

databases

step-by-step guides to using databases

databasics

- i: records & queries & keys, oh my!**
- ii: creating your first database**
- iii: data entry design**
- iv: streamlining data entry**
- v: getting information out**
- vi: exploring query types**
- vii: parameter queries**
- viii: queries with custom dialogs**

database design from scratch

- i: introduction**
- ii: simple database design**
- iii: the relational design process**
a data dictionary

which database tutorials?

[home](#)

Databases from scratch I: Introduction

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

Topic links:

[the emergence of databases](#)

[what is a database?](#)

[single and multi-file databases](#)

[a multi-file example](#)

[database programs](#)

[database program tools](#)

[using a database](#)

[when not to use a database](#)

[planning ahead](#)

Related articles:

[databases from scratch II: simple database design](#)

[databases from scratch III: the design process](#)

[databasics: a database dictionary](#)

[databasics i: records and queries and keys, oh my!](#)

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.

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

databasics ii: creating your first database

databasics iii: data entry design

databasics iv: streamlining data entry

databasics v: getting information out

databasics vi: exploring query types

databasics vii: parameter queries

databasics viii: parameter queries with custom dialogs

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 (see an example of [Database Web Publishing in Action](#)).

Single and multi-file databases

A database can contain a single table of information, such as the phone book ([click to see an example](#)), or many tables of related information. An order entry system for a business, for example, will consist of many tables:

- an 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

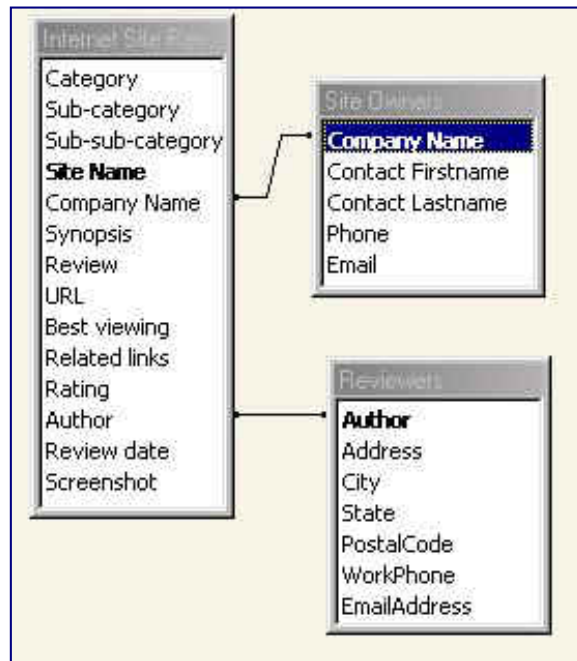
Each of these tables will be linked to one or more of the other tables, so that you can tie information together to produce reports or answer questions about the information you have in your database.

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 who 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. **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.

When not to use a database

Even though you can use a database program to do anything from managing the inventory of a parts supply warehouse to managing your personal finances, sometimes the smart option is to not use a database at all. That's because there's no point in reinventing the wheel: If you want a personal financial manager, you're far better off spending money on one of the commercial programs, such as **Microsoft Money** or **Intuit's Quicken**, than slaving for weeks creating your own version of the same thing.

The same goes for vertical market applications. Before you spend months designing a church contribution application or that parts inventory system, take a look around the Web or at your local software supplier to see if something similar has already been

created. Shareware libraries such as [ZDNet's Software Library](#) are littered with such specialist applications.

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.

We'll look at database design in detail in the next articles in this series.

[Straight on to Part II: Simple database design](#)

© 1998 Rose Vines



Support geekgirl's

Do you find the tutorials on this site useful? If so, please show your support by kicking in a few bucks to sponsor an orphanage for Afghan refugees. For a small amount, it is possible to make a difference in an area of the world which is hurting badly.



Want to know more? Read this [post on my blog](#).

[top](#)

[home](#)

[database menu](#)

Databases from scratch II: Simple Database Design

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

Topic links:

[relational databases](#)[a simple example](#)[testing your design](#)[revising your design](#)[keep on refining](#)[more is less](#)[creating useful fields](#)[permissible design
infractions](#)[computer-less design](#)

Related articles:

[databases from scratch I:
introduction](#)[databases from scratch
III: the relational design
process](#)[databasics: a database
dictionary](#)[databasics i: records and
queries and keys, oh my!](#)

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?

databasics ii: creating
your first database

databasics iii: data entry
design

databasics iv:
streamlining data entry

databasics v: getting
information out

databasics vi: exploring
query types

databasics vii: parameter
queries

databasics viii: parameter
queries with custom
dialogs

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 2 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 (whether you're asking 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 FileMaker's 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.

Straight on to Part III: The relational design process

© 1998 Rose Vines



Support geekgirl's

Do you find the tutorials on this site useful? If so, please show your support by kicking in a few bucks to sponsor an orphanage for Afghan refugees. For a small amount, it is possible to make a difference in an area of the world which is hurting badly.



Want to know more? Read this [post on my blog](#).

[top](#)

[home](#)

[database menu](#)

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 an all-in-one, flat-file table 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.

Topic links:

a sample design process:

step 1: create a table

step 2: test the table

step 3: analyse the data

uniquely identify records:

step 4: modify the design

step 5: test the table

step 6: analyse the data

eliminate recurring fields:

step 7: modify the design

step 8: test the table

step 9: analyse the data

eliminate data entry

anomalies:

step 10: modify the design

step 11: test the table

step 12: analyse the data

summary of the design

process

Related articles:

databases from scratch I:
introduction

databases from scratch II:
simple database design

databasics: a database
dictionary

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

databasics i: records and queries and keys, oh my!

databasics ii: creating your first database

databasics iii: data entry design

databasics iv: streamlining data entry

databasics v: getting information out

databasics vi: exploring query types

databasics vii: parameter queries

databasics viii: parameter queries with custom dialogs

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:

1. 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?
2. 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.
3. Redundant data entry. If the tennis fees go up to \$39, we have to go through *every* record containing tennis and modify the cost.
4. 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.
5. Redundant information. If 50 students take swimming, we have to type in both the activity and its cost each time.
6. 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. Note how the new structure lets students take any number of activities – they're no longer limited to two.

Step 8: Test sample data.

Students Table		Activities Table		
Student	ID*	ID*	Activity*	Cost
John Smith	084	084	Swimming	\$17
Jane Bloggs	100	084	Tennis	\$36
John Smith	182	100	Squash	\$40
Mark Antony	219	100	Swimming	\$17
		182	Tennis	\$36
		219	Golf	\$47
		219	Swimming	\$15
		219	Squash	\$40

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 tennis, for example, is \$36 for each and every student who takes the sport – so the student's ID has no bearing on the value contained in this field. The cost of an activity is purely dependent on the activity itself. This is a design no-no, and it's causing us problems. 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		Participants Table	
Student	ID*	ID*	Activity*
John Smith	084	084	Tennis
Jane Bloggs	100	084	Swimming
John Smith	182	100	Squash
Mark Antony	219	100	Swimming
		182	Tennis
		219	Golf
		219	Swimming
		219	Squash

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 parts. 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.

© 1998 Rose Vines



Support geekgirl's

Do you find the tutorials on this site useful? If so, please show your support by kicking in a few bucks to sponsor an orphanage for Afghan refugees. For a small amount, it is possible to make a difference in an area of the world which is hurting badly.



Want to know more? Read this [post on my blog](#).

[top](#)

[home](#)

[database menu](#)

Databasics: A Database Dictionary

Don't know the difference between a table and a database? Here's some basic database terminology defined.

Terms defined:

[database](#)
[data entry](#)
[dbms](#)
[field](#)
[flat file](#)
[foreign key](#)
[index](#)
[key field](#)
[normalisation](#)
[primary key](#)
[query](#)
[rdbms](#)
[record](#)
[relational database](#)
[report](#)
[SQL](#)
[table](#)

Related articles:

[databases from scratch I: introduction](#)
[databases from scratch II: simple database design](#)
[databases from scratch III: the design process](#)

database: A collection of related information stored in a structured format. *Database* is often used interchangeably with the term **table** (Lotus Approach, for instance, uses the term database instead of table). Technically, they're different: A table is a single store of related information; a database can consist of *one or more tables* of information that are related in some way. For instance, you could track all the information about the students in a school in a students table. If you then created separate tables containing details about teachers, classes and classrooms, you could combine all four tables into a timetabling database. Such a multi-table database is called a *relational database*.

data entry: The process of getting information into a database, usually done by people typing it in by way of data-entry forms designed to simplify the process.

dbms: Database management system. A program which lets you manage information in **databases**. Lotus Approach, Microsoft Access and FileMaker Pro, for example, are all DBMSs, although the term is often shortened to 'database'. So, the same term is used to apply to the program you use to organise your data and the actual data structure you create with that program.

field: Fields describe a single aspect of each member of a **table**. A student **record**, for instance, might contain a last name field, a first name field, a date of birth field and so on. All records have exactly the same structure, so they contain the same fields. The *values* in each field vary from record to record, of course. In some database systems, you'll find fields referred to as *attributes*.

flat file: A **database** that consists of a single **table**. Lightweight database programs such as the database component in Microsoft Works are sometimes called 'flat-file managers' (or list managers) because they can only handle single-table databases. More powerful programs, such as FileMaker Pro, Access, Approach and Paradox, can handle multi-table databases, and are called **relational database** managers, or **RDBMSs**.

foreign key: A key used in one **table** to represent the value of a **primary key** in a related table. While primary keys must contain unique values, foreign keys may have duplicates. For instance, if we use student ID as the primary key in a Students table (each student has a unique ID), we could use student ID as a foreign key in a Courses table: as each student may do more than one course, the student ID field in the Courses table (often shortened to Courses.student ID) will hold duplicate values.

index: A summary table which lets you quickly look up the contents of any **record** in a **table**. Think of how you use an index to a book: as a quick jumping off point to finding full information about a subject. A database index works in a similar way. You can create an index on any **field** in a table. Say, for example, you have a customer table which contains customer numbers, names, addresses and other details. You can make indexes based on any information, such as the customers' customer number, last name + first name (a composite index based on more than one field), or postal code. Then, when you're searching for a particular customer or group of customers, you can use the index to speed up the search. This increase in performance may not be noticeable in a table containing a hundred records; in a database of thousands of records it will be a blessing.

key field: You can sort and quickly retrieve information from a **database** by choosing one or more **fields** to act as *keys*. For instance, in a students table you could use a combination of the last name and first name fields (or perhaps last name, first name and birth dates to ensure you identify each student uniquely) as a key field. The database program will create an **index** containing just the key field contents. Using the index, you can quickly find any **record** by typing in the student's name. The database will locate the correct entry in the index and then display the full record.

Key fields are also used in **relational databases** to maintain the structural integrity of your tables, helping you to avoid problems such as duplicate records and conflicting values in fields (see **primary key** and **foreign key**).

normalisation: The process of structuring data to minimise duplication and inconsistencies. The process usually involves breaking down a single **table** into two or more tables and defining relationships between those tables. Normalisation is usually done in stages, with each stage applying more rigorous rules to the types of information which can be stored in a table. While full adherence to normalisation principles increases the efficiency of a particular database, the process can become so esoteric that you need a professional to create and understand the table design. Most people, when creating a database, don't need to go beyond the third level of normalisation, called *third normal form*. And there's no need to know the terminology: simply applying the principles is sufficient.

