire Redundant.***

Redundancy example, using real data

The sample data on the disk refers to a series of car hirings by a vehicle rental company. There are 250 records of data in the complete table, but here just the first 18 are shown:-

Vehicle	Description	Hirer	Purpose	Served By	Mileage
E567EFG	1.9 Fiat Tempra, Yellow	Mrs Alison Masters	Holiday	Peter Bradley	9391
F678FGH	2.0 Vauxhall Cavalier, Grey	Mrs Alison Masters	Shopping Trip	Jane Evans	11875
B234BCD	2.0 Renault Laguna, Blue	Mrs Alison Masters	Holiday	Roger Thomas	7313
D456DEF	3.0 Ford Explorer, Black	Henry Peters Ltd	Courtesy Car	Paul Dawson	10179
E567EFG	1.9 Fiat Tempra, Yellow	Binster United Football Club	Shopping Trip	Mary Patterson	6590
A123ABC	1.6 Ford Fiesta GL, Red	Miss Pamela Jones	Holiday	Roger Thomas	9862
B234BCD	2.0 Renault Laguna, Blue	Binster United Football Club	Courtesy Car	Jane Evans	4954
B234BCD	2.0 Renault Laguna, Blue	Wallace Thomas Ltd	Breakdown Cover	Jane Evans	4027
E567EFG	1.9 Fiat Tempra, Yellow	Wallace Thomas Ltd	Courtesy Car	Jane Evans	3226
F678FGH	2.0 Vauxhall Cavalier, Grey	Rev James Roberts	Business	Jane Evans	1209
D456DEF	3.0 Ford Explorer, Black	Mr Paul Yorke	Courtesy Car	Jane Evans	5311
E567EFG	1.9 Fiat Tempra, Yellow	Wallace Thomas Ltd	Shopping Trip	Paul Dawson	686
B234BCD	2.0 Renault Laguna, Blue	Wallace Thomas Ltd	Holiday	Jane Evans	5857
B234BCD	2.0 Renault Laguna, Blue	Rev James Roberts	Business	Jane Evans	4820
A123ABC	1.6 Ford Fiesta GL, Red	Underhill & Co Ltd	Holiday	Jane Evans	2597
A123ABC	1.6 Ford Fiesta GL, Red	Mr Paul Yorke	Breakdown Cover	Jane Evans	6715
F678FGH	2.0 Vauxhall Cavalier, Grey	Lewis & Carr	Holiday	Jane Evans	8695
B234BCD	2.0 Renault Laguna, Blue	Wallace Thomas Ltd	Business	Jane Evans	2563

It is evident from this list that there are several fields containing redundant data, as many of the contents recur. For example, the vehicle details will recur many times, once for each hiring; the vehicle descriptions will recur continuously; and whilst a hiring may be a one-off, regular customers' names will crop up again and again.

Likewise, the names of staff who dealt with the hiring will be in the table of data many times.

Recognising redundancy

This is very often discovered at the data entry stage, i.e. when a data table is used for creating some test data to see if the table design is appropriate. In effect, redundancy is spotted by the Mk 1 eyeball. However, when less obvious redundancy occurs, the database's own tools will be required to identify where duplications exist. Such redundancies need to be identified, analysed and eliminated very early in the database's life cycle, or future expansion will be greatly hindered.

If any of the table's fields are shown sorted in alphabetical order, redundancy is obvious, even when just the first 21 records are displayed. Here, three have been selected and sorted.

Vehicle	Description
A123ABC	1.6 Ford Fiesta GL, Red
A123ABC	1.6 Ford Fiesta GL, Red
A123ABC	1.6 Ford Fiesta GL, Red
A123ABC	1.6 Ford Fiesta GL, Red
A123ABC	1.6 Ford Fiesta GL, Red
A123ABC	1.6 Ford Fiesta GL, Red
A123ABC	1.6 Ford Fiesta GL, Red
A123ABC	1.6 Ford Fiesta GL, Red
A123ABC	1.6 Ford Fiesta GL, Red
A123ABC	1.6 Ford Fiesta GL, Red
A123ABC	1.6 Ford Fiesta GL, Red
A123ABC	1.6 Ford Fiesta GL, Red
A123ABC	1.6 Ford Fiesta GL, Red
A123ABC	1.6 Ford Fiesta GL, Red
A123ABC	1.6 Ford Fiesta GL, Red
A123ABC	1.6 Ford Fiesta GL, Red
A123ABC	1.6 Ford Fiesta GL, Red
A123ABC	1.6 Ford Fiesta GL, Red
A123ABC	1.6 Ford Fiesta GL, Red
A123ABC	1.6 Ford Fiesta GL, Red
A123ABC	1.6 Ford Fiesta GL, Red

Hirer
Binster County Council
Binster United Football Club
Binster United Football Club
Binster United Football Club
Binster United Football Club
Binster United Football Club
Binster United Football Club
Binster United Football Club
Binster United Football Club
Binster United Football Club
Binster United Football Club
Binster United Football Club
Binster United Football Club
Binster United Football Club
Binster United Football Club
Binster United Football Club
Binster United Football Club
Binster United Football Club
Binster United Football Club
Binster United Football Club
Binster United Football Club

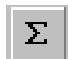
Served By
Jane Evans
Jane Evans
Jane Evans
Jane Evans
Jane Evans
Jane Evans
Jane Evans
Jane Evans
Jane Evans
Jane Evans
Jane Evans
Jane Evans
Jane Evans
Jane Evans
Jane Evans
Jane Evans
Jane Evans
Jane Evans
Jane Evans
Jane Evans
Jane Evans

Notice in this example that Binster County Council made just one hiring – but the vast majority of the data is repetitive.

This technique works, but it is rather clumsy, and requires a human eye to spot repeated data. A better method is to use the tools built into the database itself to perform a count of each time the contents of a field occurs; if a count of any field exceeds one, there are duplications in the data which are candidates for correction. In ***Access***, this is achieved in the following way, with the two fields grouped by ***Vehicle***, and the second column being a count of the first, where the count exceeds 1.

To achieve this count, it is necessary to ***group*** the data. Grouping means that reading the data columns from left to right, only ***one*** instance of duplicated data is returned. Adding the

same data field again, and changing the database's action from ***Group By*** to ***Count***, gives a numerical count of the total number of times each identical one occurs.



To make the query perform the desired grouping, click on the "Totals" button on the toolbar. Add the ***Vehicle*** field twice to the query builder.

Field:	Vehicle	Vehicle
Table:	Hirings Table	Hirings Table
Total:	Group By	Count
Sort:		
Show:	☑	☑
Criteria:		>1

Change the first instance of the "Vehicle" column to ***Group By*** and the second instance to ***Count***.

The result looks like this:-

Vehicle	CountOfVehicle
A123ABC	37
B234BCD	79
C345CDE	33
D456DEF	34
E567EFG	30
F678FGH	37

It is obvious here that significant redundancy on the contents of the ***Vehicle*** field occurs, and a similar exercise and examination of the data held in the ***Hirer, Purpose,*** and ***Served By*** reveals the following:-

Hirer, with only the first 10 records of the result shown, also sorted in descending order:-

Hirer	CountOfHirer
Wallace Thomas Ltd	38
Rev James Roberts	32
Miss Pamela Jones	24
Binster United Football Club	22
Mrs Alison Masters	22
Mr Paul Yorke	21
Underhill & Co Ltd	21
Mr John Smith	20
Lewis & Carr	18
Henry Peters Ltd	17

Purpose:-

Purpose	CountOfPurpose
Breakdown Cover	57
Shopping Trip	54
Business	50
Holiday	48
Courtesy Car	41

Served By:-

Served By	CountOfServed By
Mary Patterson	70
Jane Evans	68
Roger Thomas	48
Peter Bradley	27
Paul Dawson	26
Janet Newbold	9
Sam Swinton	1
Lee Robinson	1

The original ***Hirings Table*** is therefore significantly ***redundant.***

The next step is to extract each redundant field, reduce it to one instance of each entry, and create a new table to hold the ***normalised*** version. This will be done for each of the redundant fields, in turn. The normalised tables will then be related to the central table, which will be able to make the necessary lookups and retrieve the related data.

So far, in viewing data in the database, the Query has been a ***Select Query***, one which only displays the result and does no more than that with it. There are other types of queries, and we will use this working ***Select Query*** as the basis of a different type of Query, one which will automatically produce a new table of our choice.

It should be emphasised that whilst Select Queries make no alterations to the data, other queries can, and do. **NEVER RUN A QUERY OTHER THAN A SELECT QUERY UNTIL IT IS PERFORMING CORRECTLY.** In other words, make it work ***first*** as a select query, and ***then*** change its action.

In Access, creating a new Table automatically is done using a ***Make-Table query***. The columns shown on screen when the query runs, will become fields in the new Table. The ***Count*** field is not required, as it is not going to be one of the data fields in the to-be-created ***Vehicles Table.***

It should be noted here that each ***Vehicle*** has a corresponding ***Description***, and that the said description also recurs frequently, in time with the vehicle to which it refers. So far the ***Description*** field has not been mentioned. However, as the description of each vehicle only occurs once for that vehicle, we can, when creating the normalised ***Vehicles Table***, include a field for its description, thus reducing the incidences of ***Description*** to only one for each ***Vehicle***. If this sounds difficult, it isn't, as the steps below will demonstrate.

Firstly create a ***Select Query***, grouped to reduce the data to one instance of each ***Vehicle***, Test this before proceeding, and examine the result to ensure that each ***Vehicle*** exists only once in the result. However, as we plan to include the ***Description*** in the normalised end result, be sure to include that very field when the grouped query is being tested:-

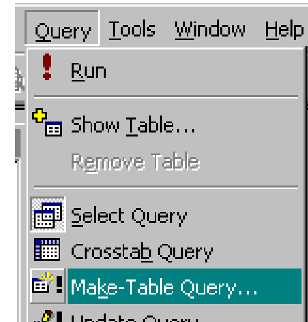
Field:	Vehicle	Description
Table:	Hirings Table	Hirings Table
Total:	Group By	Group By
Sort:		
Show:	☑	☑
Criteria:		

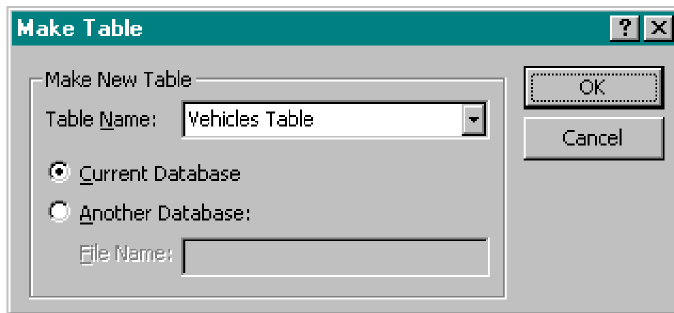
Vehicle	Description
A123ABC	1.6 Ford Fiesta GL, Red
B234BCD	2.0 Renault Laguna, Blue
C345CDE	1.0 Fiat Panda, White
D456DEF	3.0 Ford Explorer, Black
E567EFG	1.9 Fiat Tempra, Yellow
F678FGH	2.0 Vauxhall Cavalier, Grey

Here is the result of the query being run.

Notice that the ***Description*** is also normalised, as it matches its corresponding vehicle.

Now that is seen to be working, we can alter the nature of the query to be a ***Make-Table Query***, one which will automatically build the normalised table for us.





Change the query to a ***Make Table Query***. Access asks what name the new table will have; enter ***Vehicles Table*** and click OK. Now run the query.

