Databases from scratch I: Introduction

The first in a series of articles on choosing, designing and using a database program

When I was eighteen, I had a holiday job in an obscure back office of the New South Wales Stamp Duties Office. It doesn't take an overactive imagination to realise that a temporary position in this bureaucratic outpost was not the most exhilarating of experiences.

The work reached its low point when I was given the job of renumbering – by hand – 30,000 filecards. The cards were numbered from 1 to 30,000 and the powers that be wanted them renumbered 0 to 29,999 so that each batch of 100 cards would start with the same digits.

If only those filecards had been on a computer database! With a computer, this mind-numbing job, which took me several weeks to complete, could have been done in a matter of seconds. In fact, it would have been an utterly trivial task.

(1998, Rose Vine)

❖ The emergence of databases

Unfortunately, at the time I was suffering in the Stamp Duties Office, in the mid-70s, computer databases as we know them today were in their infancy. Around 1970 a researcher called Ted Codd had developed the "relational data model", which was to become the foundation stone of modern database technology. In the mid-70s, however, computer databases – particularly in the hands of end users – were not a common thing.

It wasn't until the beginning of the '80s, with the development of dBASE II (there was no dBASE I), that microcomputer-based databases started coming into their own. Although riddled with bugs, dBASE put enormous power into the hands of microcomputer developers and it remained the pre-eminent database program until the advent of Windows 3.x. With Windows 3 came a new breed of PC database, designed to be much easier to use than their DOS-based predecessors.

❖ What is a database?

Let's take a step back and define exactly what a database is. If spreadsheets are the 'number crunchers' of the digital world, databases are the real 'information crunchers'. Databases excel at managing and manipulating structured information.

What does the term 'structured information' mean? Consider that most ubiquitous of databases – the phone book. The phone book contains several items of information – name, address and phone number – about each phone subscriber in a particular area. Each subscriber's information takes the same form.

In database parlance, the phone book is a table, which contains a record for each subscriber. Each subscriber record contains three fields: name, address, and phone number. The records are sorted alphabetically by the name field, which is called the key field.

Other examples of databases are club membership lists, customer lists, library catalogues, business card files, and parts inventories. The list is, in fact, infinite. Using a database program you can design a database to do anything from tracking the breeding program on a horse stud to collecting information from the Mars Rover. And increasingly, databases are being used to build Web sites.

❖ Single and multi-file databases

A database can contain a single table of information, such as the phone book, or many tables of related information. An order entry system for a business, for example, will consist of many tables:

- n orders table to track each order
- an orders detail table for tracking each item in an order
- a customer table so you can see who made the order and who to bill
- an inventory table showing the goods you have on hand
- a suppliers table, so you can see who you need to re-order your stock from
- a payments table to track payments for orders

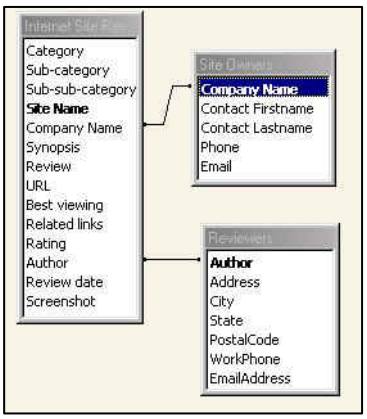
Each of these tables will be linked to one or more of the other tables, so that you can tie information together to produce reports or answer questions about the information you have in your database.

Multi-file databases like this are called relational databases. It's relational databases, as we'll see later in this series that provide exceptional power and flexibility in storing and retrieving information.

❖ A multi-file example

Relational databases are made up of two or more tables of information, which are connected in some way.

The example below shows a database used to track reviews of Internet sites which we are developing at Australian PC User magazine.



There are three tables in the database. The first is the Internet Site Reviews table, which includes information about each site, the company or person maintains the site, the review itself, and who wrote the review.

The Site Owners table contains contact details for each person or organisation that owns a site listed in the Site Reviews table. The two tables are linked to one another via the Company Name field, which they have in common.

The Reviewers table contains contact details for each person who writes site reviews. It's linked to the Site Reviews table via the Author field.

❖ Database programs

To create and maintain a computer database, you need a database program, often called a database management system, or DBMS. Just as databases range from simple, single-table lists to complex multi-table systems, database programs, too, range in complexity.