The first three levels in normalising a database are:

First Normal Form (1NF): There should be no repeating groups in a table.

For example, say you have a students table with the following structure:

```
student ID
name
date of birth
advisor
advisor's telephone
student
course ID 1
```

course description 1
 course instructor 1
 course ID 2
 course description 2
 course instructor 2

The repeating course fields are in conflict with first normal form. To fix the problems created by such repeating fields, you should place the course information in a separate course table, and then provide a linking field (most likely student ID) between the students table and the course table.

Second Normal Form (2NF): No non-key fields may depend on a *portion* of the primary key.

For example, say we create a course table with the structure:

student ID
 course ID
 course description
 course instructor

We can create a unique primary key by combining student ID + course ID (student ID is not unique in itself, as one student may take multiple courses; similarly, course ID is not unique in itself as many students may take the same course; however, each student will only be taking a particular course once at any one time, so the combination of student ID + course ID gives us a unique primary key).

Now, in 2NF, no non-key fields (course description, course instructor) may depend on a *portion* of the primary key. That, however, is exactly what we have here: the course instructor and course description are the same for any course, regardless of the student taking the course.

To fix this and put the database in second normal form, we create a third table, so our database structure now looks like this (with key fields in italics):

Student table

student ID
 name
 date of birth
 advisor
 advisor's telephone

Student courses table

student ID
course ID

Courses table

course ID
 course description
 course instructor

Third Normal Form (3FN): No fields may depend on other non-key fields. In other words, each field in a record should contain information about the entity that is defined by the primary key.

In our students table, for example, each field should provide information about the particular student referred to by the key field, *student ID*. That certainly applies to the student's name and date of birth. But the advisor's name and telephone doesn't change depending on the student. So, to put this database in third normal form, we need to place the advisor's information in a separate table:

Student table

student ID
name
date of birth
advisor ID

Student courses table

student ID
course ID

Courses table

course ID
course description
course instructor

Advisor table

advisor ID
advisor name
advisor telephone

primary key: A field that *uniquely* identifies a record in a table. In a students table, for instance, a key built from last name + first name might not give you a unique identifier (two or more Jane Does in the school, for example). To uniquely identify each student, you might add a special Student ID field to be used as the primary key.

query: A view of your data showing information from one or more tables. For instance, using the sample database we used when describing **normalisation**, you could query the Students database asking "Show me the first and last names of the students who take both history and geography and have Alice Hernandez as their advisor" Such a query displays information from the Students table (firstname, lastname), Courses table (course description) and Advisor table (advisor name), using the keys (student ID, course ID, advisor ID) to find matching information.

rd bms: Relational database management system. A program which lets you manage structured information stored in **tables** and which can handle **databases** consisting of multiple tables.

record: A record contains all the information about a single

'member' of a **table**. In a students table, each student's details (name, date of birth, contact details, and so on) will be contained in its own record. Records are also known as *tuples* in technical **relational database** parlance.

relational database: A **database** consisting of more than one **table**. In a multi-table database, you not only need to define the structure of each table, you also need to define the relationships between each table in order to link those tables correctly.

report: A form designed to print information from a database (either on the screen, to a file or directly to the printer).

SQL: Structured Query Language (pronounced *sequel* in the US; *ess-queue-ell* elsewhere). A computer language designed to organise and simplify the process of getting information out of a database in a usable form, and also used to reorganise data within databases. SQL is most often used on larger databases on minicomputers, mainframes and corporate servers.

table: A single store of related information. A table consists of **records**, and each record is made up of a number of **fields**. Just to totally confuse things, tables are sometimes called *relations*. You can think of the phone book as a table: It contains a record for each telephone subscriber, and each subscriber's details are contained in three fields - name, address and telephone.

Copyright 2008, Rose Vines

Support geekgirl's

Do you find the tutorials on this site useful? If so, please show your support by kicking in a few bucks to sponsor an orphanage for Afghan refugees. For a small amount, it is possible to make a difference in an area of the world which is hurting badly.



Want to know more? Read this [post on my blog](#).

[top](#)

[home](#)

[database menu](#)

Databasics I: Records and queries and keys, oh my!

A nice, slow approach to learning databases. Learn the lingo, start with simple stuff, and eventually take on something a little more challenging.

If you want to dive right into the nitty gritty of database design, try the companion series of articles, [Databases from Scratch](#). While there's some overlap between the two series, Databasics is aimed at complete beginners, while Databases from Scratch moves onto advanced topics, such as relational database design, more rapidly.

Some people have trouble envisaging what a database is. That's not surprising given that most databases consist of a number of component parts. But if you think of word processors as... well... processors of words, and spreadsheets as number processors, then you can think of databases as data processors. Feed a database data in any sort of guise – as numbers, text, dates, images, Web links – and it will digest all that data and turn it into usable information.

Another way to help distinguish a database from a spreadsheet (and there are, in fact, many times when you could just as easily use one as the other) is in terms of what sort of information you can get out of it. A spreadsheet lets you posit questions such as "Can I afford the monthly payments on that \$290,000 house?" or "What's the average daily rainfall for the first six months of this year?" or "If we increase our sales of doohickies by four percent while reducing the price by a dollar a piece, how will it affect profits?" Spreadsheets excel at answering financial, numerical and statistical questions such as these.

You can ask quite different questions of a database. For example, "What are the phone numbers and addresses of the five nearest post offices to the school?", "Do we have any books in our library that deal with irradiated food? If so, on which shelves are they located?", "Show me the personnel records and sales figures of our five best-performing sales people for the current quarter, but don't bother me with their address details."

As you can see, databases deal with a broader scope of information than spreadsheets. The chief job of a database is to provide answers to questions. In fact, one of the major components of most databases is a query tool – a special component which lets you formulate questions ('queries') that will elicit useful answers from all your stored data.

Topic links:

[database building blocks](#)

[think ahead](#)

[payoffs for planning](#)

[database dialect](#)

Related articles:

databasics series

ii: creating your first database

iii: data entry design

iv: streamlining data entry

v: getting information out

vi: exploring query types

vii: parameter queries

viii: parameter queries with custom dialogs

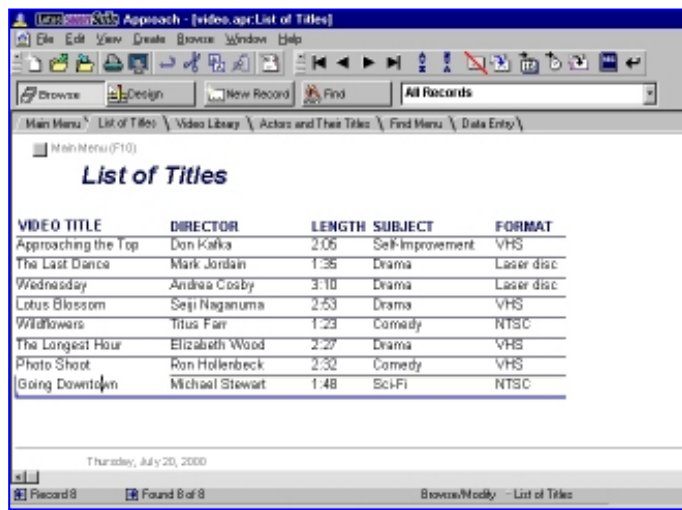
databases from scratch series

i: introduction

ii: simple database design

iii: the design process

a database dictionary

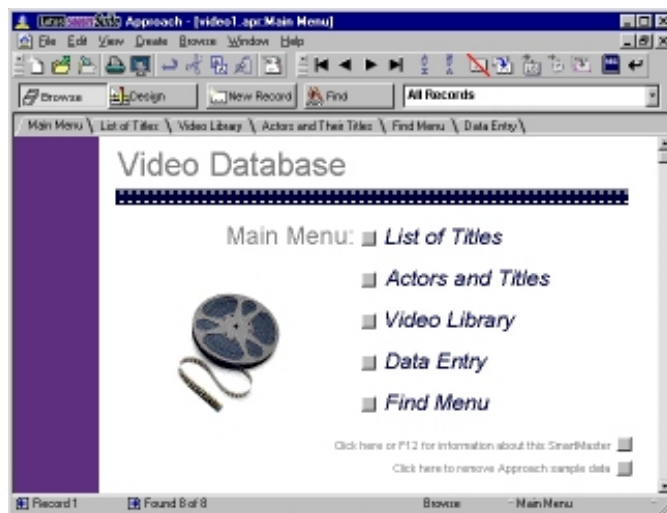


A multi-file database. (Click to see a larger image and detailed caption.)

Database building blocks

Of course, before you can ask any questions you need to build a structure to hold your data, and you need a way of adding data to that structure. So, most databases come with multiple components: a database designer for structuring the data; a forms designer for building data entry forms; a query engine for asking questions of that data; and a report builder for printing the results of a query.

It doesn't stop there. Many databases also provide a way to tie together and automate procedures so you can build entire applications that offer users push-button ease (well, that's the goal, anyway). Some database programs, such as Microsoft Access, include fully fledged programming languages for building applications. Others, such as FileMaker Pro and Lotus Approach, have beefed up macros or scripting languages that are easier to use but not quite as flexible as a programming language.



A menu-driven database application. (Click to see a larger image.)

Think ahead

The most important thing to do when you start creating a database is to think ahead. Before you even switch on the computer, think about the type of information you have to work with and the types of questions you'll want your database to answer.

For instance, say you want to create a membership list for your organisation. You'll want to store the name, address and phone number of each member, plus other details such as the year they first joined, type of membership, whether they're up to date with their membership dues, and so on. That gives us the following structure:

- Name
- Address
- Phone
- Year Joined
- Membership Type
- Dues owed

That seems clear enough. But what happens if you decide to print a membership list sorted alphabetically by family name – a very common requirement? There's no easy way to do it, so it makes sense to break the Name into its component parts: Given Name, Middle Initial, Family Name.

The same goes for the address. It, too, can be usefully broken down into Street Address, Suburb, State and Postal Code. That will enable you to print off envelopes sorted by postcode/zip and help you qualify for pre-sorted, bulk mail rates.

Payoffs for planning

As you can see, even in a seemingly simple database such as this, a little forethought can make a huge difference. Imagine that you'd used the original database structure above, entered the details for all 1700 of your members, and only then realised you wanted an alphabetical list. You'd have to add the new fields and then either re-enter all the affected data or use some advanced techniques – which, as a beginner, you're unlikely to know – to reorganise the existing data.

So before you dive in to creating your first database, get out the old pencil and paper and do some hard thinking about what you want to achieve.

While you're doing that, I'll conclude the opener to this series with a rundown on database terminology. In coming tutorials, you'll have a chance to build some databases from scratch, starting with simple, flat-file tables and working up to powerful relational databases. As you can tell from that last sentence, you'll probably need the database dictionary below, so keep it handy. If you want a more thorough grounding in database

terminology, you can always check out the [Database Dictionary](#).