There is no obvious action, no screen result, but actually a new, normalised ***Vehicles Table***, consisting of one record of data for each vehicle, has been created.

Here in Table View, are the contents of the freshly-created table. Notice how the Make-Table Query has done all the work for you:-

Microsoft Access - [Vehicles table : Table]

File Edit View Insert Format Records Tools

Close

Vehicle	Description
A123ABC	1.6 Ford Fiesta GL, Red
B234BCD	2.0 Renault Laguna, Blue
C345CDE	1.0 Fiat Panda, White
D456DEF	3.0 Ford Explorer, Black
E567EFG	1.9 Fiat Tempra, Yellow
F678FGH	2.0 Vauxhall Cavalier, Grey

This is all to the good, but there is one more step, which is to provide each record of the newly-created ***Vehicles Table*** - which now contains unique entries - with its own key field, in this case, an AutoNumber. Exit the query builder (it isn't necessary to save this query, as it won't be used again, unless you want to keep it for reference purposes) and go into Table Design for the ***Vehicles Table,*** and insert a new field, called ***Vehicle#***. Remember that key fields are terminated with the # (the hash sign) to aid us humans later on:-

The significance of the AutoNumber is that such a field will be automatically allocated the next logical, unused number when a record of data is added to the table. Thus, Access itself keeps track of the key field, and we don't have to worry about adding one manually. Click the key icon to ensure that ***Vehicle#*** will be the key field.

Field Name	Data Type
Vehicle#	AutoNumber
Vehicle	Text
Description	Text

Remember that as each record of the table needs a unique field, and very often, where no obvious key field is evident, the autonumber type of data neatly solves the problem of deciding which field is to be the key field.

Certainly the vehicle registration number itself could be used, but that means storing more data in the tables with which it is to be related, than is necessary. Adding an AutoNumber only uses 4 bytes (a long integer) per record, and therefore only occupies 4 bytes in any related record. If the actual registration number is used, it would be necessary to store that in any related tables – and this is inefficient. A four byte autonumber key field is better than a seven byte registration number. Also, remember that a vehicle can have different registration numbers, perhaps a personalised one; so that can't be used as a unique key field, as it may be re-issued, sold, or transferred to another quite different vehicle.

Instead, the new field will be called ***Vehicle#*** and be of a type called AutoNumber. An AutoNumber is an automatically incrementing counter which Access will add and increment for us, without any action on our part. This is a good example of why the # symbol is appended – it helps us humans spot this key field, later on.

Don't forget to click on this new field and make it the key field by clicking on the key icon. Save the table and in Table View, the data now looks like the panel opposite.

Microsoft Access - [Vehicles table : Table]

File Edit View Insert Format Records Tools Window

Close

Vehicle#	Vehicle	Description
1	A123ABC	1.6 Ford Fiesta GL, Red
2	B234BCD	2.0 Renault Laguna, Blue
3	C345CDE	1.0 Fiat Panda, White
4	D456DEF	3.0 Ford Explorer, Black
5	E567EFG	1.9 Fiat Tempra, Yellow
6	F678FGH	2.0 Vauxhall Cavalier, Grey
toNumber)		

In case you wondered why a modern PC database is restricted to 2 billion records – it's because on a PC, the 4 byte size of a Long Integer only allows storage of up to plus or minus 2 billion.

Now that the ***Vehicles Table*** is created, and the data for it normalised, repeat this process for the ***Hirers, Purpose*** and ***Served By*** fields, after which the ***normalisation*** process is complete, and five tables will exist:-

Hirings Table : Vehicles Table : Purpose Table : Served By Table

Now these ***normalised*** tables exist, some surgery will be performed on the original ***Hirings Table*** so that it can recognise the ***relationships*** and look up the necessary data.

Removing the redundant data

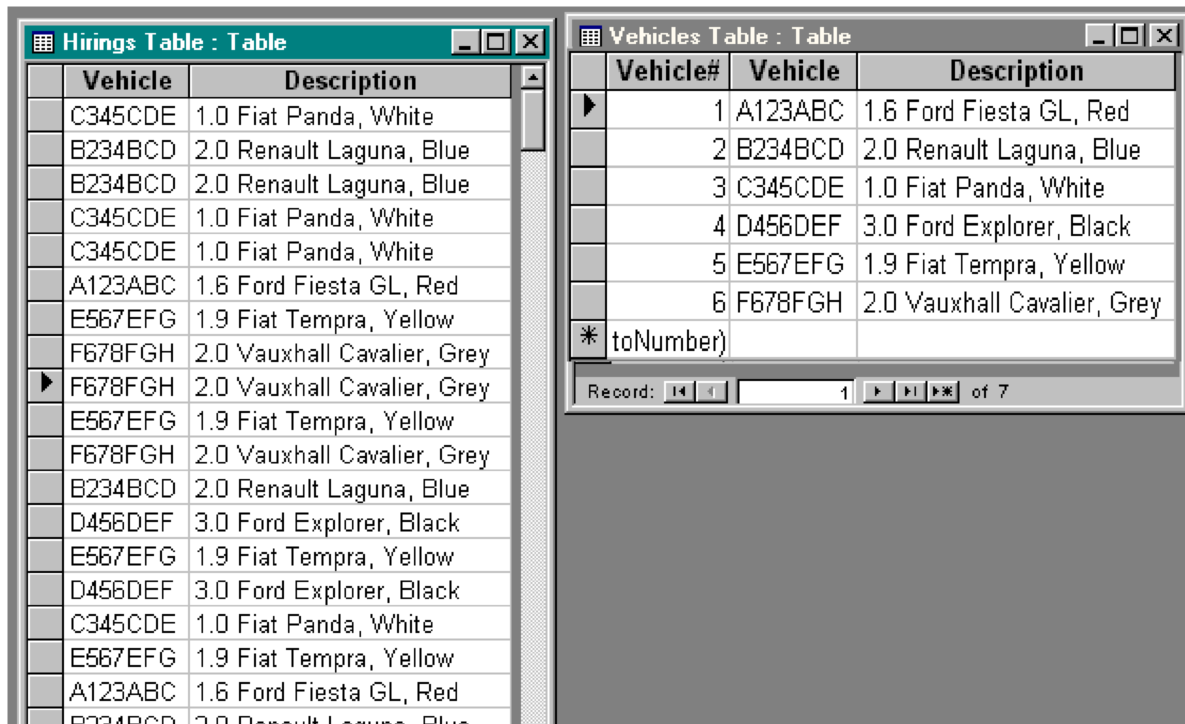
At the moment, data being held in the ***Hirings Table*** still contains the original redundant data, even though the normalised new tables now exist. The next step is to remove the duplicated data in the ***Hirings Table*** and replace it with the appropriate key field from the new normalised tables.

This can be done by hand if there is not a large amount of data, or automatically with a query if the amount of changes to be made is prohibitively large. It should be noted that changing data by hand is an extremely tedious process, and only viable if the amount of data is very small indeed.

To effect the changes by hand:-

Open the ***Hirings Table*** in normal view. Sort the data by the ***Vehicles*** field. Now reduce the size of the window in which the table fits, and shrink the other fields down to a minimum width - they won't be needed at the moment.

Open the ***Vehicles Table*** in normal view; resize this and position it alongside the other table so that both can be seen comfortably.



Hirings Table : Table

Vehicle	Description
C345CDE	1.0 Fiat Panda, White
B234BCD	2.0 Renault Laguna, Blue
B234BCD	2.0 Renault Laguna, Blue
C345CDE	1.0 Fiat Panda, White
C345CDE	1.0 Fiat Panda, White
A123ABC	1.6 Ford Fiesta GL, Red
E567EFG	1.9 Fiat Tempra, Yellow
F678FGH	2.0 Vauxhall Cavalier, Grey
F678FGH	2.0 Vauxhall Cavalier, Grey
E567EFG	1.9 Fiat Tempra, Yellow
F678FGH	2.0 Vauxhall Cavalier, Grey
B234BCD	2.0 Renault Laguna, Blue
D456DEF	3.0 Ford Explorer, Black
E567EFG	1.9 Fiat Tempra, Yellow
D456DEF	3.0 Ford Explorer, Black
C345CDE	1.0 Fiat Panda, White
E567EFG	1.9 Fiat Tempra, Yellow
A123ABC	1.6 Ford Fiesta GL, Red
B234BCD	2.0 Renault Laguna, Blue

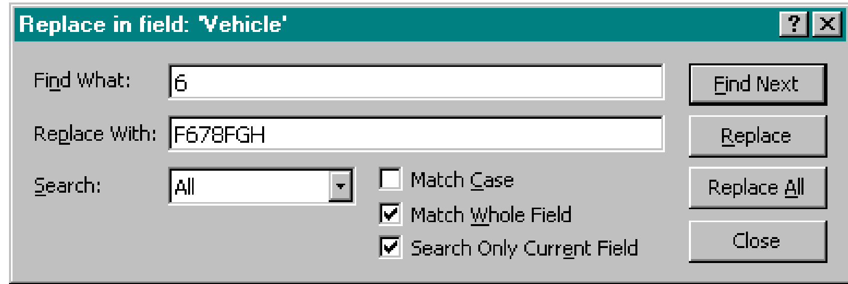
Vehicles Table : Table

Vehicle#	Vehicle	Description
1	A123ABC	1.6 Ford Fiesta GL, Red
2	B234BCD	2.0 Renault Laguna, Blue
3	C345CDE	1.0 Fiat Panda, White
4	D456DEF	3.0 Ford Explorer, Black
5	E567EFG	1.9 Fiat Tempra, Yellow
6	F678FGH	2.0 Vauxhall Cavalier, Grey
toNumber)		

Record: 1 of 7

The next step is easy, but tedious. Using ***Edit, Replace***, search for each unique occurrence of each record from the ***Vehicles Table*** as it occurs in the ***Hirings Table***, and replace it with the identifying key from the ***Vehicles Table***.

In the first instance, replace each *A123ABC* in the ***Hirings Table*** with 1.
On the 2nd instance, replace each *B234BCD* in the ***Hirings Table*** with 2.
On the 3rd instance, replace each *C345CDE* in the ***Hirings Table*** with 3.



... and so on, until all the records have been changed.