Some, such as the database component of Microsoft Works, are designed purely to manage single-file databases. With such a product you cannot build a multi-table database. You can certainly create numerous tables for storing different types of information, but there's no way to link information from one table to another. Such programs are sometimes called flat-file databases, or list managers.

Other database programs, called relational database programs or RDBMSs, are designed to handle multi-file databases. Claris FileMaker Pro is a relational database that's easy to use and fairly inexpensive.

The most popular relational databases are the offerings from the big three software companies. Lotus, Corel and Microsoft each produces a full-featured relational database application available both as a standalone program or as part of its integrated suite. Lotus has Approach, Corel has Paradox and Microsoft has Access.

❖ Database program tools

A database program gives you the tools to:

- design the structure of your database
- create data entry forms so you can get information into the database
- validate the data entered and check for inconsistencies
- sort and manipulate the data in the database
- query the database (that is, ask questions about the data)
- produce flexible reports, both on screen and on paper, that
- make it easy to comprehend the information stored in the database

Most of the more advanced database programs have built-in programming or macro languages, which let you automate many of their functions.

❖ Using a database

If the mention of programming languages makes you feel you're getting out of your depth, don't worry! Most of the database programs you're likely to encounter can be used at a variety of levels.

If you're a beginner, you'll find built-in templates, sample databases, 'wizards' and 'experts' that will do much of the hard work for you. If you find the built-in databases don't quite work for you, it's easy to modify an existing database so it fits your needs,

and it's not at all difficult to learn to create your own simple database structure from scratch

For more advanced users, the more powerful database programs enable you to create complete, custom-built, application-specific systems which can be used by others in your organisation or business.

❖ Planning ahead

There's one crucial thing you need to do whenever you create a database: plan ahead. Whether it's a single table or a collection of tables, you need to look at the information you want to store and the ways you want to retrieve that information before you start working on the computer. That's because a poorly structured database will hamstring you further down the track when you try to get your information back out in a usable form.

Databases from scratch II: Simple Database Design

The second in a series of articles on choosing, designing and using a database program

Have you ever noticed advertisements in the job classifieds for 'database architects' or 'database administrators'? It makes database design sound pretty intimidating, doesn't it? If they're so difficult to use that they require architects and administrators, what hope do we mere mortals have?

In fact, databases range from being ultra-simple to use to ultra-complex. In the world of personal computers, there are two main types of database programs. At the simple end of the scale are flat-file databases, also called single-file or list managers. These programs are as easy to learn as a word processor or spreadsheet. The initial concepts may take a little more time to absorb than word processing, but they're well within anyone's grasp. If you're using a database at home, in a class at school or in a small club or organisation, chances are the simple flat-file database will fill your needs.

❖ Relational databases

Things can get more complex when you use the other type of PC database program, called a relational database. With a relational database program you can create a range of databases, from flat-file structures to demanding multi-file systems. If you're using a database in your small business, a large organisation, or an ambitious school project, you're likely to need at least some of the features of a more complex relational database.

Whichever type of database program you use, the most crucial step in using it is to design your database structure carefully. The way you structure your data will affect every other action. It will determine how easy it is to enter information into the database; how well the database will trap inconsistencies and exclude duplicate records; and how flexibly you will be able to get information out of the database.

❖ A simple example

Let's take an ultra simple example: the phonebook. Say you've been given the job of placing your school or organisation's phone book on computer.

It should be easy: all you need is the name, the address and the phone number of each person. Your initial table design thus consists of three fields: name, address and phone number. Right?

Let's check.

Testing your design

The way to see if your database design works is to test it with some sample data, so feed the following records into your hypothetical table:

- J. T. Apples, 100 Megalong Dr Haberfield, 4992122
- B. York, 2/53 Alice Leichhardt, 5050011
- M. R. Sullivan, 9 Jay Leichhardt, 4893892

- B. J. Anderson, 71 Wally Rd Glebe, 2298310

Now tell the database program to sort the information:

- B. J. Anderson, 71 Wally Rd Glebe, 2298310
- B. York, 2/53 Alice Leichhardt, 5050011
- J. T. Apples, 100 Megalong Dr Haberfield, 4992122
- M. R. Sullivan, 9 Jay Leichhardt, 4893892

Revising your design

Immediately, you can see this is not what you want. You want the table sorted alphabetically by last name, not initials.