Database dialect

Here's a quick guide to some of the more common database terms.

database: A collection of related information stored in a structured format. Database is often used interchangeably with the term table (Lotus Approach, for instance, uses the term database instead of table). Technically, they're different: a table is a single store of related information; a database can consist of one or more tables of information that are related in some way. For instance, you could track all the information about the students in a school in a students table. If you then created separate tables containing details about teachers, classes and classrooms, you could combine all four tables into a timetabling database. Such a multi-table database is called a *relational database*.

data entry: The process of getting information into a database, usually done by people typing it in by way of data-entry forms designed to simplify the process.

dbms: Database management system. A program which lets you manage information in databases. Lotus Approach, Microsoft Access and FileMaker Pro, for example, are all DBMSs, although the term is often shortened to 'database'. So, the same term is used to apply to the program you use to organise your data and the actual data structure you create with that program.

field: Fields describe a single aspect of each member of a *table*. A student *record*, for instance, might contain a last name field, a first name field, a date of birth field and so on. All records have exactly the same structure, so they contain the same fields. The values in each field vary from record to record, of course.

flat file: A database that consists of a single table. Lightweight database programs such as the database component in Microsoft Works are sometimes called 'flat-file managers' (or list managers) because they can only handle single-table databases. More powerful programs, such as Access, FileMaker Pro and Approach, can handle multi-table databases, and are called relational database managers, or RDBMSs.

index: A summary table which lets you quickly locate a particular record or group of records in a table. Think of how you use an index to a book: as a quick jumping off point to finding full information about a subject. A database index works in a similar way. You can create an index on any field in a table. Say, for example, you have a customer table which contains customer numbers, names, addresses and other details. You can make indexes based on any information, such as the customers' customer number, last name + first name (a composite index based on more than one field), or postal code. Then, when you're searching for a particular customer or group of customers, you can use the index to speed up the search.

key field: You can sort and quickly retrieve information from a

database by choosing one or more fields to act as keys. For instance, in a students table you could use a combination of the last name and first name fields as a key field. The database program will create an index containing just the key field contents. Using the index, you can quickly find any record by typing in the student's name. The database will locate the correct entry in the index and then display the full record.

primary key: A field that *uniquely* identifies a record in a table. In a students table, a key built from last name + first name might not give you a unique identifier (two or more Jane Does in the school, for example). To uniquely identify each student, you might add a special Student ID field to be used as the primary key.

record: A record contains all the information about a single 'member' of a table. In our students table, each student's details (name, date of birth, contact details, and so on) will be contained in its own record.

relational database: A database consisting of more than one table. In a multi-table database, you not only need to define the structure of each table, you also need to define the relationships between each table in order to link those tables correctly.

table: A single store of related information. A table consists of records, and each record is made up of a number of fields. You can think of the phone book as a table: It contains a record for each telephone subscriber, and each subscriber's details are contained in three fields – name, address and telephone.

Straight on to Part II: Creating your first database

© 2001 Rose Vines



Support geekgirl's

Do you find the tutorials on this site useful? If so, please show your support by kicking in a few bucks to sponsor an orphanage for Afghan refugees. For a small amount, it is possible to make a difference in an area of the world which is hurting badly.



Want to know more? Read this [post on my blog](#).

Support geekgirl's

Do you find the tutorials on this site useful? If so, please show your support by kicking in a few bucks to sponsor an orphanage for Afghan refugees. For a small amount, it is possible to make a difference in an area of the world which is hurting badly.



Want to know more? Read this [post on my blog](#).

[top](#)

[home](#)

[database menu](#)

Databasics II: Creating your first database

How to build a simple, single-file database. With step-by-step instructions for FileMaker Pro, Lotus Approach and Microsoft Access.

Topic links:

[single-table inadequacies](#)

[multi-table flexibility](#)

[when a single file will do](#)

[creating a membership database](#)

[creating the table in lotus approach](#)

[creating the table in filemaker](#)

[creating the table in access](#)

[what have we done](#)

[data types](#)

[field sizes](#)

[allowing for international variations](#)

[a word about field names](#)

[formatting and validation](#)

Related articles:

This tutorial guides you through building a simple, single-file database. In a single-file database, also known as a flat-file database, you put all your information into a single table. This is the simplest form of database to create, but it has some limitations and disadvantages. The most important of these limitations are that single-table databases are incapable of representing some real-world data and they create more work when it comes to data entry.

Imagine trying to create a single-table database to track inventory and sales. It simply wouldn't work. You'd need at least two tables and, in fact, many more (inventory, suppliers, sales, customers, and so on) to do the job.

Single-table inadequacies

When it comes to data entry, single-file databases often create work for you. For instance, if you create a flat-file database to catalogue your CDs, you have to put all the details, including the artist information, into one table. Say you want to include information such as the artist/band's recording label, band members, a discography and artist notes. How's that going to work? Your table structure might look something like this:

CD name
 CD date
 genre
 tracks
 artist/band name
 band members
 recording label
 discography
 notes

For each Beatles CD you own, you'll have to type in all those details. That means you'll have to type the names of all of the Beatles' releases repeatedly.

Multi-table flexibility

On the other hand, if you use a multi-file relational database, you can store the CD details (name, date, tracks and so on) in a CD table and store the artist details *once* in an Artist table. Your CD table will look like this:

CD name

databasics series

i: records and queries and keys, oh my!

iii: data entry design

iv: streamlining data entry

v: getting information out

vi: exploring query types

vii: parameter queries

viii: parameter queries with custom dialogs

databases from scratch series

i: introduction

ii: simple database design

iii: the design process

a database dictionary

CD date
genre
tracks
artist/band name

Your Artist table will look like this:

artist/band name
band members
recording label
discography
notes

You then link the two tables using the artist/band name field (that's why it's called a *relational* database – you define relationships between the tables) and enter the artist information once only. Each time you add a subsequent Beatles CD to your collection, you type *The Beatles* in the artist field and the database looks up the other details for you. It not only minimises effort on your part, it also ensures consistency of information and minimises the chance of introducing errors into the data.

When a single file will do

Having said all that, there are some applications where a single-file database is all you need. Also, if you're dealing with small quantities of information, it may not be worth your effort to design a relational database. You may prefer to create a simple all-in-one table and put up with any additional typing this necessitates.

So, in this tutorial we're going to focus on creating a flat-file membership database from scratch. It should be generic enough that you can use it as a basis for a membership application for your own organisation or club.

You can use the copy of Lotus Approach 97 we included on last month's cover CD to create your database or any other database program you like. The details will differ, but the principles are the same regardless of the program you use. I'll also include instructions on how to create the Membership database using FileMaker Pro 5 and Access 2000.

Creating a membership database

Let's get started by creating the table structure. We can then step back and analyse what we've done. I'll step you through the process in three popular database programs, Lotus Approach 97, FileMaker Pro 5 and Microsoft Access 2000. If you have a recent version of any of these programs, you should find the following instructions will work pretty well for you.

Creating the table in Lotus Approach

Here's how to proceed using Approach 97:

1. Start up Approach and, in the Welcome to Lotus Approach dialog, click the Create A New File Using A SmartMaster tab, select Blank Database and click OK.

2. In the New dialog, name the database *Members* and click Create.

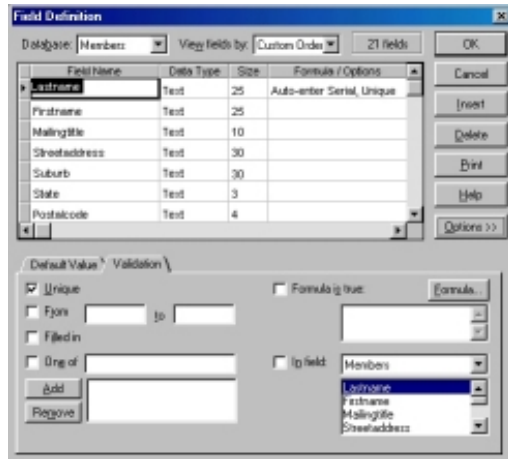


Figure 1. Creating the table structure in Approach. (Click for a larger image.)

3. In the first row of the Creating New Database dialog, type *MembershipNumber*, press Tab twice, type 5 in the Size box and click Tab once more to move to the next row.

4. Continue adding the following fields (name, data type and size):

First name, Text, 25
 Last name, Text, 25
 MailingTitle, Text, 10
 StreetAddress, Text, 30
 Suburb, Text, 30
 State, Text, 3
 PostalCode, Text, 4
 WorkPhone, Text, 20
 HomePhone, Text, 20
 ListWorkPhone, Boolean
 ListHomePhone, Boolean
 Email, Text, 60
 Joined, Date
 Active, Boolean
 MembershipType, Text, 9
 MembershipExpires, Date
 FeesPaid, Boolean
 AmountPaid, Numeric, 5.2
 Committee, Text, 12
 Skills, Text, 254

5. Highlight the MembershipNumber field you created, click the Options button, click the Validation tab and put a tick beside Unique. This will ensure that each member has a unique membership number.

6. Click OK to create the database. Approach will display your fields on a data entry form titled Blank Database.

Form2 \ Worksheet 1

Blank Database

Membership# 1 Firstname Rose Lastname Vinas Mailingtitle Ms

Streetaddress 123 Blue Jay Way Suburb Wentworth Falls State NS Postcode 2782 Workphone 02 4777-0000 Homephone 02 4777-0001

Listworkphone Listhomephone E-mail Joined Active Membershiptype

Yes No rosevines@home.com 4/6/1997 Yes full

MembershipExpires Feespaid Amountpaid Committee Skills

Figure 2. A standard data entry form in Approach. (Click for a larger image.)

Membership \ Worksheet 1

Member Details

Membership # 1

First name Rose Last name Vinas Mailing title Ms

Street address 123 Blue Jay Way Suburb Wentworth Falls State NSW Postcode 2782

Work phone 02 4777-0000 List work phone Yes Home phone 02 4777-0001 List home phone No E-mail rosevines@home.com

Joined 4/6/1997 Active Yes Membership Expires 4/6/2001 Membership type full Fees paid Yes Amount paid \$50.00 Committee

Skills none identifiable

Record 1 Found 2 of 2 Browse/Modify Membership

Figure 3. A customised data entry form in Approach. (Click for a larger image.)

Creating the table in FileMaker

To create the Members table in FileMaker Pro 5:

1. Start FileMaker and in the Open New or Existing File dialog box type *Members* in the File Name box and click Open. When FileMaker prompts you to create the file, click Yes.

2. In the Field Name box type MembershipNumber and press Enter. Unlike Approach and Access, you don't need to specify the field length in FileMaker.

3. Add the following fields in similar fashion:

FirstName
 LastName
 MailingTitle
 StreetAddress
 Suburb
 State
 PostalCode
 WorkPhone
 HomePhone

ListWorkPhone
ListHomePhone
Email
Joined (click Date in the Type section before pressing Enter)
Active (reselect Text in the Type section before pressing Enter)
MembershipType
MembershipExpires (click Date in the Type section and press Enter)
FeesPaid (reselect Text in the Type section and press Enter)
AmountPaid (click Number in the Type section and press Enter)
Committee (reselect Text in the Type section)
Skills

4. Select MembershipNumber in the field list, click the Options button, select the Validation tab and put a tick beside Unique.
5. Click Done. FileMaker will display your fields in a default data-entry form.