Here is the ***Hirings Table*** after this work has been done:-

	Vehicle	Description	Hirer	Purpose	Served By	Mileage
	1	1.6 Ford Fiesta GL, Red	Miss Pamela Jones	Holiday	Roger Thomas	9862
	1	1.6 Ford Fiesta GL, Red	Mr Paul Yorke	Shopping Trip	Roger Thomas	10325
	1	1.6 Ford Fiesta GL, Red	Mrs Alison Masters	Shopping Trip	Roger Thomas	12113
	1	1.6 Ford Fiesta GL, Red	Mr John Smith	Business	Roger Thomas	7094
	1	1.6 Ford Fiesta GL, Red	Mrs Rachel Dawson	Breakdown Cover	Roger Thomas	3394
	1	1.6 Ford Fiesta GL, Red	Mr Paul Yorke	Holiday	Roger Thomas	8215
	1	1.6 Ford Fiesta GL, Red	Henry Peters Ltd	Courtesy Car	Roger Thomas	5762
	1	1.6 Ford Fiesta GL, Red	Binster County Council	Business	Roger Thomas	1831
	1	1.6 Ford Fiesta GL, Red	Binster United Football Club	Shopping Trip	Mary Patterson	4278
▶	1	1.6 Ford Fiesta GL, Red	Miss Pamela Jones	Courtesy Car	Jane Evans	8244
	1	1.6 Ford Fiesta GL, Red	Binster United Football Club	Courtesy Car	Roger Thomas	3237
	1	1.6 Ford Fiesta GL, Red	Underhill & Co Ltd	Shopping Trip	Jane Evans	12026
	1	1.6 Ford Fiesta GL, Red	Lewis & Carr	Shopping Trip	Mary Patterson	4034
	1	1.6 Ford Fiesta GL, Red	Mr John Smith	Holiday	Sam Swinton	8261
	1	1.6 Ford Fiesta GL, Red	M. Guillaume Fenettra	Courtesy Car	Mary Patterson	7296
	1	1.6 Ford Fiesta GL, Red	Mr Paul Yorke	Holiday	Janet Newbold	7636
	1	1.6 Ford Fiesta GL, Red	Mr Paul Yorke	Breakdown Cover	Jane Evans	6715
	1	1.6 Ford Fiesta GL, Red	Underhill & Co Ltd	Breakdown Cover	Roger Thomas	5155
	1	1.6 Ford Fiesta GL, Red	M. Guillaume Fenettra	Courtesy Car	Peter Bradley	2495
	1	1.6 Ford Fiesta GL, Red	Underhill & Co Ltd	Holiday	Jane Evans	10102
	1	1.6 Ford Fiesta GL, Red	Underhill & Co Ltd	Holiday	Jane Evans	2597

Once all the contents of the ***Vehicle*** field have been changed to the correct key number from the ***Vehicles Table***, the penultimate step is to change the data type for the ***Vehicle*** field in the ***Hirings Table*** to a *Number* of *Long Integer* type. This is what Access needs to be able to establish the relationships and look up the data between the tables.

Repetitive Brain Injury

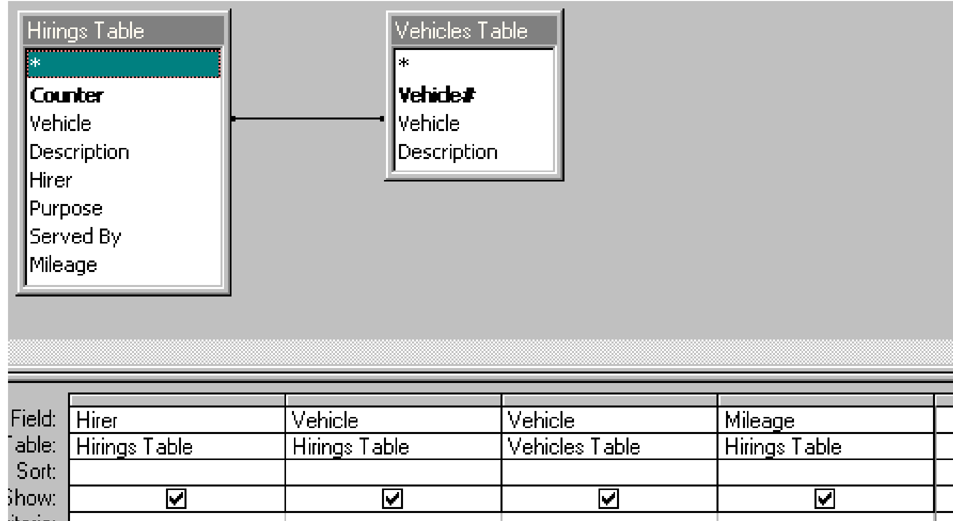
Anyone who has done this job by hand will have found out that if more than a few records of data are to be changed, the ugly head of Repetitive Brain Injury (or plain old-fashioned

boredom) raises itself, and the possibility of errors increases. Far better is the automated method, again using tools already available in the Access environment.

To do this automatically via query, it is necessary to establish an ***Update Query*** which will substitute the new ***Vehicle#*** for the old ***Vehicle*** in the ***Hirings Table.*** Remember that this should NEVER be done without establishing that the query works first of all as a ***Select Query***. Only when it is seen to be selecting the correct data is it changed to an ***Update Query***.

Start a new Query and add the ***Vehicles Table*** and the ***Hirings Table*** to the builder. Now drag a relationship line from ***Vehicle*** in the ***Vehicles Table*** to ***Vehicle*** in the ***Hirings Table***, as per the diagram opposite.

Add the fields as shown, and run the query.



This will force Access to show only the records from both tables where the linked data fields match; in this case, all of them.

Here is the result, showing only the first 21 records of data, when the ***Select Query*** is run.

Hirer	Hirings Table.Vehicle	Vehicles Table.Vehicle
Miss Pamela Jones	A123ABC	A123ABC
Mr Paul Yorke	A123ABC	A123ABC
Mrs Alison Masters	A123ABC	A123ABC
Mr John Smith	A123ABC	A123ABC
Mrs Rachel Dawson	A123ABC	A123ABC
Mr Paul Yorke	A123ABC	A123ABC
Henry Peters Ltd	A123ABC	A123ABC
Binster County Council	A123ABC	A123ABC
Binster United Football Club	A123ABC	A123ABC
Miss Pamela Jones	A123ABC	A123ABC
Binster United Football Club	A123ABC	A123ABC
Underhill & Co Ltd	A123ABC	A123ABC
Lewis & Carr	A123ABC	A123ABC
Mr John Smith	A123ABC	A123ABC
M. Guillaume Fenettra	A123ABC	A123ABC
Mr Paul Yorke	A123ABC	A123ABC
Mr Paul Yorke	A123ABC	A123ABC
Underhill & Co Ltd	A123ABC	A123ABC
M. Guillaume Fenettra	A123ABC	A123ABC
Underhill & Co Ltd	A123ABC	A123ABC
Underhill & Co Ltd	A123ABC	A123ABC

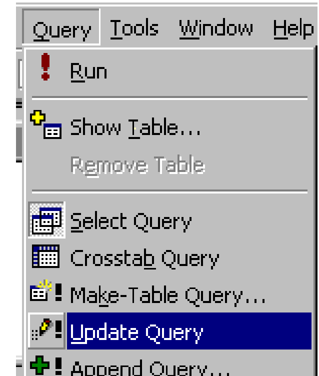
All 250 records of data should have been selected and be shown when the query is run on screen.

It doesn't look much different – but it is!

It ought to be noted here that Access, like most other database systems, follows a standard method of naming tables and their fields. In the above example, the same field name exists in two separate tables, and it would be confusing if they were to be both displayed without some indication of to which table they belong. Where a possible ambiguity exists, Access precedes each identically named field with the name of the table which "owns" it, followed by a dot. Where there is no possibility of confusion, Access just displays the field name on its own.

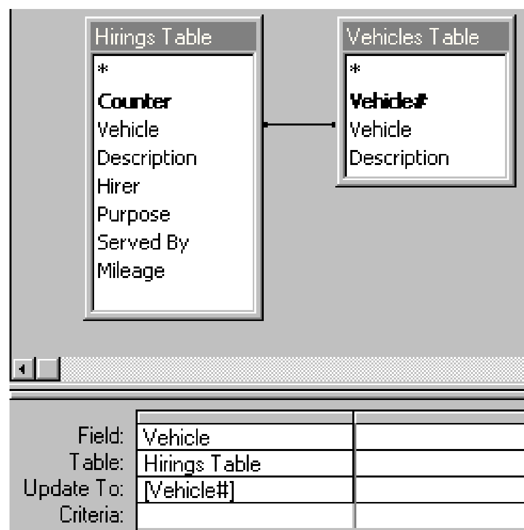
Having established that the Query as a ***Select Query*** is operating as expected, the next stage is to have it update the ***Vehicle*** in the ***Hirings Table*** to substitute the corresponding ***Vehicle#*** from ***Vehicles Table***.

Change the Query to an ***Update Query*** but don't run it yet :-



Now it is necessary to tell Access what data is required to be updated where, when the query is run. As soon as the query is changed to an Update Query, a new record appears in the builder, underneath "Table". This is where the new contents of that field are made apparent. To change the contents of ***Vehicle*** from the ***Hirings Table*** to the contents of ***Vehicle#*** in the ***Vehicles Table***, enter the field name in the small space of "Update To".

Ø Note that the field name MUST be enclosed in [square brackets] or Access will treat the entry as pure text. For example, if you just entered ***Vehicle#*** in the ***Update To*** grid, Access assumes that you want this piece of text "Vehicle#" inserting into the field, and will put quotes around it. Be absolutely certain that you have the field name which holds the data being inserted into the other field, within square brackets, as per the example below.



Vehicle	Description
5	1.9 Fiat Tempra, Yellow
6	2.0 Vauxhall Cavalier, Grey
2	2.0 Renault Laguna, Blue
4	3.0 Ford Explorer, Black
5	1.9 Fiat Tempra, Yellow
1	1.6 Ford Fiesta GL, Red
2	2.0 Renault Laguna, Blue
2	2.0 Renault Laguna, Blue
5	1.9 Fiat Tempra, Yellow
6	2.0 Vauxhall Cavalier, Grey
4	3.0 Ford Explorer, Black
5	1.9 Fiat Tempra, Yellow
2	2.0 Renault Laguna, Blue
2	2.0 Renault Laguna, Blue
1	1.6 Ford Fiesta GL, Red
1	1.6 Ford Fiesta GL, Red
6	2.0 Vauxhall Cavalier, Grey
2	2.0 Renault Laguna, Blue
3	1.0 Fiat Panda, White
2	2.0 Renault Laguna, Blue
6	2.0 Vauxhall Cavalier, Grey

When you are sure that this is correct, run the query, which will place the correct key field from ***Vehicles Table*** into the appropriate field from ***Hirings Table***. The query doesn't have any screen output when it runs, and will appear to have done nothing. However, if the data is examined afterwards in Table View, here is the result, again just the first few records.

Notice that the ***Update Query*** has worked, and has now substituted the key field (***Vehicle#***) from the normalised table (***Vehicles Table***) and placed it into the correct field (***Vehicle***) in the other (***Hirings***) table.

The Final Step…

The final step, whether the changes have been done by hand or by means of an Update Query, is to adjust the ***Hirings Table*** and change the data type for ***Vehicle*** from text to a Long Integer, and change its name from ***Vehicle*** to ***Vehicle#*** to show that it is now a related field, which will form a lookup between one table and the other. Also, its new name will match the name held in the corresponding ***Vehicles Table***, and be easier to spot later on; hence the # symbol being appended to it.

The provision of a field to hold this link, or lookup, in the ***Hirings*** – or any other – Table, is termed a ***Posted*** key field.

In design view for ***Hirings***:-

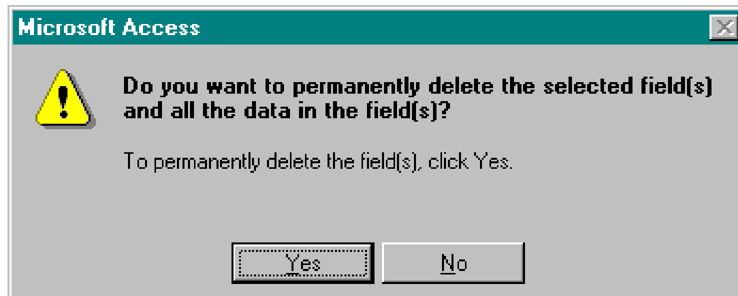
Field Name	Data Type
Counter	AutoNumber
Vehicle#	Number
Description	Text
Hirer	Text
Purpose	Text
Served By	Text
Mileage	Number

General | Lookup

Field Size	Long Integer
Format	Byte
Decimal Places	Integer
Input Mask	Long Integer

As the field for ***Description*** has been created as part of the ***Vehicles Table***, where each one occurs only once, this is no longer required by the ***Hirings Table***, and may be safely deleted from it.