How can you fix this? Well, you could do some complex manipulation using a database feature called 'string functions' – if your program supports such a feature. Or, you could come up with a better table design in the first place: last name, initials, address and phone number. Feed your test data into this revised structure then tell the database program to sort your table using the last name followed by the initials. This time, you'll get the correct alphabetical listing:

- Anderson, B. J., 71 Wally Rd Glebe, 2298310
- Apples, J. T., 100 Megalong Dr Haberfield, 4992122
- Sullivan, M. R., 9 Jay Leichhardt, 4893892
- York, B., 2/53 Alice Leichhardt, 5050011

Keep on refining

Don't stop there. The table can be even more effective if you break the structure down further. For instance, if you'd like to have an easy way to list only those people who live in Leichhardt, this design won't help you. But with a little more work, you can break your database structure down further into last name, initials, street address, suburb, and phone number.

With this structure, you'll be able to sort your database alphabetically by last name or by suburb, and you'll be able to pluck out all those people who live in a particular suburb.

Take a look at this process of table refining in action

More is less!

Notice how creating an efficient table structure consists of breaking down your fields into simpler and simpler components? You end up with a table with many more fields than you might originally have thought necessary, but each of those fields houses much more basic information.

There's a technical term for this process: normalisation. If you wanted to become a database architect you'd have to become conversant with normalisation and functional dependencies and normal forms. If you're happy to remain a normal human, you can safely ignore these terms, provided you keep in mind that your task is to create a database structure that provides an efficient store for your information and that makes it flexible and easy to extract useful information.

In some ways, creating a database that's effective and simple to use is almost an anti-intuitive process. For example, our initial structure:

- name
- address
- phone number

Seems like it's a simpler design than our end result:

- first name
- last name
- street address
- suburb
- phone number

Creating useful fields

What you need to remember is that while the structure might look more complex, the contents of each field have been reduced to the simplest useful components. I say "useful" because we could, of course, break each field down further. For instance, we could break the street address field into two fields, street number and street name. But what would be the point? There's no need to extract or sort information in the database simply by street number or name, and so it's not a useful basis for breaking up the field. On the other hand, if we wanted to deal with multi-line addresses which are common for businesses, such as:

- Suite 5
- 122 Jones Street

then it makes sense to break the address field down into two simpler fields, address line 1 and address line 2. You're not likely to want to sort information based on only one of these fields, nor are you likely to use either of these fields in isolation. What is likely is that you'll want to have an easy way to print address line 1 and address line 2 as separate lines when addressing envelopes. So this field division becomes useful when getting information out of your database.

Permissible design infractions

I should add that creating these address sub-fields is not, technically, a good solution. Notice how with this table design, many records will have nothing in the address line 1 field, as most people's address consists of one line, rather than two. So you'll be wasting a lot of space in your database. Additionally, what about addresses that require more than two lines, such as:

- Suite 5
- Level III, Building A20
- 122 Jones Street

Our new table structure can't cope with this. Should we add another address line to ensure we cater to the infrequent address that needs three lines? Should we add a fourth line just in case…?

You can see the problems you can create by not getting the design right. A technically rigorous solution is to remove the address lines from our phonebook table

altogether, and stick them in an address table, that we then link to the phonebook table by a common field.

However, there's no need to fuss too much about such details. Many databases get by with minor infractions of database design rules, and you shouldn't feel hampered by such rules provided your table structures:

- provide flexible and simple output
- an online query of the database or printing reports
- eliminate redundant or duplicated information
- exclude inconsistencies

Computer-less design

One thing I hope you've noticed is that we've done all our design without the aid of a computer. This is as it should be, it lets you focus on the significance of the task without the distractions of trying to learn a database program at the same time.

You can design and test your database structure without going near a computer. The only thing you really need to know is the type of database program you'll use if it's a flat-file database, such as Microsoft Works, you'll be limited to single-table database design. If it's a relational program, such as Claris FileMaker Pro, Microsoft Access or Lotus Approach, you can design single- or multi-table databases.

In the next article in this series, we'll move on to relational design. You've seen how breaking down your fields into simpler components in a single table can help make it easier to get useful information out of your database. When we look at relational design, you'll discover how extending this process lets you address the other two goals of database design: eliminating redundant information and excluding inconsistencies.

Databases from scratch III: The Design Process

This third article in the series delves into the database heartland by exploring relational database design.