Creating the table in Access

To create the Members table in Microsoft Access 2000 (the process is fairly similar in Access 97):

1. Open Access and in the initial dialog box, select Blank Access Database and click OK.
2. In the File New Database dialog, type *Members* in the File Name box and click Create. You'll see a Members: Database window on your screen with Tables selected in the Objects panel.
3. Double-click Create Table In Design View.
4. Type MembershipNumber in the Field Name box and press Tab. In the Field Size box in the Field Properties section at the bottom of the window, replace the 50 with 5.
5. Click in the row beneath MembershipNumber and type FirstName, press Tab and change the Field Size value to 25.
6. Continue adding the following fields:
LastName, Text (data type), 25 (field size)
MailingTitle, Text, 10
StreetAddress, Text, 30
Suburb, Text, 30
State, Text, 3
PostalCode, Text, 4
WorkPhone, Text, 20
HomePhone, Text, 20
ListWorkPhone, select Yes/No in the Data Type column (a shortcut is to tab to the field and press Y), then Tab twice to move to the next row.
ListHomePhone, Yes/No data type
Email, Hyperlink data type
Joined, Date data type
Active, Yes/No
MembershipType, Text, 9
Membership expires, Date
FeesPaid, Yes/No

AmountPaid, Currency
Committee, Text, 12
Skills, Text, 255

7. Select the MembershipNumber field in the list and click the Primary Key icon (it looks like a little key) on the toolbar.

8. Click the Close box for the Table 1: Table window and, when prompted to save the changes, click Yes, type *Members* in the Table Name box and click OK.

9. Unlike the Approach and FileMaker, Access doesn't automatically display a default data entry screen. You can do so by clicking Members in the Members: Database window and then clicking the New Object: AutoForm button on the toolbar (it looks like a form with a lightning bolt across it).

What have we done?

You now have a Members table ready for you to enter information. Admittedly, the default data entry forms created by the database programs aren't much to look at and we haven't taken advantage of any shortcuts and special features that make databases easy to use. But it's a start.

Before we take the next steps – cleaning up the data entry form and thinking about improving the data structure – let's analyse what we've done so far.

To create our table, we defined a field for each item of information we want to store. The choice of fields was mostly a commonsense matter, although a couple bear closer examination.

We broke each member's name into two parts: first name and last name. This allows us to sort and search our database using either of those fields.

Similarly, we broke the address into component parts to give us a structure which lets us sort, search and group members by state, postcode and suburb. This division will also make it easy when we want to print address labels or envelopes. If you put the address all in one field, it becomes a very difficult task to create mailing labels which have the name on the first line, the address on the second line, and so on. It's sometimes desirable to break the address down even further, and include two street address lines (address line 1 and address line 2). This allows for addresses such as:

Suite 123, 14th Floor
997 Banks Drive
Bluegum Ridge, VIC, 3999

This breaking down of data into usable component parts is a vital step in creating a useful database.

Data types

Each field we've included has its own *data type*. The data type defines the type of information which may be stored in a field. The majority of our fields are text data type. Text fields can hold alphanumeric information, including letters, numbers, spaces and punctuation marks.

Other common data types include numeric (also known as number), currency (a specialised form of numeric field), date/time, Boolean (also called Yes/No), hyperlink, memo (for storing large quantities of text) and picture/object. Not all database programs support all these data types and our simple data structure uses only four types: text, numeric, Boolean and date.

Boolean fields are logical fields which may contain either a 0 or 1, or another binary pair such as True/False or Yes/No. They're useful when you want Yes/No answers to questions. We've used them in our database in the *ListHomePhone*, *ListWorkPhone*, *active* and *FeesPaid* fields to answer the questions "Should I list the member's home/work number when printing reports?", "Is this an active member?" and "Are the member's fees up to date?"

Notice how we've used the text data type for both the phone numbers and postal codes. Why not use the numeric data type?

With phone numbers, the answer's obvious: These numbers frequently contain non-numeric characters, such as parentheses and hyphens: (02) 4782-0000 for example. By using text data type we allow for such characters, as well as allowing for additional details such as *ext 34* (although you could, if you wish, create an additional field called *WorkExtension* to handle extension numbers).

As for the postcode, although this field will contain only numbers, we don't treat postcodes as numbers, that is, use them in numerical calculations. Because of this, and because of the way database sort and format numbers differently from text, always store this type of information in a text field.

Field sizes

Why is each field the size it is?

The most important thing about the size of your fields is that you make them big enough to accommodate the largest possible piece of information they will need to store.

With names and addresses, be generous. You may not be able to imagine a family name longer than 15 characters, but Ms Clarissa Worthington-Wettersley is going to be really annoyed when all her club correspondence is addressed to Ms Worthington-Wet.

As for fields where you're not quite sure how much info you need to store, such as the *Skills* field we've included, one approach is to allow the maximum permissible size for a text field, which is usually around 254 or 255 characters. Another approach is to use a memo data type. This type of field allows for text of almost any length to be entered.

If you're using FileMaker, you'll notice it doesn't even bother

asking you about field sizes. FileMaker manages field size dynamically. You can store about 65,000 characters in any text field. When you're designing your database, you can use formatting options on data entry forms (called Layouts in FileMaker) to limit data entry in fields.

Allowing for international variations

You may have noticed that the postalcode field in our table allows for only four characters. That's perfect for Australian postcodes, but won't work for the United States and many other countries.

You'll need to adjust the size depending on the requirements of your region. If you're creating a table that must accommodate US zip codes, you must decide whether a 5-character field will do, or whether you need to make the field 10 characters wide to allow for five-four zips (include the extra character to allow for the hyphen).

With phone numbers you should allow for formatting characters (including parentheses, plus signs and hyphens), extensions, international dialling codes and so on. I chose 20 characters as it handles most of these concerns.

If your database may be used in more than one locale, keep in mind such regional differences. For instance, two-character state codes work for the US, but not for Australia and many other countries. The same goes for phone numbers: Be careful to avoid truncated and specially formatted phone fields that don't work internationally.

A word about field names

You'll notice that many of the field names are compound words, such as FirstName and MembershipType. Why is that so? Why not make them more readable by including spaces?

Well, although all three database programs allow you to create field names which contain spaces, if you end up using your database with scripting tools such as JavaScript or with advanced data access technologies such as ADO (ActiveX Data Objects), you'll find spaces in fieldnames are unacceptable. So, even if you don't think you'll end up using such esoterica, it pays to allow for the possibility by eliminating spaces.

When it comes to creating data entry forms, you can always change the field name which is displayed above a data entry box by adjusting its *caption*, something we'll look at later.

Formatting and validation

We've created our database with almost no validation or formatting. The only exception is that we've made the MembershipNumber unique (in Access, we've turned it into a key field, which has the same effect).

Each database program lets you limit and control the type of data entered into each field and check that it matches permissible values. You might like to explore some of the options available to you.

In Approach, select Field Definition from the Create Menu to modify your database structure and check out the options. In FileMaker, select Define Fields from the File Menu and in Access, click the Tables button in the Object pane then right-click the Members table and select Design View from the pop-up menu.

Much of the data validation and automated data entry can be handled via the data form itself or by transforming our flat-file table into a relational database. We'll look at both of these options in coming tutorials.

[Straight on to Part III: Data entry design](#)

© 2001 Rose Vines



Support geekgirl's

Do you find the tutorials on this site useful? If so, please show your support by kicking in a few bucks to sponsor an orphanage for Afghan refugees. For a small amount, it is possible to make a difference in an area of the world which is hurting badly.



Want to know more? Read this [post on my blog](#).

[top](#)

[home](#)

[database menu](#)

Databasics III: Data entry design

Data entry screens are the face your database applications shows to users. Some simple guidelines ease data entry and help make data entered more accurate.

Topic links:

[data entry guidelines](#)

[i. organise fields logically](#)

[ii. don't clutter](#)

[iii. don't force users to scroll](#)

[iv. go easy on the fonts](#)

[v. go easy on the colours](#)

[vi. abandon the defaults](#)

[vii. check your tab order](#)

[viii. be consistent](#)

[ix. include help](#)

[x. provide shortcuts](#)

[xi. validate data as it's entered](#)

[experimenting with forms](#)

It's staggering how many shareware and commercial database applications have appalling data entry screens. Many developers seem to think that well-oiled inner workings are all that's needed to sell an application, when any user knows that, when you get down to it, the interface *is* the app.

When you're designing a database application, you're taking on the role of a developer. As you do so, remember to keep the soul of a user. While the brain work in building a database comes during the design stage, the hard slog comes when you – or other people – start adding data to that structure, especially when there are copious amounts of data to add. You can alleviate much of the tedium of data entry by ensuring your data entry forms are logically organised, easy on the eyes and efficient.

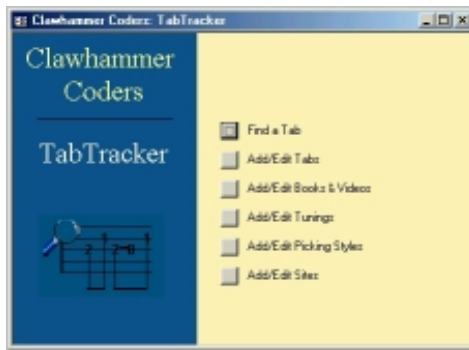


Figure 1. Think of your users when designing everything from data entry screens to a main menu for your database application. (Click the image for more details.)

Data entry guidelines

Exactly how you design your data entry screens will depend on the database program you're using, the amount of data you're dealing with, the needs and likes of the data entry personnel, and any application-specific requirements that may exist.

If you're designing a database purely for your own use it's still worth designing well-thought-out data entry screens. After all, why should you make life harder for yourself when a little effort in the design phase will make using your database easier for all time?

The following guidelines will help you design data attractive and easy-to-use data entry screens.

Related articles:**databasics series**

i: records and queries and keys, oh my!

ii: creating your first database

iv: streamlining data entry

v: getting information out

vi: exploring query types

vii: parameter queries

viii: parameter queries with custom dialogs

databases from scratch series

i: introduction

ii: simple database design

iii: the design process

a database dictionary

i. Organise fields logically

Group related fields together and use boxes or colour coding to make it easy for users to zero in on information quickly.

ii. Don't clutter

Space fields on the screen so users can easily spot the field they need to edit.

iii. Don't force users to scroll

If possible, make sure all your fields are visible simultaneously, so users don't have to keep switching between the keyboard (for data entry) and the mouse (for scrolling), and so users can see all information at a glance. It's also worth avoiding scrolled data entry screens as people have a tendency to forget about those invisible fields lurking off the bottom of the screen and leave them blank.

If you have a large number of fields in your database and you don't want to create clutter but still want to avoid scrolling, try using a tabbed interface (see Figure 2). With a tabbed interface, you can separate fields into logical groupings, with each group on its own tab. This approach avoids clutter while keeping everything on the screen.

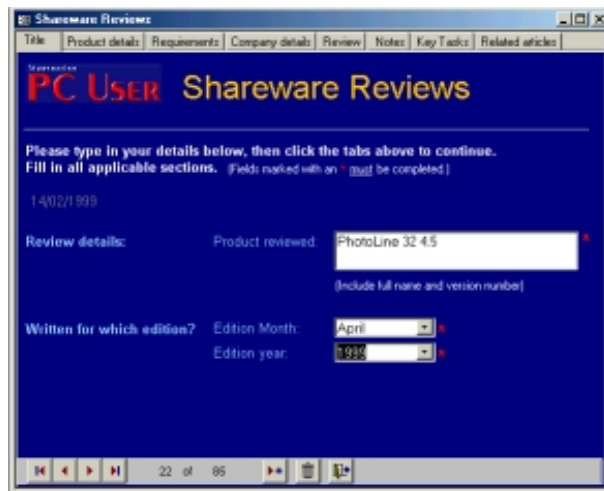


Figure 2. Using tabs for multi-screen data entry. (Click the image for more details.)

iv. Go easy on the fonts

At the most, your data entry screens should use three fonts – one for the headings, one for field labels, and one for the field contents themselves.