Field Name	Data Type
Counter	AutoNumber
Vehicle#	Number
Description	Text
Hirer	Text



Click ***Yes*** and the redundant field of ***Description*** is deleted.

Save and exit the table. Access warns you that you may lose data as a result of this change; if you are in any doubt, check again very carefully that all the contents of the ***Vehicle*** field have been replaced by the appropriate ***Vehicle#*** number from the new ***Vehicles Table***. One good way to do this is to sort the field into order - there should be no text at all - it should all be numerical values corresponding to the vehicle table.

Having successfully saved the changed ***Hirings Table***, it will be able to look up 1 in the ***Vehicles Table***, and return a value of “A123ABC” with the appropriate description, and so on.

Other Types of Redundancy, and Assumptions

It may appear to the analyst in the first steps of learning the techniques of coping with redundancy that there may be occasions when a data field is left deliberately blank, or empty. This is never actually done in practice, and can be avoided as follows.

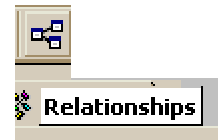
In an example of motor ***Vehicles*** and ***Owners***, it could be argued that a vehicle is ownerless, because it has only just been built, imported, or it has been scrapped. In such a case, surely the data field intended to hold a reference to the ***Owner*** would be empty, or null?

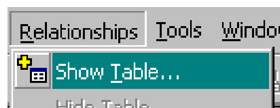
Not so! The inclusion of three extra "owners" called "Constructed" "Imported" and "Scrapped" will allow the data model to avoid any null associations of this sort. In fact, as all such relationships should have the Enforce Referential Integrity box checked – of which more very shortly - it is compulsory to have a valid entry in the ***Owner#*** field in the ***Vehicles Table***.

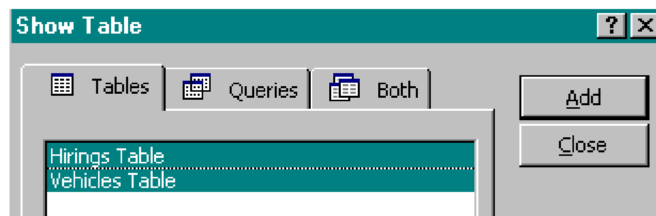
However, any such assumptions should always be documented, so that they and the reasons behind them, are clear.

Creating The Relationships

The very final step is to establish the relationship. Click on the ***Relationships*** icon (right) and then ***Relationships, Show Table*** (below).







Click ***Hirings Table***, ***Add***.

Repeat with ***Vehicles Table.***

Then click ***Close.***

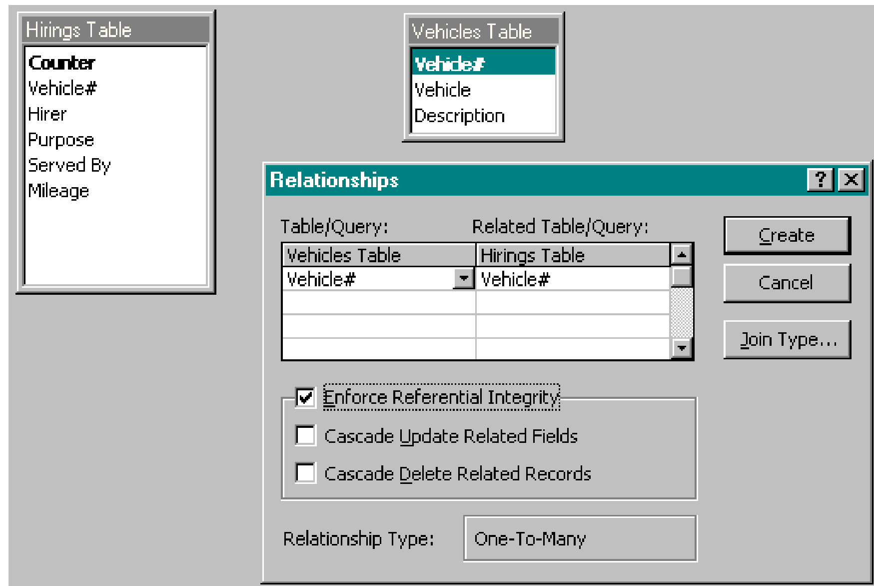
One-To-Many

The final objective is to associate one instance of ***Vehicle#*** in the ***Vehicles Table*** to many instances of ***Vehicle#*** in the ***Hirings Table.*** This relationship is called a "one-to-many" and forms the backbone of the relational database process.

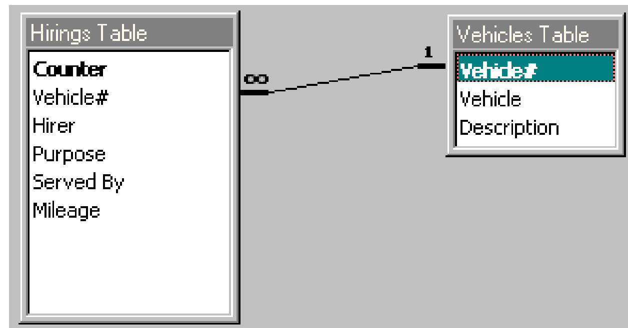
The technique here is to drag the data field from the ***one*** end of the relationship and drop it onto the correspondingly named data field on the ***many*** end. This is one reason why key fields end in a # symbol; this does allow us humans to pick them out very quickly, when presented with a list of them.

Drag ***Vehicle#*** from ***Vehicles Table*** and drop it on ***Vehicle#*** in the ***Hirings Table.***

Now in the Relationships editor, click on ***Enforce Relational Integrity***. This makes Access create a lookup relationship between the two tables, ensuring that the ***Vehicle#*** entered into the ***Vehicle#*** field in ***Hirings Table*** must already exist in the ***Vehicles Table***.



Finally, click ***Create***. Here is the finished screen:-



(If Access shows an error at this stage, the probable answer is that you performed a manual search-and-replace operation and did not do this correctly, or it was not thorough enough, and that there are ***Vehicle#*** numbers in the ***Hirings Table*** with no corresponding key or in the ***Vehicles Table***. Check the tables and correct these errors, or it is impossible to continue. It is also possible that the data type of the ***Vehicle#*** in ***Hirings Table*** was not changed to a Long Integer. These are two most common errors)

Note that Access shows an infinity symbol at the "many" end and a "1" at the "one end" and the result should be identical as the diagram above. If by mischance you have dragged the

relationship the wrong way, click on the connecting line to make it bold and then press d before trying again.

The ***Update Query*** method is better – and it can't make a mistake!

Now repeat the above steps for the **Hirer, Purpose,** *and* **Served By** fields, creating separate normalised tables for each of these fields, adding an autonumber, and replacing the appropriate field contents with the corresponding autonumber from the newly created tables.

When you have completed the normalisation, your ***Hirings Table*** should look like this (*opposite*), again with just the first few records displayed, and some columns not shown full width:-

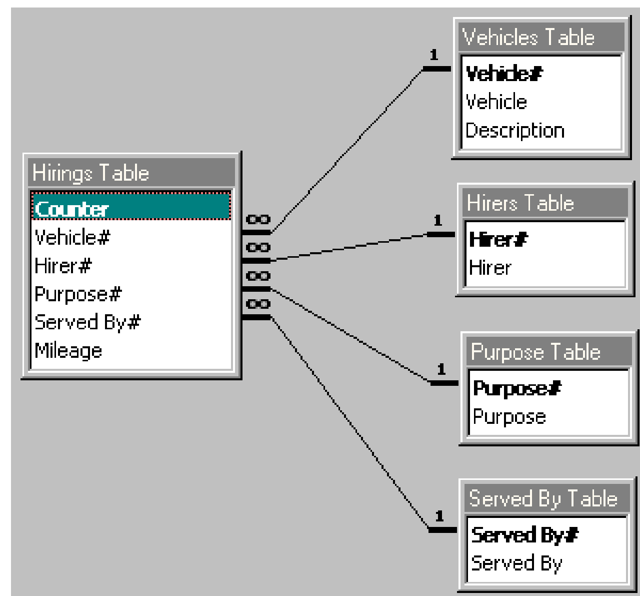
Vehicle#	Hirer	Purpose	Served By	Mileage
5	14	6	6	9391
6	14	1	1	11875
2	14	7	7	7313
4	4	5	5	10179
5	2	4	4	6590
1	8	7	7	9862
2	2	1	1	4954
2	19	1	1	4027
5	19	1	1	3226
6	17	1	1	1209
4	11	1	1	5311
5	19	5	5	686
2	19	1	1	5857
2	17	1	1	4820
1	18	1	1	2597
1	11	1	1	6715
6	6	1	1	8695
2	19	1	1	2563
3	7	4	4	3369
2	7	1	1	3983
6	6	4	4	5955

In fact, the only recognisable field which remains is the mileage! We will not be normalising this, as the mileage figures are unlikely to recur sufficiently frequently. Here is how the Table looks in design mode (below):-

Field Name	Data Type
Counter	AutoNumber
Vehicle#	Number
Hirer#	Number
Purpose#	Number
Served By#	Number
Mileage	Number

all these are of "Long Integer" type

The final step in the normalisation process is to create all the links, with the last stage looking like this:-



There are two other features available on the Relationships window which are worth explaining. The first is ***Cascade Update Related Fields***. If this is checked, Access will

automatically change the posted key field to match any changes made to the original key field. For example, if in ***Vehicles Table***, the key field is changed for some reason, under normal circumstances this would not be allowed if there were already some entries of that key field in the ***Hirings Table***. However, if this box is checked, Access will kindly update the records in the ***Hirings Table*** to reflect such changes.

Cascade Delete Related Records means that a record of data in the ***Vehicles Table*** which is deleted will automatically delete any records of data in ***Hirings Table***, if such are associated with that particular ***Vehicle#***. This is not normally allowed.

Whilst this is sometimes handy, use it with caution, because it can have far reaching consequences. Deleting a ***Vehicle*** which appears to be unused may result in unknown records in ***Hirings*** being deleted as well.

Many-to-Many : A First Taste

Without realising it, what has been done here is not so much a series of one-to-many relationships such as ***Hiring*** to ***Vehicle***, but a many-to-many relationships such as that which exists between ***Vehicle*** and ***Served By***.

So what's the big difference between a one-to-many relationship and a many-to-many relationship? The identification and use of many-to-many tables is really what sorts out the professionals from the amateurs in terms of data analysis and database design. Assigning a field in one table which can hold the key field from another table, and look up the data held there, is easy enough to do and is the first step in removing redundancy.

However, allowing one field in one table to look up many instances of another field in another table is not so easy. Thinking back to the Council Tax exercise, consider the relationship between a ***Payment*** and a ***Tax Payer***. Whilst the payer will make many payments, the payment itself will only ever refer to the one person who made it. This is a classic one:many scenario. One ***Tax Payer*** can make many ***Payments***.

Suppose, however, that for some reason it is necessary to allow many people to contribute to the Payment. Each identification of the contributing ***Tax Payer*** in the ***Payment Table*** must or course be recorded. But – how many contributing Tax Payers shall we allow? 5? 10? 50? The untrained analyst will do this:-

Payments Received Table

Field name	Data Type	Size	Examples
TaxPayer 1#	Long Integer	4	[Looked up in Tax Payer Table]
TaxPayer 2#	Long Integer	4	[Looked up in Tax Payer Table]
TaxPayer 3#	Long Integer	4	[Looked up in Tax Payer Table]
TaxPayer 4#	Long Integer	4	[Looked up in Tax Payer Table]
… and so on …			
TaxPayer 999#	Long Integer	4	[Looked up in Tax Payer Table]
Date Paid In	Date/Time	4	28-July-1999
Amount Paid	Currency	4	£123.45

This looks as if it will work, doesn't it? But it has serious problems. Most importantly, the number on contributing Tax Payers is fixed, and limited by the number of pre-built-in fields. If we allow, say, for 5 Tax Payers, and there happens to be more than that (a club maybe, where dozens of members share the expenses?) our database will fail.