One of the best ways to understand database design is to start with all-in-one, flat-file tables design and then toss in some sample data to see what happens. By analysing the sample data, you'll be able to identify problems caused by the initial design. You can then modify the design to eliminate the problems, test some more sample data, check for problems, and re-modify, continuing this process until you have a consistent and problem-free design.

Once you grow accustomed to the types of problems poor table design can create, hopefully you'll be able to skip the interim steps and jump immediately to the final table design.

A sample design process

Let's step through a sample database design process:

We'll design a database to keep track of students' sports activities. We'll track each activity a student takes and the fee per semester to do that activity.

Step 1: Create an Activities table containing all the fields: student's name, activity and cost. Because some students take more than one activity, we'll make allowances for that and include a second activity and cost field. So our structure will be: Student, Activity 1, Cost 1, Activity 2, Cost 2

Step 2: Test the table with some sample data. When you create sample data, you should see what your table lets you get away with. For instance, nothing prevents us from entering the same name for different students, or different fees for the same activity, so do so. You should also imagine trying to ask questions about your data and getting answers back (essentially querying the data and producing reports). For example, how do I find all the students taking tennis?

Activities Table

Student	Activity1	Cost1	Activity2	Cost2
John Smith	Tennis	$36	Swimming	$17
Jane Bloggs	Squash	$40	Swimming	$17
John Smith	Tennis	$36		
Mark Antony	Swimming	$15	Golf	$47

Step 3: Analyse the data. In this case, we can see a glaring problem in the first field. We have two John Smiths, and there's no way to tell them apart. We need to find a way to identify each student uniquely.

Uniquely identify records

Let's fix the glaring problem first, then examine the new results.

Step 4: Modify the design. We can identify each student uniquely by giving each one a unique ID, a new field that we add, called ID.

We scrap the Student field and substitute an ID field. Note the asterisk (*) beside this field in the table below: it signals that the ID field is a key field, containing a unique value in each record. We can use that field to retrieve any specific record. When you create such a key field in a database program, the program will then prevent you from entering duplicate values in this field, safeguarding the uniqueness of each entry.

Our table structure is now: ID, Activity 1, Cost 1, Activity 2, Cost 2

While it's easy for the computer to keep track of ID codes, it's not so useful for humans. So we're going to introduce a second table that lists each ID and the student it belongs to. Using a database program, we can create both table structures and then link them by the common field, ID. We've now turned our initial flat-file design into a relational database: a database containing multiple tables linked together by key fields. If you were using a database program that can't handle relational databases, you'd basically be stuck with our first design and all its attendant problems. With a relational database program, you can create as many tables as your data structure requires.

The Students table would normally contain each student's first name, last name, address, age and other details, as well as the assigned ID. To keep things simple, we'll restrict it to name and ID, and focus on the Activities table structure.

Step 5: Test the table with sample data.

Students Table

Student	ID*
John Smith	084
Jane Bloggs	100
John Smith	182
Mark Antony	219

Activities Table

ID*	Activity1	Cost1	Activity2	Cost2
084	Tennis	$36	Swimming	$17
100	Squash	$40	Swimming	$17
182	Tennis	$36		
219	Swimming	$15	Golf	$47

Step 6: Analyse the data. There's still a lot wrong with the Activities table:

- Wasted space. Some students don't take a second activity, and so we're wasting space when we store the data. It doesn't seem much of a bother in this sample, but what if we're dealing with thousands of records?
- Addition anomalies. What if #219 (we can look him up and find it's Mark Antony) wants to do a third activity? School rules allow it, but there's no space in this structure for another activity. We can't add another record for Mark, as that would violate the unique key field ID, and it would also make it difficult to see all his information at once.
- Redundant data entry. If the tennis fees go up to $39, we have to go through every record containing tennis and modify the cost.
- Querying difficulties. It's difficult to find all people doing swimming: we have to search through Activity 1 and Activity 2 to make sure we catch them all.
- Redundant information. If 50 students take swimming, we have to type in both the activity and its cost each time.
- Inconsistent data. Notice that there are conflicting prices for swimming? Should it be $15 or $17? This happens when one record is updated and another isn't.

Eliminate recurring fields

The Students table is fine, so we'll keep it. But there's so much wrong with the Activities table let's try to fix it in stages.