For best readability, use sans serif fonts such as Arial, Verdana and Trebuchet MS (sans serif fonts lack the little 'feet' or extra

small lines at the end of characters that you'll find in serif fonts such as Times New Roman). If large amounts of text will be entered in text fields, you may wish to use Times New Roman or another serif font for the fields. Choose standard fonts, especially if your database will be used on a variety of computers. Make sure all text is of a readable size.

v. Go easy on the colours

Avoid lairy colours. Instead choose soft colours or those that provide good contrast for the fields and captions. If a number of people will be using your database application, keep in mind that some of them may be colour blind or suffer from other eyesight difficulties.

vi. Abandon the defaults

Dump the default data entry screens created by Approach, Access, FileMaker and most other databases. They are usually poorly organised and either look boring or hard to read. Access is particularly bad in this respect, while FileMaker offers the best standard screens.

Use these default screens as a starting point and tweak them or, if you don't mind the extra work, build your screens from scratch.

vii. Check your tab order

Ensure that Tab order is correct. During data entry, you move from field to field using the Tab key. It's important you ensure the Tab order is consistent, so that the insertion point moves from the first field on the screen to the last without jumping about. Incorrect tab order will not only frustrate your users, but it may cause them to enter data into the wrong field.

You can change the tab order in Lotus Approach via the View Menu, Show Tab Order option when in Design Mode. In FileMaker Pro, you can change the tab order in Layout Mode via the Layout Menu, Set Tab Order option. In Access, open the form in Design View and choose Tab Order from the View Menu.

viii. Be consistent

If your application has more than one data entry screen, use the same look and the same organisational principles for each screen. If, for example, you place a Close button on one data entry screen so users can close the screen with a mouse click, include the same box in a similar location on other screens.

ix. Include help

Add descriptive help where possible. Your data entry forms should be as self-explanatory as possible, with commonsense

labels on each field.

You can make data entry even easier by providing help for each field. In Microsoft Access, for example, you can add a pop-up field tooltip which will appear when the user lets their cursor linger on a field (see Figure 3).

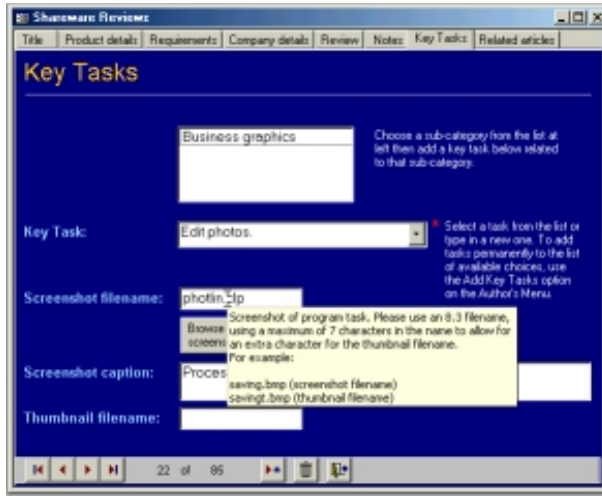


Figure 3. Add help via the use of Access's tooltips. (Click the image for more details.)

If your database doesn't support such tooltips, you can include instructions directly on the data entry form itself. For instance, if you want people to enter phone numbers without including any special characters (such as hyphens or parentheses) you can add a message to that effect beside the phone number field, thus: *Enter numbers only, eg. 0291237745.*

x. Provide shortcuts

You can speed up data entry dramatically by providing shortcuts such as drop-down boxes with predefined choices, auto-filled fields, multiple choice fields and so on (see Figure 4). Some of these features you can add during the database definition stage, although we kept things simple in the previous tutorial and avoided such options. Others you can add when you design your data entry forms. Still others you can introduce by using lookup tables or related tables, which let users 'look up' related information and have it entered automatically.

We'll look at all these options in the future.

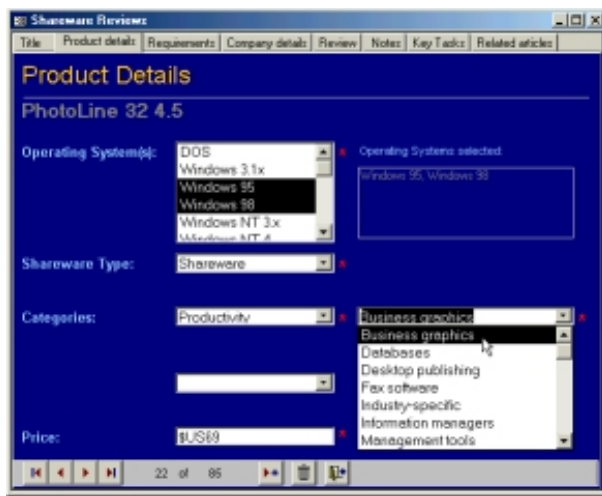


Figure 4. Drop-down lists make light work of data entry. (Click the image for more details.)

xi. Validate data as it's entered

You should design your forms so they check for invalid data and give users a chance to correct mistakes they've made. Much of this validation can be done during database definition; some you'll add during form design.

We'll look at data validation in more detail in the future as well.

Experimenting with forms

Each database program provides different tools for designing data entry forms. Most are very fiddly to work with, so it pays to practice. One of the best ways to do that is to use the standard data entry form created by the database and then mess around with it. Don't do this experimenting on a 'live' database. Create a sample one from scratch, if you like, and work with it.

If you're using Lotus Approach, you modify the data entry screens by clicking the Design button on the toolbar. Then click the Properties button on the toolbar (its icon is a yellow square overlaying a blue square at an angle) and then try clicking different objects on the screen and inspecting and changing the contents of the Properties box for that object.

In FileMaker Pro select Layout Mode from the View Menu. Right-click an object to adjust its properties.

In Access, select a table in the Tables section of the database objects window, click the New Object: Autoform button on the toolbar to create a new form, and then click the View button on the toolbar to switch to Design View. Click the Properties button on the toolbar and select different objects on the form to adjust the settings.

We'll do some hands-on forms design in the next article in this series.

Straight on to Part IV: Streamlining data entry

© 2001 Rose Vines



Support geekgirl's

Do you find the tutorials on this site useful? If so, please show your support by kicking in a few bucks to sponsor an orphanage for Afghan refugees. For a small amount, it is possible to make a difference in an area of the world which is hurting badly.



Want to know more? Read this [post on my blog](#).

[top](#)

[home](#)

[database menu](#)

Databasics IV: Streamlining data entry

Make data entry easier and more accurate by streamlining data entry tasks.

Topic links:

[the membership table](#)

[describing our fields](#)

[forcing data into a mold](#)

[creating an input mask](#)

[assuming common values](#)

[eliminating errors](#)

[adding validation help](#)

[adding lookup tables](#)

[lookup lists in filemaker](#)

[lookup lists in approach](#)

Related articles:

databasics series

[i: records and queries and keys, oh my!](#)

[ii: creating your first database](#)

[iii: data entry design](#)

[v: getting information out](#)

In the previous tutorial we discussed some guidelines for creating data entry screens that are easy to use. In this tutorial, we'll put those guidelines into action by refining the membership database we introduced in [part two](#) of this series. I'll step through the process in Microsoft Access 2000. If you're using FileMaker Pro or Lotus Approach, read through the tutorial, and at the end you'll find some brief directions to achieve the same results using those database programs.

At this stage, we'll still be working on the data structure (or *data definition*), rather than the data entry form itself. That's because Access (as well as Approach and FileMaker Pro) lets us lay the foundations for efficient form design while we're working on the table definition process. By adjusting field properties and options, we can build data entry shortcuts and safeguards right into the structure of our database.

The membership table

We're going to use the same table structure we used in part two of this series (see Table 1). If you need to create the table from scratch, do so using Design View and enter the structure shown in Table 1. If you already have a Members table ready to modify, open the table in Design View (click the Tables section in the Objects list, highlight the Members table, and click the Design button).

Field name	Data type	Size
MembershipNumber*	Text	5
FirstName	Text	25
LastName	Text	25
MailingTitle	Text	10
StreetAddress	Text	30
Suburb	Text	30
State	Text	3
PostalCode	Text	4
WorkPhone	Text	20
HomePhone	Text	20
ListWorkPhone	Yes/No	

vi: exploring query types

vii: parameter queries

viii: parameter queries
with custom dialogs**databases from
scratch series**

i: introduction

ii: simple database design

iii: the design process

a database dictionary

ListHomePhone	Yes/No	
Email	Hyperlink	
Joined	Date	
Active	Yes/No	
MembershipType	Text	9
MembershipExpires	Date	
FeesPaid	Yes/No	
AmountPaid	Currency	
Committee	Text	12
Skills	Text	255

Table 1. The Members table structure (* the asterisk denotes a key field).

Note: Increase the size of the PostalCode field if you live in the US or another country which uses more than four characters for postal/zip codes.

Describing our fields

There are several changes we need to make to our original table structure to streamline data entry design.

Our fieldnames are descriptive enough but they won't make very readable captions on a data entry form. So highlight the MembershipNumber field and type *Membership #* in the Caption field property near the bottom of the screen (this time we include the space as it's merely a label on a data entry form, not part of our table structure).

Move to the next field and add the caption *First name*. Continue adding the labels:

Last name
Mailing title
Street # and name
Suburb
State
Postcode (you may want to label this *Zip* if you're in the US)
Work phone
Home phone
List work number?
List home number?
Email
Member since
Active?
Membership type
Membership expires
Paid up?
Amount paid
Committee
Skills

Now scoot back up to the MembershipNumber field and set its Required property to Yes. This forces the data entry operator to type in a value for this field, ensuring that each member is

assigned a number.

Forcing data into a mold

If you wish, you can add an *input mask* to the Work phone and Home phone fields. An input mask restricts the way data is entered into a field, so that it is consistent. For instance, if you want all phone numbers to be entered in the standard Australian format (nn) nnnn-nnnn, you can define an input mask to ensure that. If you're US-based, you might want to use an input mask that looks like this: (nnn) nnn-nnnn.

Be wary about using input masks. Don't use them if there's any chance at all that some data will not fit the mask. For instance, if some of the phone numbers in your table are Australian national numbers in the form nn nn nn or US toll-free numbers in the form 1 nnn nnn-nnnn, your input mask won't fit.

Creating an input mask

Where you're sure the data will be consistent, feel free to add a mask. To add a mask with the Australian (nn) nnnn-nnnn format to the Work phone field, for instance, you:

1. Click the Work Phone field and then click in the Input Mask property box.
2. Click the button displaying an ellipsis (...) to display the Input Mask Wizard.
3. Click the Phone Number mask in the list and click Next.
4. Edit the Input Mask to read *(99)9999 9999* and make sure the underscore (_) character is selected in the Placeholder list. You can test your mask by typing sample data in the Try It box. Click Next. (If you'd like to know more about input mask symbols, open Help and type *inputmask* in the Index keyword box .)
5. You can choose to store the data either with or without the parentheses. Choose Without The Symbols and click Next, then Finish.