Sure – we could build in 1,000 ***Tax Payer*** fields. Imagine, though, the hassle of creating and maintaining such a table. It would be impossible – and you can bet that one day, sooner or later, the Council will need 1,001 linked records – and again, our database will fail. Also, coding up this huge number of linked fields will use vast tracts of disk space, most of which will be unused, where a single ***Tax Payer*** has made the entire payment.

The ***Tax Payers*** table could hold thousands of fields, each linked to a payment record … but this is even more awkward, for exactly the same reasons. There has to be a better way.

There is – it involves creating an intermediate table "in between" the two tables which require a many:many relationship. This process is called decomposition, and when complete, allows an unlimited number of records in one end of the link to be related to an unlimited number of records at the other end of the link.

With this new concept in mind – and don't worry about it too much right now, because it is explained in far more detail, with diagrams, in the third chapter – think now of the car hire scenario. Some thought should show that a member of staff would almost certainly serve the came customer several times; and that the customer would be served by the same member of staff several times, during their associations with the company. Similar relationships exist between Hirer and Vehicle. The same vehicle would be likely to be hired more than once by the same hirer, especially if it's a regular customer or hired on corporate contract.

The unfortunate unskilled data analyst when confronted with the need to relate multiple Hirer records with multiple vehicle records will adopt the same problematic solution we looked at just now between ***Payments*** and ***Tax Payers***. He or she will create a series of fields in each table to try and accommodate a number of keys from the linked table, such as in the ***Hirers Table***, he will add ***Vehicle 1, Vehicle 2, Vehicle 3*** and so on. But his problem is just as serious. How many Vehicle 1 … Vehicle nn field should be added? How many times might a Hirer hire a vehicle; once, ten times, fifty times? Conversely, how many ***Hirer***# fields should be built into the ***Vehicles Table***? Once the tables are built, the construction of them is pretty much fixed.

The solution to this vexing problem is recognised when there is a need to relate one table many times to another table and vice versa. This is called decomposition and requires a "holding" table in between the two tables. The decomposed or holding table holds its own structure, with a one-to-many relationship with the other two tables.

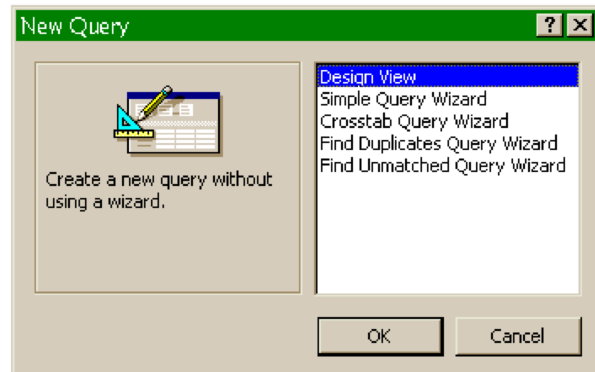
What was done with the exercise above was to create a decomposed table called the ***Hirings Table***. For the moment understand that the ***Hirer 1*** … ***Hirer 999*** solution is not the correct method and should never be adopted.

Don't get agitated if you feel that this many:many "decomposition" is pushing you out of your depth. You've already created exactly this, without realising it, and all that is important now is a recognition of the fact that they exist. Chapter 3 covers them, when we look at proper diagrammatic database design.

Lookups in action

Now that the relationships are in place, it is a matter of creating a query which performs the necessary lookups, and plugs in the related data.

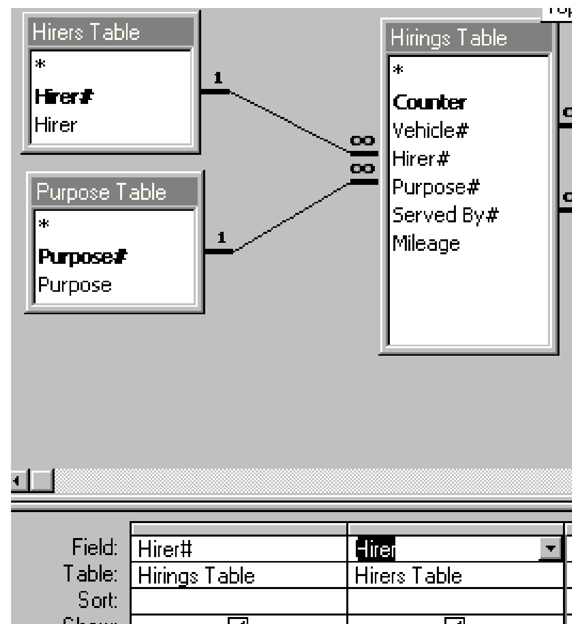
Click on ***Queries, New, Design View, OK***



Add the four related tables; it's a good idea to shuffle them around so that each is clearly seen.

It makes no difference in which order these are displayed on the screen. Having one or the other table first, or higher, is not significant in any way.

The way in which the lookups operate can be demonstrated by building a simple query which first of all just returns the ***Hirer*** details and subsequently adds the ***Vehicle*** details.



Drag ***Hirer#*** from the ***Hirings Table*** and drop it into the query builder Add ***Hirer*** from the ***Hirers Table***. Now do you see why adding the # symbols was a good idea? If you didn't bother, you'll soon wish you had!

Run the query.

Here is the result, showing just the first 20 records. See that the lookups are working, and that a value of *10* in the ***Hirings Table*** has correctly returned *Mr John Smith* from the ***Hirers Table***.

10 is of course stored many times in ***Hirings*** – but only once does *Mr John Smith* occur in ***Hirers***.

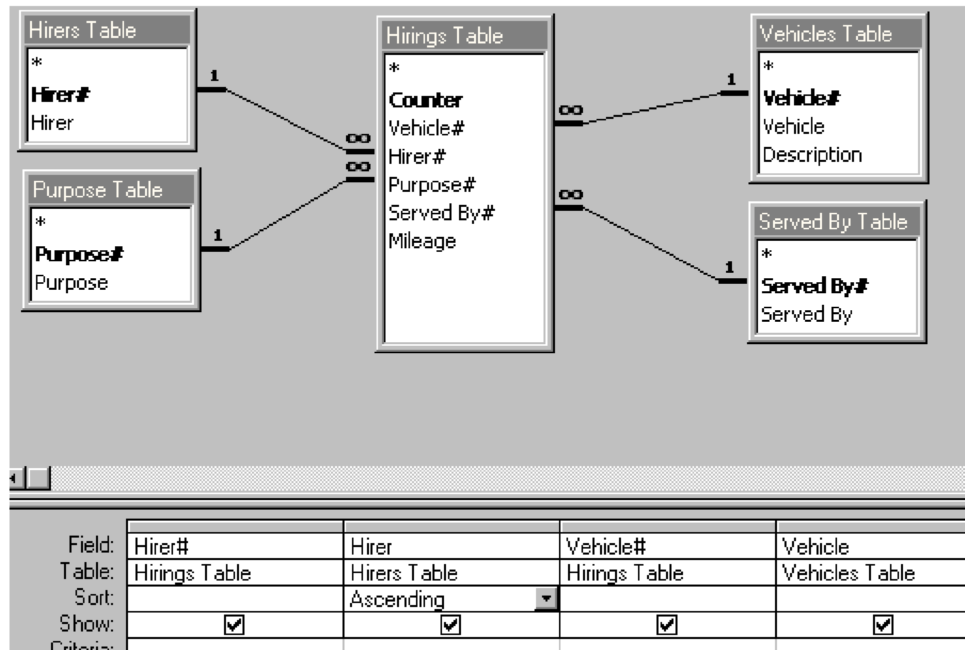
Normalisation is therefore working.

Return to query design.

Hirer#	Hirer
19	Wallace Thomas Ltd
11	Mr Paul Yorke
10	Mr John Smith
18	Underhill & Co Ltd
8	Miss Pamela Jones
10	Mr John Smith
10	Mr John Smith
8	Miss Pamela Jones
6	Lewis & Carr
11	Mr Paul Yorke
17	Rev James Roberts
14	Mrs Alison Masters
18	Underhill & Co Ltd
19	Wallace Thomas Ltd
6	Lewis & Carr
11	Mr Paul Yorke
14	Mrs Alison Masters
8	Miss Pamela Jones
19	Wallace Thomas Ltd
10	Mr John Smith
17	Rev James Roberts

Now drag ***Vehicle#*** from the ***Hirings Table*** and drop it on the next column. Add ***Vehicle*** and ***Description*** from the ***Vehicles Table.*** Sort the data as you wish. Run the Query by clicking on the shout (also known as the exclamation mark.)

Here is how it looks in the query builder, remembering that the placement of the tables on the builder's screen is not significant in any way:-



And here is the result of the query running:-

	Hirer#	Hirer	Vehicle#	Vehicle	Description
	1	Binster County Council	1	A123ABC	1.6 Ford Fiesta GL, Red
▶	2	Binster United Football Club	5	E567EFG	1.9 Fiat Tempra, Yellow
	2	Binster United Football Club	5	E567EFG	1.9 Fiat Tempra, Yellow
	2	Binster United Football Club	2	B234BCD	2.0 Renault Laguna, Blue
	2	Binster United Football Club	3	C345CDE	1.0 Fiat Panda, White
	2	Binster United Football Club	1	A123ABC	1.6 Ford Fiesta GL, Red
	2	Binster United Football Club	3	C345CDE	1.0 Fiat Panda, White
	2	Binster United Football Club	6	F678FGH	2.0 Vauxhall Cavalier, Grey
	2	Binster United Football Club	1	A123ABC	1.6 Ford Fiesta GL, Red
	2	Binster United Football Club	5	E567EFG	1.9 Fiat Tempra, Yellow
	2	Binster United Football Club	1	A123ABC	1.6 Ford Fiesta GL, Red
	2	Binster United Football Club	6	F678FGH	2.0 Vauxhall Cavalier, Grey
	2	Binster United Football Club	2	B234BCD	2.0 Renault Laguna, Blue
	2	Binster United Football Club	6	F678FGH	2.0 Vauxhall Cavalier, Grey
	2	Binster United Football Club	2	B234BCD	2.0 Renault Laguna, Blue
	2	Binster United Football Club	2	B234BCD	2.0 Renault Laguna, Blue
	2	Binster United Football Club	2	B234BCD	2.0 Renault Laguna, Blue
	2	Binster United Football Club	4	D456DEF	3.0 Ford Explorer, Black
	2	Binster United Football Club	2	B234BCD	2.0 Renault Laguna, Blue
	2	Binster United Football Club	4	D456DEF	3.0 Ford Explorer, Black
	2	Binster United Football Club	6	F678FGH	2.0 Vauxhall Cavalier, Grey

Access shows the ***Vehicle*** and ***Description*** for the related ***Vehicle#***, "plugging in" the desired data, each portion of which is stored only ***once*** in its ***related*** table.