Step 7: Modify the design. We can fix the first four problems by creating a separate record for each activity a student takes, instead of one record for all the activities a student takes.

First we eliminate the Activity 2 and Cost 2 fields. Then we need to adjust the table structure so we can enter multiple records for each student. To do that, we redefine the key so that it consists of two fields, ID and Activity. As each student can only take an activity once, this combination gives us a unique key for each record.

Our Activities table has now been simplified to: ID, Activity, Cost.

Step 8: Test sample data.

Students Table

Student	ID*
John Smith	084
Jane Bloggs	100
John Smith	182
Mark Antony	219

Activities Table

ID*	Activity*	Cost
084	Swimming	$17
084	Tennis	$36
100	Squash	$40
100	Swimming	$17
182	Tennis	$36
219	Golf	$47
219	Swimming	$15

Step 9: Analyse the data. We know we still have the problems with redundant data (activity fees repeated) and inconsistent data (what's the correct fee for swimming?). We need to fix these things, which are both problems with editing or modifying records.

Eliminate data entry anomalies

As well, we should check that other data entry processes, such as adding or deleting records, will function correctly too.

If you look closely, you'll find that there are potential problems when we add or delete records:

- Insertion anomalies: What if our school introduces a new activity, such as sailing, at $50. Where can we store this information? With our current design we can't until a student signs up for the activity.

- Deletion anomalies: If John Smith (#182) transfers to another school, all the information about golf disappears from our system, as he was the only student taking this activity.

Step 10: Modify the design. The cause of all our remaining problems is that we have a non-key field (cost) which is dependent on only part of the key (activity). Check it

out for yourself: the cost of each activity is not dependent on the student's ID, which is part of our composite key (ID + Activity). The cost of an activity is purely dependent on the activity itself. By checking our table structures and ensuring that every non-key field is dependent on the whole key, we will eliminate the rest of our problems.

Our final design will thus contain three tables: the Students table (Student, ID), a Participants table (ID, Activity), and a modified Activities table (Activity, Cost).

If you check these tables, you'll see that each non-key value depends on the whole key: the student name is entirely dependent on the ID; the activity cost is entirely dependent on the activity. Our new Participants table essentially forms a union of information drawn from the other two tables, and each of its fields is part of the key. The tables are linked by key fields: the Students table:ID corresponds to the Participants table:ID; the Activities table:Activity corresponds to the Participants table:Activity.

Step 11: Test sample data.

Students Table

Student	ID*
John Smith	084
Jane Bloggs	100
John Smith	182
Mark Antony	219

Participants Table

ID*	Activity*
084	Tennis
084	Swimming
100	Squash
100	Swimming
182	Tennis
219	Golf
219	Swimming

Activities Table

Activity*	Cost
Golf	$47
Sailing	$50
Squash	$40
Swimming	$15
Tennis	$36

Step 12: Analyse the results.

This looks good:

- No redundant information. You need only list each activity fee once.
- No inconsistent data. There's only one place where you can enter the price of each activity, so there's no chance of creating inconsistent data. Also, if there's a fee rise, all you need to do is update the cost in one place.
- No insertion anomalies. You can add a new activity to the Activities table without a student signing up for it.

- No deletion anomalies. If John Smith (#219) leaves, you still retain the details about the golfing activity.

Keep in mind that to simplify the process and focus on the relational aspects of designing our database structure, we've placed the student's name in a single field. This is not what you'd normally do: you'd divide it into firstname, lastname (and initials) fields. Similarly, we've excluded other fields that you would normally store in a student table, such as date of birth, address, parents' names and so on.

A summary of the design process

Although your ultimate design will depend on the complexity of your data, each time you design a database, make sure you do the following:

- Break composite fields down into constituent part.

Example:

Name becomes lastname and firstname.
Create a key field which uniquely identifies each record.
You may need to create an ID field (with a lookup table that shows you the values for each ID) or use a composite key.

- Eliminate repeating groups of fields

. Example:

If your table contains fields Location 1, Location 2, Location 3 containing similar data, it's a sure warning sign.

- Eliminate record modification problems

(such as redundant or inconsistent data) and record deletion and addition problems by ensuring each non-key field depends on the entire key. To do this, create a separate table for any information that is used in multiple records, and then use a key to link these tables to one another.

References:

http://www.geekgirls.com/database_dictionary.htm