Assuming common values

If one of your fields usually contains a particular value, you can speed up data entry by specifying that value as the *default* for the field. Access will then fill in that value for you – allowing you to modify it whenever you need to.

Say, for instance, that our organisation is based in Tasmania and the vast bulk of our members live locally. We can save time by doing this:

1. Clicking the State field.

2. Typing *TAS* in the Default Value field.

Eliminating errors

We can get rid of a lot of errors in our database at the source by validating data as it is entered. To do so, you define a *validation rule* for a field. All data entered will be checked to see whether it conforms to the rule; if it doesn't, the user will be prompted to correct the error and won't be able to continue until it has been fixed.

Say, for example, each membership number takes the form of a single, uppercase letter followed by four numbers. Further, the uppercase letter represents the membership type – F for Full, A for Associate, L for Life. To add this rule:

1. Click the MembershipNumber field.
2. Click in the Validation Rule property and type *Like "[FAL]####"*.

If you'd like to know more about validation rules, click F1 while your cursor is in the Validation Rule box. You can find out more about the use of wildcard characters (like the # used in the rule above) by opening Access Help and typing *wildcard* in the Index or Answer Wizard.

Adding validation help

If you do use validation rules, make sure you also create a matching validation text property. The validation text is the message that pops up if the user types in an invalid entry. If you don't create your own validation text, you'll leave your users at the mercy of Access's default error messages, which are quite likely to cause heart arrhythmia (see Figure 1). So, click in the Validation Text box and type something friendly and useful.

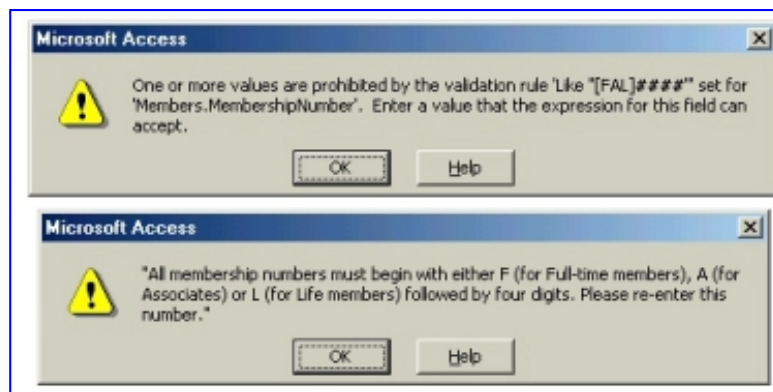


Figure 1. Which error message would you rather be confronted with? Access's in-built one (top) or the custom-made one (bottom)? The lesson is, don't rely on inbuilt error messages – create your own so your users are informed rather than bemused. (Click the image to see a full-sized picture of the messages.)

Adding lookup tables

We're going to do one more very cool thing before we finish with our table structure. We're going to help speed up data entry by providing pre-defined choices for certain fields (see Figure 2).

For example, we have three membership types – Full, Associate and Life. We're going to make it so our data entry operator can pick the membership from a drop-down list, instead of having to type the value each time.

To do this, we need to define a list of the permissible membership types and then allow the operator to 'look up' the appropriate value. Fortunately, Access provides a Lookup Wizard that steps us through this process.

1. Click in the Data Type box beside the MembershipType field. A drop-down arrow will appear. Click it and select Lookup Wizard.
2. Select I Will Type In The Values That I Want and click Next. This option lets us create a quick list of our three values. The other option (I Want The Lookup Column To Look Up The Values In A Table Or Query) is useful when you have either a changeable or very large lookup list – say a list of all postcodes. With this option, you can store the lookup values in their own updateable table.
3. In the Number Of Columns box type *1* and then type *Full*, press Tab and type *Associate*, press Tab and type *Life* under the Col1 heading.
4. Adjust the column width so it just fits the longest value by dragging the right edge of the column header leftwards, then click Finish.

Note, if you want to restrict the values to those contained in this list, you should either select MembershipType in the field list, click the Lookup tab, and change the Display Control from Combo Box to List Box; or you should create a Validation Rule that limits the permissible values to *Like "Full" Or Like "Life" Or Like "Associate"*.

Figure 2. Lookup lists can be huge timesavers when it comes to data entry time. Use them liberally where appropriate.

Now that you've created one lookup list, try doing the same thing for the MailingTitle (Ms, Mr, Dr, Professor, and so on), State (TAS, VIC, NSW, QLD, et cetera) and Committee (Membership, Volunteers, Fund Raising, et cetera) fields.

Lookup lists in FileMaker

FileMaker Pro and Lotus Approach offer similar features for data defaults, masks, validation and lookup lists. For example, here's how you can define a lookup list for the MembershipType field in FileMaker Pro:

1. Open the Members database and choose Define Value Lists from the File Menu.
2. Click New, type *Membership Types* in the Value List Name box, and then type *Full, Associate and Life* (separated by carriage returns) in the Use Custom Values box.
3. Click OK and then Done.
4. Switch to Layout Mode (via the View Menu) and right-click the Membership Type field.
5. Select Field Format from the pop-up list.

6. Select Pop-up List, and select the Membership Types value list from the drop-down Using Value List box.
7. Click OK, revert to Browse Mode, and test out your new value list.

Lookup lists in Approach

To do the same thing in approach:

1. Open the Members database and click the Design button.
2. Right-click the Membership Type field, select Field Properties from the pop-up menu, and in the Basic tab click Field Definition.
3. Click the Options button and then the Validation tab.
4. Click One Of, type *Full*, click Add, type *Associate*, click Add, type *Life*, click Add, and click OK.

Straight on to Part V: Getting information out

© 2001 Rose Vines



Support geekgirl's

Do you find the tutorials on this site useful? If so, please show your support by kicking in a few bucks to sponsor an orphanage for Afghan refugees. For a small amount, it is possible to make a difference in an area of the world which is hurting badly.



Want to know more? Read this [post on my blog](#).

[top](#)

[home](#)

[database menu](#)

Databasics V: Getting information out

Queries let you make sense of the data you have stored in your database. Learn how to create simple, yet remarkably useful, queries by example.

Topic links:

[queries](#)

[query language](#)

[making comparisons](#)

[using operators to make comparisons](#)

[broadening and narrowing the scope](#)

[using qbe](#)

[an example in access](#)

[a more complex example](#)

[the same thing in filemaker pro](#)

[and in lotus approach](#)

[more complex still](#)

[sample data](#)

Related articles:

[databasics](#)

The whole point of using a database is to turn *data* into *information*. Data are facts which have no inherent meaning; information is data put into context to convey meaning.

Think of a customer database containing information such as customer names, addresses and phone numbers. In the State field, you'll find the values QLD, VIC, WA, NT, NSW and so on repeated over and over (that is, if you live in Australia – adjust the values to suit your locale). Without a context, these are meaningless items of information. Put a question to your database such as "What percentage of our customers live in WA?", however, and the resulting answer is useful, meaningful information.

Queries

We've spent the last few tutorials looking at how to structure a database and how to design data entry forms. It's vital to get this design phase right, otherwise you can't be sure any information you get from your database will be trustworthy. Additionally, you want to make sure data entry is as easy as possible not only for the comfort of the people stuck with this task, but also to help ensure data is entered correctly. The more attention you pay to the design phase, the better. But the really interesting and useful stuff happens when you start trying to get information out of your database. This is where you turn the stored data into information – a process called querying.

A query is a question you 'ask' a database in order to get information in a useful format. As databases don't understand plain English (although there have been attempts to design database programs that do), you have to phrase your question according to rules. The rules are usually referred to as a *query language*.

Query language

If this sounds a little intimidating, don't let it faze you. Chances are you're already familiar with using a simple form of query language on the Internet. That's because Web search engines are, in fact, a type of database. When you type a search phrase such as:

piano steinway bechstein

into a Web search engine, the search engine queries itself using

series

i: records and queries and keys, oh my!

ii: creating your first database

iii: data entry design

iv: streamlining data entry

v: getting information out

vi: exploring query types

vii: parameter queries

viii: parameter queries with custom dialogs

databases from scratch series

i: introduction

ii: simple database design

iii: the design process

a database dictionary

the three keywords you've supplied. Your search will turn up all pages that contain the word piano or Steinway or Bechstein, or any combination of the three words.

If you've done any advanced Web searching, you may have tried creating Boolean search queries (named after George Boole, an English mathematician who helped develop symbolic logic). In Boolean queries you use logical operators such as AND (or &), OR and NOT to create more precise searches. Like this:

piano AND (steinway OR bechstein)

This search query will locate Web pages that contain the words *piano* and *Steinway*, or the words *piano* and *Bechstein*, or all three words.

Querying a database is similar. The query language may be more extensive and you'll be given tools that make the job very precise, but if you can use a search engine you won't have too many problems learning how to query a database.

The actual way you create a query depends on the database program you use. For example, the database in Microsoft Works uses an ultra-simple, fill-in-the-blanks approach called *query by example* (QBE for short). Microsoft Access also has a QBE feature (called *Filter By Form*), but you'll do most of your querying in Access using the Query Builder, a highly sophisticated query tool. Lotus Approach and FileMaker Pro also offer a choice of methods to query your data, including QBE.

Making comparisons

You need more than Boolean operators to create a query. You also need a way to compare one value to another, and that's where *comparison operators* come into play. In combination with Boolean operators, they let us create questions about our data that the database can understand.

You'll be familiar with comparison operators from school arithmetic. Table 1 shows you a list of them.

Operator	Meaning
=	is equal to
<>	is not equal to
>	is greater than
>=	is greater than or equal to
<	is less than
<=	is less than or equal to

Table 1. The main comparison operators

To understand how to use comparison operators, imagine a database containing a country table which includes the following fields (the data type of each field is shown in parentheses):

Fieldname	Field type
-----------	------------

Name	text
Area	numeric
Population	numeric
UnitedNationsMember	yes/no
JoinedUN	date
SecurityCouncilMember	yes/no

(Note: You'll find a link at the end of this article which will let you download a copy of the Countries database containing 194 country records, which you can use to try out your own queries.)

Using operators to make comparisons

Consider the types of information you might try to get from the database. Here are some examples:

1. What's the population of Namibia?
2. Which countries are members of the Security Council and when did they join the UN?
3. Which countries are not members of the UN?
4. Which countries have a population under 20 million?
5. Which countries names begin with A, B or C?
6. Which countries have a population over 20 million or an area of over 2 million square kilometres?
7. Which countries have a population over 20 million and an area over 2 million square kilometres?

Take that first question – What's the population of Namibia? To rephrase it so our database can understand what we're asking, we make use of the = (is equal to) operator:

Show me the record in the Countries table where the Name field="Namibia", and display the value of the Population field for that record.