To see the full version, drag the fields in this order onto the query builder. Note that "Mileage" is last on the right but is not displayed in full on the screenshot:-

Hirers Table: *, **Hirer#**, Hirer

Purpose Table: *, **Purpose#**, Purpose

Hirings Table: *, **Counter**, Vehicle#, Hirer#, Purpose#, Served By#, Mileage

Vehicles Table: *, **Vehicle#**, Vehicle, Description

Served By Table: *, **Served By#**, Served By

Field:	Hirer#	Hirer	Vehicle#	Vehicle	Description	Purpose#	Purpose	Served By#	Served By	Mil
Table:	Hirings Tat	Hirers Tabl	Hirings Tat	Vehicles Ta	Vehicles Tal	Hirings Tat	Purpose Ta	Hirings Tat	Served By T	Hir
Sort:		Ascending								
Show:	☑	☑	☑	☑	☑	☑	☑	☑	☑	
Criteria:										

… and run the query. Access returns the full list, with the lookups in place and plugged in. Note that for display purposes, some columns are not shown full width:-

Hirer#	Hirer	Vehicle#	Vehicle	Description	Purpose#	Purpose	Served By#	Served By	Mileage
1	Binster County	1	A123ABC	1.6 Ford Fiesta G	2	Business	7	Roger Thor	183
2	Binster United	5	E567EFG	1.9 Fiat Tempra,	5	Shopping Tr	4	Mary Patter	832
2	Binster United	5	E567EFG	1.9 Fiat Tempra,	5	Shopping Tr	4	Mary Patter	659
2	Binster United	2	B234BCD	2.0 Renault Lagu	4	Holiday	7	Roger Thor	977
2	Binster United	3	C345CDE	1.0 Fiat Panda, V	4	Holiday	7	Roger Thor	942
2	Binster United	1	A123ABC	1.6 Ford Fiesta G	4	Holiday	4	Mary Patter	830
2	Binster United	3	C345CDE	1.0 Fiat Panda, V	5	Shopping Tr	4	Mary Patter	214
2	Binster United	6	F678FGH	2.0 Vauxhall Cav	5	Shopping Tr	4	Mary Patter	279
2	Binster United	1	A123ABC	1.6 Ford Fiesta G	5	Shopping Tr	4	Mary Patter	427
2	Binster United	5	E567EFG	1.9 Fiat Tempra,	4	Holiday	4	Mary Patter	362
2	Binster United	1	A123ABC	1.6 Ford Fiesta G	3	Courtesy C	7	Roger Thor	323
2	Binster United	6	F678FGH	2.0 Vauxhall Cav	5	Shopping Tr	4	Mary Patter	844
2	Binster United	2	B234BCD	2.0 Renault Lagu	5	Shopping Tr	5	Paul Dawso	367
2	Binster United	6	F678FGH	2.0 Vauxhall Cav	3	Courtesy C	4	Mary Patter	514
2	Binster United	2	B234BCD	2.0 Renault Lagu	3	Courtesy C	1	Jane Evans	495
2	Binster United	2	B234BCD	2.0 Renault Lagu	2	Business	4	Mary Patter	650
2	Binster United	2	B234BCD	2.0 Renault Lagu	2	Business	4	Mary Patter	827
2	Binster United	4	D456DEF	3.0 Ford Explorer	5	Shopping Tr	4	Mary Patter	325
2	Binster United	2	B234BCD	2.0 Renault Lagu	2	Business	4	Mary Patter	961
2	Binster United	4	D456DEF	3.0 Ford Explorer	1	Breakdown	6	Peter Bradl	1187
2	Binster United	6	F678FGH	2.0 Vauxhall Cav	2	Business	5	Paul Dawso	69

Description	Purpose#	Purpos
Jumbo Jet	1	Breakdov
1.9 Fiat Tempra, Yellow	1	Breakdov
Jumbo Jet	1	Breakdov
3.0 Ford Explorer, Black	1	Breakdov
1.9 Fiat Tempra, Yellow	1	Bre
3.0 Ford Explorer, Black	1	Bre
2.0 Renault Laguna, Blue	1	B
Jumbo Jet	1	Bre
2.0 Renault Laguna, Blue	1	Breakdov
3.0 Ford Explorer, Black	1	Breakdov
2.0 Renault Laguna, Blue	1	Breakdov
3.0 Ford Explorer, Black	1	Breakdov
2.0 Renault Laguna, Blue	1	Breakdov
1.9 Fiat Tempra, Yellow	1	Breakdov
Jumbo Jet	1	Breakdov
1.9 Fiat Tempra, Yellow	1	Breakdov
2.0 Renault Laguna, Blue	1	Breakdov
2.0 Renault Laguna, Blue	1	Breakdov
2.0 Vauxhall Cavalier, Grey	1	Breakdov
1.6 Ford Fiesta GL, Red	1	Breakdov
3.0 Ford Explorer, Black	1	Breakdov
1.6 Ford Fiesta GL, Red	1	Breakdov
Jumbo Jet	1	Breakdov

altering this one

instantly changes all the rest

Now the normalisation, relationships and lookups are complete. The successful operation can be readily checked, by altering one of the related fields and seeing if this is immediately changed elsewhere.

For example, change any instance of ***1.0 Fiat Panda, White*** to ***Jumbo Jet*** and every occurrence of the former is instantly replaced with the latter as soon as the user moves away from the record of data just altered.

However... caution!

This is why it is extremely dangerous to allow users to actually see the queries which are being used to drive the database. Do not present end-users with a query on the screen for them to edit the database contents. It is easy to think that changing the data applies to just ***one*** record, when in fact the user may well be changing the information at the ***many*** end of the relationship, which is then applied instantly everywhere.

In the above example, the user may suppose that he is changing the vehicle for one car hiring record, when in fact he is changing the contents of the field in the ***Vehicles Table***. This being related to many records in the ***Hirings Table***, all the other occurrences are changed as well. This is likely to have catastrophic results, as key data may be overwritten.

The correct way to amend the car being hired is to change the relevant lookup key stored in the ***Hirings Table***, from one number to another. This forces the database to lookup the new related value in the ***Vehicles Table***, and does not amend the contents of the latter in any way.

That takes me back to my first point - that the relational database's ***power*** is also its ***weakness***.

Always use a Select Query to feed data to a Form or Report, and never as an action in its own right.

The QUERY should be considered to be "the engine room which the passenger never visits". Certainly the passenger can feel the engines at work - but the works are never actually seen.

Whilst Form Design is not within the scope of this book, I have built a working data entry form for the database, with which readers are invited to tinker and experiment - but please do make a backup copy of the original form!

Part 3 : Data Modelling and Database Designing

Avoiding major surgery by advance planning

The example used in Part 2, the table of vehicle hirings, demonstrates vividly how much work is required to correct design flaws in the initial data modelling. Redundancy had to be recognised, extra tables considered and created, and a great deal of editing of the first single table was then necessary before the relationships could be effected.

Errors of this nature usually result in a poorly planned database - or one which has had no planning at all - soon showing its shortcomings, which will worsen, perhaps fatally, as the database grows. The ***Hirings*** example given was not a very complex database, yet substantial corrective “surgery” was required. The amount of effort needed to resolve such problems in a large database can readily be imagined, and may be considered to be an exponential of the size of the database.

Also, expanding the original poorly-designed database is progressively more difficult, and the developer will spend more and more time maintaining the structure, at the expense of his other tasks. Operation is likely to suffer greatly, and the whole exercise is far better planned out long before fingers stray anywhere near a keyboard.

Very considerable amounts of work may be saved by avoiding such redundancy at the early, design and modelling stages. The data analyst's task is to remove the need for this surgery by correct data analysis, database design, and planning, with all tables, key fields and fields planned out on paper, or better still, on a large whiteboard. Only when the analyst has the complete data model crystal clear in his mind does he create the actual database, or hand the design over to the developers or programmers.

Many analysts are specialists in design and don't create the database themselves, acting in a consultancy capacity, planning and overseeing the design and modelling to ensure that the programmers, who are themselves specialists in their own field, construct the desired result.

The analyst's primary tool is the ***ERD*** or ***Entity-Relationship Diagram***, which is a visual overview of the data model and which shows all the vital component parts of the database. Constructing the ERD clarifies the relationships in the analyst's mind, and provides the foundation of the end product.

Entity-Relationship Diagrams

The technique of data modelling removes the redundant data and presents the programmer with the **Entity-Relationship Diagram** (ERD) which shows the various entities and the degree of relationship between them. The ERD is a high level abstract view, not intended to show fields. Semantics play an important role in constructing the ERD as all parties involved must be quite clear on nomenclature and the nature of the association between linked entities.

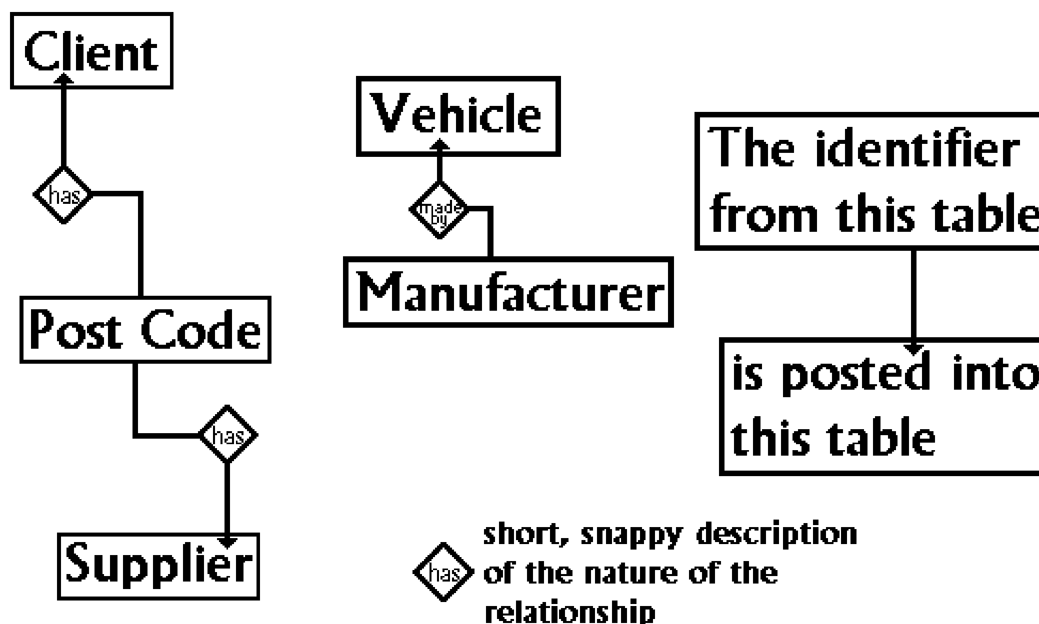
On seeing the growing or completed ERD, it is common for those untrained in data analysis techniques to believe that far too many tables will be created. Correct explanation, however, will clarify the need for these.

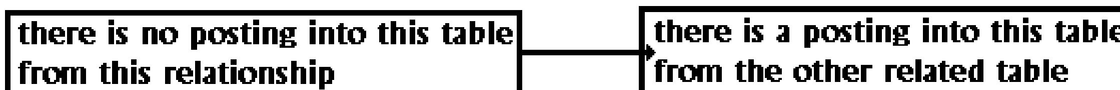
At this stage the data modelling exercise is quite independent of any specific hardware platform or software application, and may be implemented in any system; a hand coded program, an existing in-house application, or a relational database package.

An Entity-Relationship Diagram (ERD) explained

In data analysis methodology, an entity is drawn in a rectangle. A relationship with another entity is shown with a connecting line between the two rectangles, qualified by a diamond which explains the relationship, in simple plain English format.

In the example below, notice how one single table of Post Codes is related to more than one other table – thus providing centralised information, shared out rather than duplicated. Working along the same lines, tables of Manufacturers and Suppliers could also be linked into one dealing with Purchase Orders.





It should be stressed that at this design stage, the analyst is not concerned with the minutiae of the planned database. Only the main component parts (the entities), and the data fields which form the links and relationships between the entities, are the subject of the ERD methodology. When the data modelling is complete, and the entities are drawn up in skeletal form, does the analyst concern himself with the fine points of each entity.

In effect, at this stage, a very abstract or high-level view should be adopted; stand back from the details of the database, and be concerned only with the main structures and how they connect.

Having drawn the first two entities in their rectangles, the question is asked:-

"Does Entity A have a relationship with Entity B?" in other words, *"Do we want to look up any of the fields in Entity B from Entity A?"*

and then

"If it does, how many times can it be related?" in other words*, "Zero times, just once, or many times?"*

If the answer to the first question is "Yes", then **Entity A** does indeed have a relationship with Entity B, and a line is drawn connecting the two rectangles. A diamond is added along the line, with a simple expression of the nature of the relationship. Keep these short and snappy; it may take several attempts to find a word which exactly suits the nature of the association.

(If no reason to perform lookups is found, the two entities are not related, no connection is drawn, and the second question is not required.)

If the answer to the second question is "Zero" then there will be no embedded arrow at the **A** end.

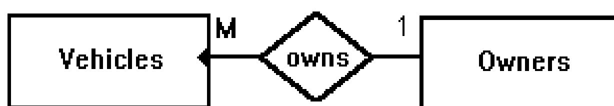
However, should the answer to the second question be other than "Zero", then **Entity A** has a compulsory link with **Entity B**, the line at the **A** end has an embedded arrow, indicating that it is "plugged into" the B end, and that there will be a field in the Entity A table structure to hold it. As there is no need to create a field in **Entity B** to hold a value (or lookup) from **Entity A**, there is no embedded arrow at the **A** end.

If the answer is "Many times" – which will usually be the case – the symbol **M** is written alongside the embedded arrow. This shows that **Entity A** will look up many records of data in **Entity B.** If any record in **Entity A** is related to just one record (and this is rare) in **Entity B**, a **1** is shown alongside the **B** end.

In other words, any data table which looks up data in another table will require the key field from that table as one of its own fields. Such is termed a ***posted*** key field***.***

Now repeat the two question in the other direction, i.e. from **B** to **A.** Does Entity B look up anything in Entity A?

Taking the previous example of a database of Owners and Vehicles, the ERD would look like this:-



This ERD demonstrates that:-

"A Vehicle must have exactly one owner; an Owner doesn't have to own a vehicle, but if he does, he can own many of them."

This diagrammatic representation of the two tables shows that there will be a field in the ***Vehicles Table*** which will hold the key field from the ***Owners Table***, but not vice versa; there will be no field in the Owners Table holding a key from the Vehicles Table. This is classic one-to-many relationship.

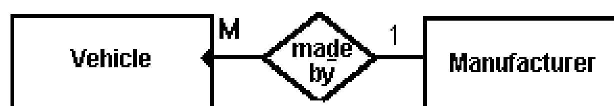
By a "zero" relationship, we mean that an Owner may exist in the ***Owners Table***, but not currently be associated with any vehicles. For example, someone who has sold his only vehicle and has not yet replaced it; or perhaps a deceased Owner. Considerable thought on the "what if?" basis forms an important part of data analysis.

What if, then, a vehicle comes to the end of its life, and is scrapped – or a new vehicle, freshly manufactured or imported, does not yet have an owner? In this simple example, it is **assumed** that there are ***Owners*** called "New" and "Scrapped" and that there is not a requirement to track changes of ownership.

It should be apparent that this example could readily have relationships with other tables such as Manufacturers, Colours, Engine Types, Tax Class, etc.

Another example of a many-to-one relationship

A table of vehicles will undoubtedly contain the name of the manufacturer, and information relevant to that manufacturer such as address, post code, contact details and so on. To store these repeatedly in the ***Vehicles Table*** would incur massive redundancy, and so the Manufacturers Table will be created to hold these details once, with a lookup from the ***Vehicles Table***. Here is the next stage of the ERD:-



This data model demonstrates that "A ***Vehicle*** must be made by exactly one ***Manufacturer;*** a ***Manufacturer*** doesn't have to make any ***Vehicles***, but if he does, he can make many of them." There will be a data field in the ***Vehicles Table*** which will hold the key field from the ***Manufacturers Table***. This key field will probably be ***Manufacturer#***.

Many-to-many relationships

The non-data-analyst is frequently confounded by the need to relate entities, not on a one-to-many basis, but a many-to-many basis. This always requires an intermediate table and is referred to as ***decomposition***. The technique is illustrated by taking the above example of Owners and Vehicles and adding the requirement to track changes of ownership.

The untrained database developer will either:-

Include in the fields for ***Vehicle***, a fixed number of ***Owner*** fields such as *Owner1, Owner2, Owner3* etc., or
Include in the fields for ***Owners***, a fixed number of ***Vehicle*** fields such as *Vehicle1, Vehicle2, Vehicle3* etc.

Neither system will work, for the simple reason that the number of changes of ownership is fixed by the number of fields pre-built-into the table. Once these are exhausted, no further updates are possible. Also, in an ***Owners*** record with many *Vehicle1 ... Vehicle5* fields, but only one ***Vehicle*** ever owned, significant wasted, spare and unused space occurs in the data table.

This is termed a "many-to-many" relationship. "A vehicle may have many owners; an owner may own many vehicles." It might even be stated that "A vehicle may be owned more than once by the same owner."

Another example might be that of a database required to associate students with lecturers. A lecturer will probably deliver many courses; a student will probably attend several different courses. The untrained database developer will create a ***Students Table*** with provision for attending a fixed number of courses such as Course1, Course2, Course3, &etc. The ***Lecturers Table*** will have Course1, Course2, Course3, &etc. In both cases, as soon as the number of courses to be stored exceeds the number of data fields to hold them, the database will fail.

Of course it would be possible to create a ***Students Table*** with Course1 ... Course500, and have provision for a far greater storage of attended courses than will be needed. This method has two significant disadvantages. Firstly, it is very costly to code the database and create the data entry screens when up to 500 courses may be stored. Also, since most students will never attend this vast number of courses, huge amounts of disk storage space are wasted storing empty spaces which are never likely to be used.

Then, of course, there is bound to be the student who attends his 501st course; or the lecturer who delivers his 501st course... at which point the database fails again. The Course1 ... Course500 method is far too inflexible and inefficient for serious use, and a better way of expressing "many-to-many" relationships must be explored.

The correct solution is a three table model, with a new entity created in between the two ends of the many-to-many relationship. This new entity is called a ***decomposition***, and allows an ***unlimited number*** of one end of the relationship to be associated with an ***unlimited number*** of the other end of the relationship. Returning to the problem of Vehicles and Owners, here is the three table ***decomposed*** solution:-

Owners Table			Owned Vehicles Table				Vehicles Table	
Owner#	**Owner**		**Owned Vehicle#**	**Owner#**	**Vehicle#**		**Vehicle#**	**Vehicle**
1	Charlwood		1	1	1		1	A123ABC
2	Fox		2	4	1		2	B234BCD
3	Bryant		3	5	3		3	C345CDE
4	Kennard		4	5	1		4	D456DEF
5	Gibson		5	4	4		5	E567EFG
6	Pickard		6	2	3		6	F678FGH
			7	1	6			
			8	3	2			
			9	4	1			
			10	5	1			

To ensure that the mechanics of this are understood before progressing further, from the above list of data with the three tables, answer the following questions:-

1 Which vehicles has Bryant owned?[1]
2 Who has owned F678FGH?[2]
3 Has any owner never owned a vehicle?[3]
4 Who were the first and last owners of C345CDE?[4]
5 Are any vehicles still in the hands of the original owner?[5]

From the above chart it is apparent that some owners have owned several cars and that some cars have been owned more than once by the same owner. This allows unlimited changes of ownership, and therefore unlimited expansion of the database - a crucial concept of database design. The model above also supports tracking of changes of ownership, so the history of a vehicle can be identified; it would be entirely possible for an owner to own the same vehicle more than once, perhaps in the case of a classic or collector's vehicle.

Owner 1 (Charlwood) has owned two cars (Vehicles 1 and 6), but he sold Vehicle 1 to Owner 4 (Kennard) who then sold it to Owner 5 (Gibson) who sold it back to Owner 4 (Kennard again) and who then sold it back again to Owner 5 (Gibson again).

Here, it has been assumed that the requirement is to **track changes of ownership**, and not simply to record who owns the vehicle right now. This is known as an audit trail. There is an important difference in the data modelling between these two requirements. If there was no need to record changes of ownership, the ***Owned Vehicle Table*** could have had a joint key field of a combined ***Owner#*** and ***Vehicle#***. This still allows an unlimited number of data records, but with the important exception that the same owner cannot own the same vehicle twice. In effect, the ability to record an audit trail - a record of how a vehicle has changed hands - is lost.

If this audit trail is necessary, the joint key field of ***Owner#*** and ***Vehicle#*** will not do. Perhaps the addition of the date of the sale, as a third joint key field, would do the trick? Yes, until the

[1] Vehicle #2 (B234BCD)
[2] Owner #1 (Charlwood)
[3] Yes; owner #6 (Pickard)
[4] First was owner #5 (Gibson); last was owner #2 (Fox)
[5] Yes, three; vehicle #2 (B234BCD) is still owned by owner #3 (Bryant); vehicle #4 (D456DEF) is still owned by owner #4 (Kennard) and vehicle #6 (F678FGH) is still owned by owner #1 (Charlwood)

same owner sold and bought the same car on the same day; perhaps at an auction? In this latter case, the database will fail, and the idea is to produce a data model and working database which will operate for a long time, and stand some expansion at a later date.

A better solution, and one which does support the audit trail, is a new key field for each change of ownership. Now the modelling allows a true number of unlimited owners and vehicles.

Here is the ERD for the decomposed entities:-

This ERD demonstrates that:-

"A ***Vehicle*** doesn't have to be an ***Owned Vehicle***, but if it is, it can have many Owners during its life. An ***Owner*** doesn't have to own any ***Owned Vehicle***, but if he does, he can own many of them. An ***Owned Vehicle*** is exactly one ***Vehicle*** which at any given date is owned by exactly one ***Owner.***"

Given that a suitable key field is found for the decomposed ***Owned Vehicle Table*** - and this is an excellent candidate for the autonumber - such a model also permits the same ***Owner*** to own the same ***Vehicle*** many times – perhaps the case with a rare or classic car, which may be repeatedly bought and sold between dealers or collectors. This structure allows unlimited numbers of changes of ownership, solving the problem of tracking changes of ownership.

If a ***Vehicle*** has been scrapped, it will occur in the ***Owned Vehicles Table*** as being related to an ***Owner*** called "Scrapped". Likewise, there will be an owner called "New" to represent a freshly constructed or imported vehicle, which has yet to be registered.

The problem with decomposed tables is choosing the correct key field. Sometimes a joint key field will suffice. In the vehicles and owners database, it would be advantageous to allow the storing of all changes of ownership, so that a vehicle's history may be tracked. If the joint key field ***Owner#, Registration Number*** is chosen, this will prevent the same owner from owning the same vehicle more than once. This is fine until one owner re-purchases a vehicle previously owned by him, at which point the database will fail.

In this scenario, a better alternative is to create an autonumber for the decomposed table, which, being unique, will permit the analysis described above. In another scenario, it may not be necessary or desirable to allow this sophistication, and a joint key field will be acceptable. Discussion with the end-users will clarify the situation.

Many-To-Many : A Vital Concept

It is extremely important that the budding analyst and database designer understands the necessity to decompose many:many relationships with an intermediate table. In order to provide some self-test scenarios, the following exercises should be undertaken.