Let's paraphrase the rest of our questions:

2. Show me the records where SecurityCouncilMember=Yes, and display the value of the JoinedUN field for the matching records
3. Show me the records where UnitedNationsMember<>Yes
4. Show me the records where Population<20,000,000
5. Show me the records where Name<"D"
6. Show me the records where Population>20,000,000 OR Area>=2,000,000

7. Show me the records where Population>20,000,000 AND Area>=2,000,000

Broadening and narrowing the scope

Most of these are self explanatory, but let's take a closer look at the last two where we've combined comparison and Boolean operators.

Example 6 displays all countries which *either* have a population of more than 20 million *or* which have an area of 2 million square miles or more *or* which have both a population over a million and an area that's 2 million square miles or more.

Example 7 displays only those countries which have *both* a population of over 20 million *and* an area of 2 million square miles or greater.

The OR operator broadens the scope of the query to include more records. The AND operator narrows the scope of the query. One other thing you may have noticed in example 5: You're not restricted to using operators such as < and > with numbers alone. They work equally well with text and dates.

Using QBE

If all these operators have your head in a spin, never fear! Most database programs do all the hard work of concocting queries in the background, while you merely fill in a form to show the type of information you're after. Nevertheless, if you want to create more complex queries or milk the most subtle information from your data, you'll certainly put this knowledge to use. Even using query by example you'll use comparison operators regularly.

Let's step through a query by example in Access, FileMaker and Approach. You can download a copy of the [sample database](#) to try it for yourself.

geekgirl:geopolitical update

Since I first wrote this article, the world has moved on. Both Switzerland and the new nation of East Timor have joined the United Nations (welcome to both of you!). I have updated the sample database to reflect these changes, but have left the text of these articles intact, partly for historical interest but also as a clear example of why database queries are so wonderful: Even though the data in the Countries database has changed, well-constructed queries will continue to provide correct results.

Unfortunately, I haven't had time to update the population details in the database, so take all results with a grain of salt!

An example in Access

1. Double-click the Countries table in the Database window to open the table in Datasheet View. This view shows a list of all the

records in the database.

2. Click the Filter By Form button on the toolbar (it has a picture of a funnel on top of a form). Your records will disappear and you'll be presented with a single blank record.

3. In the name field type *Namibia* and then click Apply Filter (the button with the filter on it). The matching record will be displayed.

4. Click the Apply Filter button (it's now called the Remove Filter button) once more to redisplay all your records.

A more complex example

Now, for a slightly more complex query, let's search for countries that have a population of 50 million or less which are also members of the UN Security Council:

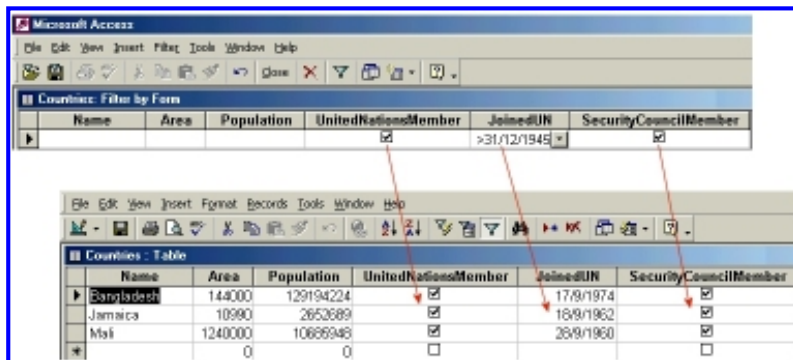
1. Keep the Countries database open in Datasheet View.

2. Click the Filter By Form button.

3. If "Namibia" still shows in the Name field, delete it.

4. In the Population field, type ≤ 50000000 (don't put commas in the number) and tick the box in the SecurityCouncilMember field.

5. Click the Apply Filter button. You should see the records for Mali, Jamaica and the Ukraine.



Query By Example is delightfully simple, but it still has its pitfalls. For example, if you fail to tick (place a checkmark in) the UnitedNationsMember field in this Access query, you'll get no matches. That's because you'd be asking for records where the country is *not* a member of the United Nations but *is* a member of the UN Security Council, which, of course, cannot be. Click the image to see a full-sized version.

The same thing in FileMaker Pro

1. Open the Countries database and press Ctrl-F to go into Find Mode.

2. Type *Namibia* in the Name field and press Enter to display the

matching record. Note that the number of matching records is displayed on the left.

3. Click the Show All Records button (the one with an eye) to redisplay all your records.

For the second query:

1. Press Ctrl-F to enter Find Mode.
2. Tab down to the Population field and type `<=50000000`.
3. Tab to the SecurityCouncilMember field, type `Yes` and press Enter to display the matching records. Flip through the matches by clicking the Records icon on the left.

And in Lotus Approach

1. Open the Countries database and click the Find button to display a blank form.
2. Type `Namibia` in the Name field and press Enter to display the matching record. In the status bar, you'll see the number of records located.
3. Click the drop-down box which currently displays `<Current Find/Sort>` and select All Records to redisplay all records.

For the second query:

1. Click Find.
2. Type `<=50000000`.
3. Tab to the SecurityCouncilMember field (the label gets truncated to `SecurityCo` because the Approach file format doesn't allow field names longer than 10 characters), type `Yes` and press Enter. You can flip through the matching records by clicking the Go To The Next Record button on the toolbar.

More complex still

So far we've merely skimmed the surface of using queries. The QBE examples show how easy it is to get useful information out of a database, but there's a whole lot more to querying, including the ability to save and re-use queries, and to create queries that prompt the user for information. We'll examine queries in more detail in the next tutorial.

Sample Data

The following links let you download a copy of the Countries table used in this tutorial. Choose the correct version for your database.

[Access 2000 sample](#)

[Access 97 sample](#)

[FileMaker Pro sample](#)

[Lotus Approach sample](#)

[Straight on to Part VI: Exploring query types](#)

© 2001 Rose Vines



Support geekgirl's

Do you find the tutorials on this site useful? If so, please show your support by kicking in a few bucks to sponsor an orphanage for Afghan refugees. For a small amount, it is possible to make a difference in an area of the world which is hurting badly.



Want to know more? Read this [post on my blog](#).

[top](#)

[home](#)

[database menu](#)

Databasics VI: Exploring query types

Explore comparative queries, queries which use calculated fields, select queries, action queries and more.

Topic links:

[comparative queries](#)

[using access's query designer](#)

[saving and re-running queries](#)

[query types](#)

[a select query example](#)

[calculated fields](#)

[an action query](#)

[sample data](#)

Related articles:

databasics series

[i: records and queries and keys, oh my!](#)

[ii: creating your first database](#)

[iii: data entry design](#)

[iv: streamlining data entry](#)

In the previous tutorial we used Query By Example (QBE) on a database of countries to answer questions such as 'Which countries are members of the UN Security Council?' and 'Which countries have a population over 20 million and an area over 2 million square kilometres?' These are simple queries which we created using QBE and basic operators such as > (greater than).

It's possible to ask more complex questions, such as:

- Which country joined the UN most recently?
- Which countries are in the top 5% in terms of area?
- Which five countries have the lowest population density?

The way you tackle these queries depends on which database program you use. I'll be using Microsoft Access in the examples, as it has querying tools that leave most other PC databases in the shade. You'll find a [sample copy of the database](#) at the end of this article.

Comparative queries

The simple queries we explored last time did nothing more than find a matching value (Name=Namibia), or compare a value in a field to a constant (Population<=20000000).

We can take this a step further and run queries that will compare one piece of data against the rest of the data in the table. These are real bread-and-butter queries. For instance, in a sales database, you might want the answers to questions such as: Which region had the lowest sales for the quarter? Who are our five best salespeople? Which salespeople performed better this quarter than last?

To ask these types of questions, you need to use something a little more sophisticated than QBE. In Access, you use the Query Designer.

Using Access's Query Designer

Let's start with the question, Which country joined the UN most recently? Here's how to get the answer:

1. Open the Countries database in Access.

v: getting information out

vii: parameter queries

viii: parameter queries
with custom dialogs

**databases from
scratch series**

i: introduction

ii: simple database design

iii: the design process

a database dictionary

2. In the main Database Window, click Queries in the Objects list.
3. Double-click Create Query In Design View. A Query window opens together with a Show Table box containing all the tables in our database.
4. Select the Countries table, click Add, and then click Close. This adds the Countries database to the Query window, so we can work with it.
5. Click Name in the list of fields and then hold down the Ctrl key and click JoinedUN (you may need to scroll to see it). Drag the two fields to the left-hand column in the query design grid. The two fields will appear side by side in the grid. The top row in the grid shows the field name; the second row shows which table the field belongs to (we'll get into multi-table queries at a later date); the third row lets you specify a sort order for the results; the fourth row (Show) specifies whether you want the contents of this field displayed in the results; the fifth and sixth rows specify criteria for selecting records.
6. In the JoinedUN column on the grid, click in the Sort row and, from the drop-down box, select Descending. This tells Access we want the table sorted with the most recent records (those with the 'largest' dates) first.
7. In the toolbar, beside the Sigma button you'll see an empty drop-down box. This is the Top Values button. Type 1 in this box. We're looking for the 'top date', so to speak – the most recent date in the JoinedUN field – so we want the top, single value.
8. Run the query by clicking the Run button (it has an exclamation mark on it). You'll see the result is Tuvalu, which joined the UN on the 5th September, 2000.

geekgirl:geopolitical update

In fact, if you run this query on the updated version of the Countries database I now have online, you'll find the answer is not Tuvalu, but East Timor. Both East Timor and Switzerland have joined the United Nations since I first wrote this article. While I've left the article text the same, I have updated the database to reflect these recent additions to the United Nations family (I haven't had a chance to update the population figures, alas).

These changes show how useful database queries can be: Provided the data in your database is accurate and up to date, you can use a query you created years ago on your current data and get an accurate and up-to-date result.

Saving and re-running queries

Click the Close box on the Query window. You'll be asked whether you want to save the query. Click Yes, name the query

Most Recent Member and click OK.

Why should you save a query? After all, you already know the answer to your question is Tuvalu. But then, the answer won't always be Tuvalu. Say Switzerland decides to join the UN this year, or a new country emerges from one of the bubbling spots on the globe and becomes a new member? (See the Geopolitical Update above.)

By saving the query, you can re-run it at any time and find the up-to-date answer to your question. This may seem a fairly trivial operation with our simple example, but when you're creating complex queries or, more importantly, when your data changes regularly, saved queries eliminate a lot of work.

To use your saved query, simply double-click it in the Queries list.

Query types

The *Most Recent Member* query you created is called a *Select query*. Access lets you create five different types of query:

Select queries. Used to retrieve data from one or more tables and display the results in a datasheet, which you can save or modify. You can also use *Select queries* to group records and calculate sums, averages and so on.

Parameter queries. For creating on-the-fly queries which prompt the user for criteria at the time the query is run. For example, you can create a parameter query that answers the question: Which countries have a population greater than X and less than Y? Each time you run the query, it will prompt you for the values of X and Y. Thus you can use the same query repeatedly to discover different information.

Crosstab queries. Used to summarise data from one field and group it in tabular fashion according to two criteria.

Action queries. Queries that make changes to the records in a table. There are four type of action queries: *Delete queries* remove records from a table; *Update queries* make global changes to a group of records in a table; *Append queries* add records from one or more tables to the end or one or more tables; *Make-table queries* create a new table from all or part of the data in an existing table.