1. Replan the DVD Shop database to support many actors / actresses with many films, remembering that “a film has many actors / actresses, and an actor/actress will be in many films.”
2. Take the scenario of a Hire Car Company. What many:many relationships might exist? Remember that a customer may well hire the same car several times, perhaps even in the same day! Would it be sensible to track a hire car’s history, i.e. maintain an audit trail of which customer has hired it? Perhaps a vehicle’s service history would benefit from recording which mechanics serviced and maintained it, so that if a problem arises, the mechanic can be identified?
3. Think about how a database for a Theatre might be organised. A Production will have many Performances, and a Performance will have many Seats, each of which may be booked or reserved by a member of the Audience, who might well wish to see the show again later in its production run. Would it be necessary to track who has occupied each seat?
4. A classic problem is that of Hospital Beds. Each of these may in its service life be located in many Wards, and be occupied by many Patients. Each bed requires regular maintenance and steam cleaning (to kill any possible infection). Would it be likely that any Bed might be occupied by the same Patient on the same Ward, more than once? Is there a need to track who has occupied a Bed, and on which Ward it was at the time?

One-To-One Relationships

In the course of drawing up the ERD it is entirely possible that analysis will show that two tables which, at first look, appear to be subject to a 1:M relationship are, in fact, related on a 1:1 basis. One-to-one relationships are rare in database design and if such occurs it should be subjected to intense scrutiny. If, after further analysis, a genuine 1:1 model is produced, then the answer is simply to merge the two tables concerned.

Null, empty or blank data fields

What should happen if the content of a data field is consistently empty, or empty in a significant number of cases? If the analyst's credo is to be taken at face value, storing empty spaces is an anathema! The *occasional* empty field is permissible and in some cases, unavoidable. Take, for example, a database of employees, where the analyst had created space to record the employee's reason for leaving, and information relevant to where they are moving jobs – perhaps the name of the next employer.

In many cases, this data will be blank, where employees are still currently employed. It may be assumed that *eventually* these fields will contain data, if only to record an employee's decease. However, it could be argued that significant unused space will occur, and in a large corporate employee database, this could amount to a tidy sum in disk space.

It is suggested that common sense be applied in such a situation. Use the database's own grouping and counting tools to identify the percentage of empty data fields. If this reaches a figure higher than 15 per cent, it is better policy to remove the empty fields from the Employees Table and create a relationship with the removed contents:-

EMPLOYEES TABLE

Employee#	Title	Forename	Surname	[other data]
001	Mr	Joe	Soap	Address etc
002	Mrs	Joan	Soap	Address etc
003	Miss	Anita	Parsons	Address etc
004	Mrs	Leah	Brewer	Address etc
005	Miss	Marcia	O'Leary	Address etc

REASONS FOR LEAVING TABLE

Reason#	Reason for Leaving
001	Retirement
002	Early retirement
003	Redundancy
004	Dismissed
005	Unknown
006	Higher salary
007	Better prospects
008	Deceased

EMPLOYEE LEAVING TABLE

Leaving#	Employee#	Reason#	New Place of Work	Date
001	003	006	Jones & Sons Ltd	11/02/2016
002	004	001	Retired	11/02/2016
003	005	004	Unknown	11/02/2016

Here, the use of an AutoNumber as the key field should be noted. Why has the Employee# not been used instead? This would work fine until an employee left the company and was then re-employed, perhaps to leave again … and the use of Employee# must be unique.

The above strategy eliminates null or empty data fields in the **Employees Table**, but obviously requires more work to implement the extra tables to hold the changed data.

Skeleton Tables

Once the data modelling stage of the ERD is complete – and this is likely to involve many changes to the initial data model - the next step, the **Skeleton Tables**, show the table names, key fields, and key fields which have been linked to other tables. These, formerly called foreign key fields, are now called **Posted** key fields, and form the links or relationships between the tables.

TABLE NAME (key field, *posted key field*, ...) the ... part means that the need for extra fields is recognised, but these are outside the scope of the initial data model.

Skeleton Tables again do not show every last field and are not intended to do so; but they do show the full data model and its relationships.

The skeleton tables for the decomposed ERD are:-

VEHICLES (Registration Number, ...)
OWNERS (Owner#, ...)
OWNED VEHICLE (Change#, *Owner#*, *Registration Number*, ...)

Checking the ERD and Skeleton Tables

The drawing up of the skeleton tables is easily checked against the ERD by the following method:-

1. Count the entities on the ERD, these should match the number of table names.
2. This figure should also match the number of key fields, with the exception that if a (probably decomposed) table is created which uses a twin (joint) key field rather than a single key field, one is subtracted from the total.
3. Count the links with an arrow at the end, these should match the number of posted key fields.

Once the ERD is constructed, the developer will have a very much clearer picture of what component parts the database consists. The ERD is also readily explained to the end-users of the database, who are often able to suggest improvements in nomenclature, drawn from their practical experience of dealing with the real data. The analyst should be prepared to repeatedly edit and redraw the ERD in response to comment and input from the end-users, until the picture of the data, its construction and component parts are all quite clear in everyone's mind.

Time spent at the analysis and design stage pays off handsomely once the database is being developed, as many errors will have been eliminated in advance. It is fatal to try and construct a relational database without considerable attention to the modelling, as errors and omissions which come to light once the software has gone into service will be expensive in time and cost to correct.

Creating the full data model

Add in the various fields which comprise the individual fields in each table. Considerable liaison with the end-users of the database will be required, to ensure that the table structure stores what they want. It is also useful to check with end-users and ascertain typical queries with which the database is expected to cope.

It is also more than possible that alterations to what was though to be the final model of the ERD will be necessary after development has started; consult with the end-users where clarification is required. Extra redundancy may be apparent, and this must be eliminated before the database can be put into service. The process of finalising the data model is nearly always a circular one, requiring many referrals to the end-users in order to clarify the fine points of the proposed database and to answer the many “well, what if *this* happens?” type questions.

Never regard the first ERD as the final stage – it will usually require many redraws and adjustments. A whiteboard will pay dividends!

Summary

One hour of planning saves **ten** hours of work later on. This is probably the most useful 1:M relationship the analyst can put into effect!

A **large whiteboard** is a most valuable design tool.

Talk to the end-users and about what data is required to store. **Look at examples of the data**; take a sample of any card index or other existing system. Make sure the purpose and end result of the database is clear. Ask the users what they want the system to do, and what they think ought to be on a "wish list" of functionality. Be prepared to talk to users several times, as ideas form.

Whilst talking to the users, **collect all the snippets of information to be stored**, which will become the data fields. **Group these into Entities** and identify any field which occurs in more than one table; place these into **normalised** tables of their own. Decide on a **key field for each entity**, using any existing nomenclature. Again, be ready for changes to the data model as it develops; more contact with the end users will probably be necessary.

Plan the database on paper or whiteboard using the ERD (Entity-Relationship Diagram) method, before any work is done of the actual keyboarding. It is common for the data modelling to be continually changed as ideas are tried out, adopted or rejected. That is why the planning stage is so crucial.

Draw up the **full ERD**, showing all the entities and their associations, and then **compose the Skeleton Tables**, checking these against the ERD. Eliminate any resulting errors.

Don't go near a keyboard until you have a rock solid foundation for the data, and have "the target at which to shoot".

Compose the Full Tables and implement them in the database application, creating the relationships.

Enter some test data and establish the results against what is known to be correct, particularly that the lookups are functioning correctly and as expected, especially if any calculated fields are being used.

Even at this stage, more **discussion with the end users** and a revision of the ERD is likely.

Continue this process until the model is complete. Design and implement the queries, data entry screens, and reports.

About the Author

Rob Davis was born in 1954 and after various unsatisfying jobs, bought a home computer – an Oric Atmos – in 1983. This was an immediate success and he learned programming from the ground up, writing many programs for domestic use before graduating to an Amstrad PC1512 in January 1987. At this time, very few private individuals owned IBM-compatible personal computers.

Interpreted GW-BASIC was soon replaced by compiled QuickBASIC and by this time, producing commercial software, some very comprehensive bespoke multi-user relational databases were written in this language. The change to the Visual environment was traumatic, and it was some time before Visual Basic and Access found favour.

During a year's "time out" to take a full-time Master's Degree in Information Technology, Rob found that the methods taught for database design rang an immediate bell. Having refined the Howe method taught by Nigel Roberts, and by then involved with teaching students himself, he found that students being shown basic data analysis techniques lacked a book at "Beginner" level, Consequently he decided that a multitude of student notes on the subject could be brought together and composed into a book.

Whilst working as a Database Consultant and Programmer, Rob tutored very popular evening classes in Access, formal Database Design, and Visual Basic. Many IT lecturers have little or no experience of actually writing industrially useful operational software, and can only teach the mechanics of the application, without being able to demonstrate its use a problem solving tool. Rob is very aware of the skills and techniques, as well as the problems, which a professional database developer or programmer will face up at the sharp end of the software industry, and constantly injects real life situations into his classes.

Rob is now working as IT Manager for a construction company in the West Midlands. Somehow he also finds time to deal with a substantial amount of email on his many interests, notably RAF Bomber Command 1939-45, Campanology, Motorcycling (he owns a classic 1981 Honda CX500A as well as a newer Honda NT650 Deauville), teach his piano-accordion new tunes, and shoot the English Longbow. He is married to Sandy and has three adult stepdaughters.

Email rob.davis@blueyonder.co.uk Internet http://www.robdavis.webhop.org

INDEX OF CONTENTS

A

Age, 15
Attributes, 9
  use of [square brackets], 39
Audit trail, 58
Author, 66

B

Beginners' Common Mistakes, 16
Beginners' Common Mistakes, 13

C

Card index, 8
Cascade Delete Related Records, 45
Cascade Update Related Fields, 44
Copyright, 2

D

Data
  Grouping and counting, 31
Data analysis
  reasons for, 8
Database
  Creating the full model, 4, 64
  data collection, 4, 8
  end users, 8
  Expanding, 53
  Modelling and Designing, 53
  Repairing design mistakes, 29
  Three R's of Design, 22
Disk space
  saving, 16

E

Email / URL, 2
End-users, 17
Enforce Relational Integrity, 43
Entities, 8, 9
ERD (Entity-Relationship Diagram), 53
  Checking, 4, 63
  diagram example of M
    1, 55
  Diagram of M
    M solution, 59
  Many-to-many relationships, 57
  Method explained, 4, 54
  One-to-one, 4, 61

F

Fields, 8, 9

G

Grouping, 31

I

Identifier
  # symbol, use of, 34

K

Key field, 8

L

Lookups, 51
  In action, 48

M

Many-to-Many
  First Taste, 45

N

Normalisation, 11, 51
  Example of tables afterwards, 35

Q

Query
  Caution!, 52
  Design view, 48
  Make-table, 32
  Select, 32, 39
  Update, 38, 39

R

RDBMS (Relational Database
  Management System)
  defined, 5
  why it exists, 7
Record, 8
Redundant, 4, 8, 10, 16, 32
  example of data, 4, 29
  Other types of redundant data, 4, 42
  Recognising and detecting, 30
  Removing redundant data, 36
Relationships, 51
  Creating, 4, 42

Decomposed, 57
Diagram of related tables, 44
Many-to-many, 57
Many-to-many solution example, 57
Null, blank or empty atributes, 4, 61
One-to-many, 43
Row, 8

S

Skeleton Tables, 63
Summary, 65

T

Tables, 8, 9
adding later, 10
Checking skeleton tables, 4, 63
Decomposed, 57
reason for relating, 10
Skeleton, 63
Terminology, 2