SQL queries. A query created using SQL, which is a highly advanced querying language. SQL queries give you enormous flexibility, but require a high degree of expertise to use effectively.

A select query example

Let's quickly run through a second select query, answering the question: Which countries are in the top 5% in terms of area?

1. Double-click *Create Query In Design View*.

2. Select the Countries table, click Add, and then click Close.
3. Click Name in the list of fields, Ctrl-click Area, and drag the two fields to the grid.
4. In the Area column, choose Descending from the Sort drop-down box.
5. In the Top Values box on the toolbar, type 5% (or select it from the drop-down list).
6. Click Run to execute the query.

Calculated fields

How about the question: Which five countries have the lowest population density? We don't have a Population Density field in our table, so how can we calculate it?

We do it by creating a new field which becomes part of our results (note, though, that the field does not become part of the existing table structure). Here's how:

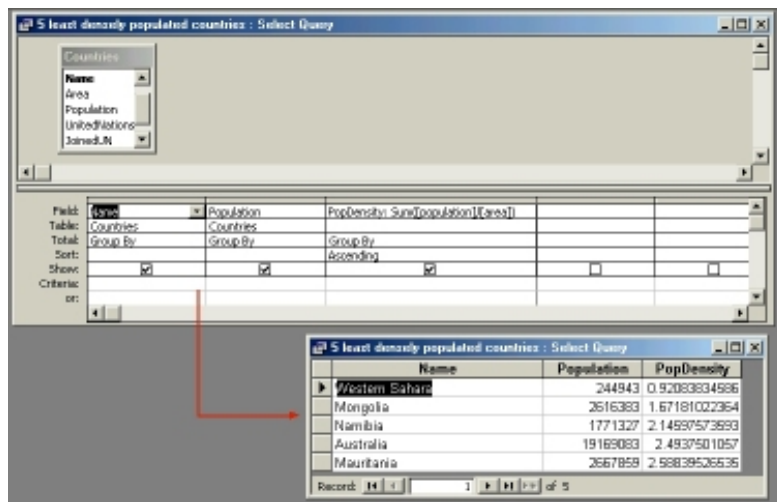
1. Double-click Create Query In Design View.
2. Select the Countries table, click Add, and then click Close.
3. Drag the Name and Population fields onto the grid.
4. In the top row of the empty third column on the grid, type:

PopDensity: population/area

and press Enter. We're creating a new column (PopDensity) whose values will be calculated by dividing each country's population by its area. By the way, if you can't see the entire contents of this column, drag the right-hand edge of the column header to the right to resize it.

5. In the Top Values box on the toolbar, type 5.
6. Click the Totals button (it has the Sigma sign on it). A new Totals row appears in the grid.
7. In the PopDensity column choose Ascending from the Sort box.
8. Click Run to execute the query.

The results show the names, population and population density for the world's least populated countries.



This Select query in Access finds the five most sparsely populated countries in the world using a calculated field, PopDensity. Click the image to see a full-size screenshot.

An action query

Let's finish up by converting that last query into an action query. This query doesn't merely provide us with the results; it saves them in a new table which includes a PopDensity field in its structure. The original table remains intact and unchanged; it's the new results table which has the extended structure.

1. If you have the query results still showing, click the View button at the left end of the toolbar to view the query in design mode once more. If you have already closed the query, recreate it.
2. Click the down arrow on the Query Type button on the toolbar and select Make-Table Query.
3. Name the query *Sparsely Populated*, leave the Current Database option selected, and click OK.
4. Run the query and, when prompted, say Yes to create the new table.
5. Close the query, click Tables in the Objects panel, and you'll see a new table called Sparsely Populated. Double-click it to see the query results in the new table. You'll notice the new table contains five records (the most sparsely populated five countries) and three fields – Name, Population and PopDensity.

Sample Data

The following links let you download a copy of the Countries table used in this tutorial. Choose the correct version for your database.

Access 2000 sample

[Access 97 sample](#)

Straight on to Part VII: Parameter queries

© 2001 Rose Vines



Support geekgirl's

Do you find the tutorials on this site useful? If so, please show your support by kicking in a few bucks to sponsor an orphanage for Afghan refugees. For a small amount, it is possible to make a difference in an area of the world which is hurting badly.



Want to know more? Read this [post on my blog](#).

[top](#)

[home](#)

[database menu](#)

Databasics VII: Parameter Queries

Interactive, on-the-fly parameter queries add a huge amount of flexibility and power to run-of-the-mill static queries.

If the title of this tutorial sounds a little intimidating, never fear. Toss the term 'parameter query' around at a party and you may get a few gasps of admiration, but understanding and using this highly useful type of query is no big deal.

A parameter query is interactive – it prompts the user for information when it's run. You can use a parameter query to ask the user for criteria for retrieving records or for a particular value to insert in a field. This makes parameter queries great for generating dynamic, on-the-fly results.

If you think of the queries we've examined in earlier articles in this series, while the data may change, the query itself is static:

- Which countries have a population over 20 million and an area of over 2 million square kilometres?
- Which country has the smallest population density?

With parameter queries, both the data and the queries may change over time:

- Which countries have a population between x and y?
- Which countries joined the United Nations in the nth decade of 2000?

You obtain the values for those variables – x, y and n – by asking the user to supply them each time the query is run.

Creating a parameter query

Let's take a look at some parameter queries in action, using Microsoft Access, which provides tools that let us do this job easily. We'll use the Countries database we've used previously (you'll find a download link at the bottom of this page) and start by creating a non-parameter query to answer the question "Which countries have a population greater than one hundred million?" We'll then transform that query into a parameter query and contrast the results.

Here goes:

1. Open the Countries database in Access.
2. In the main Database Window, click Queries in the Objects list.
3. Double-click Create Query In Design View to open the Query window and Show Table box.
4. Select the Countries table to add it to the Query window,

Topic links:

[creating a parameter query](#)

[parameter tips](#)

[obtaining a range](#)

[a range example using dates](#)

[multiple values](#)

[avoiding endless dialogs](#)

[sample data](#)

Related articles:

databasics series

[i: records and queries and keys, oh my!](#)

[ii: creating your first database](#)

[iii: data entry design](#)

[iv: streamlining data entry](#)

[v: getting information out](#)

vi: exploring query types

viii: parameter queries
with custom dialogs**databases from
scratch series**

i: introduction

ii: simple database design

iii: the design process

a database dictionary

click Add, and then click Close.

- Click Name in the list of fields, hold down the Ctrl key and click Population. Drag the two fields to the left-hand column in the query design grid to add the fields to the query.
- In the Population column in the grid, type:

>100000000

in the Criteria row.

- Click the Run button (it has an exclamation mark on it) to run the query. You should see 11 countries listed, each with a population greater than 100 million.

That's our simple, non-parameter query, which we created by providing a constant (100 million) as the matching criteria. Now let's change it into a more flexible parameter query, which uses variables provided by the user instead of a constant.

- Click the View button (it's the left-most button on the toolbar) to return to Design View, so we can modify the query.
- Delete the contents of the Criteria cell in the Population column and replace it with:

>[Show countries with a population over:]

You may need to resize the column (by dragging the right edge of the column header to the right) so you can see the entire contents.

- Click the Run button. You'll be presented with a dialog box containing the prompt, "Show countries with a population over:" Fill in any value you like and click OK to see the results.
- Save this query by clicking the Save button and naming it *Population over x*.

Parameter tips

Neat, huh? Let's examine the query to see how we did this.

The criteria consists of the greater than operator (>) followed by the prompt we wish to use enclosed in square brackets:

>[Show countries with a population over:]

That's simple enough, but there are two important things to keep in mind when you create your parameter prompts. First, be succinct. Access lops the prompt off at 37 characters, so if you are verbose you'll end up with shabby-looking, truncated prompts.

Second, be as clear as possible. Although you want to be succinct, don't become cryptic. For instance, you could just as easily use the criteria:

>[Pop greater than:]

or even

>[Pop?]

Those will work, but if someone else tries to use your database or you use the query only occasionally, a more informative prompt will avoid confusion and ensure the correct input.

Obtaining a range

Your parameter queries are not restricted to a single prompt. You can use them to find a range of values or obtain multiple criteria.

Here's a query which answers the question, "Which countries have a population between x and y?" by prompting for a lower and an upper value:

1. If you still have the results of the *Population over X* query open, click the View button to return to Design View. If you closed the query after saving it, highlight it in the Queries list and click Design to open it in Design View.

2. In the Criteria cell of the Population column, add:

AND <[and a population under:]

The full criterion should now read:

>[Show countries with a population over:] AND <[and a population under:]

3. Run the query and you'll be prompted twice. Enter a lower limit in the first dialog and an upper limit in the second, then click OK to see the results.

4. From the File Menu choose Save As and name the query *Population Range*. By saving the query, you'll be able to run it repeatedly, supplying different values each time.

This range query is similar to queries we've explored before, using two comparison operators (> and <) and the logical operator AND, but instead of using constant values, it substitutes two parameters obtained via dialogs.

A range example using dates

Let's try one more range parameter query, this time using dates. This one will answer questions such as "Which countries joined the United Nations in the nth decade of 2000?"

1. Create a new query in Design View.
2. Select the Countries table and click Add to add it to the query, then close the Show Tables dialog.
3. Drag the fields Name and JoinedUN onto the query grid.

4. In the Criteria cell of the JoinedUN column, type:

>[Joined UN after what date:] And <[and before what date:]
5. Run this query and respond to the prompts by typing dates in the format dd/mm/yyyy. To view all countries that joined the United Nations during the nineties, for instance, enter the values 31/12/1989 and 1/1/2001. If you use mm/dd format instead of dd/mm format for dates, you'll want to make that first value 12/31/1989, of course.
6. Save this query and name it *Date Range*.

You can, of course, use other operators (<>, =, <= and so on) and logical expressions (OR, NOT) in range queries.

Multiple values

As well as matching a range of values in a single field, you can use parameter queries to match values in more than one field.

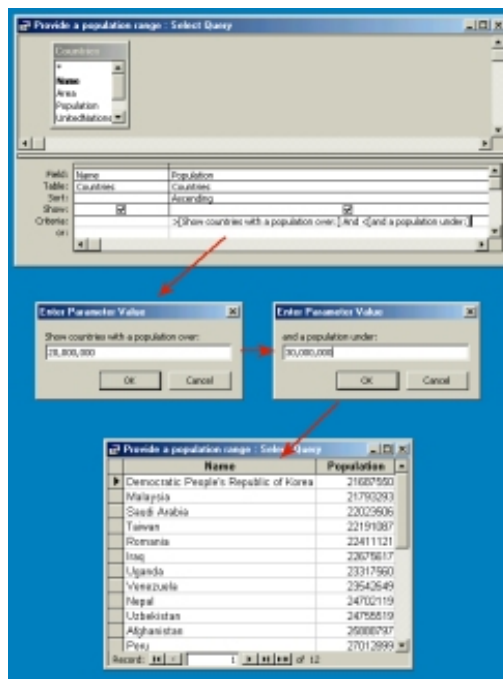
Let's modify the Date Range query to let us not only specify a date range, but also limit the answer to countries over a certain population.

1. Open the Date Range query in Design View.
2. Drag the Population field from the field list onto the query design grid to the right of the JoinedUN field. Our query now contains three fields – Name, JoinedUN and Population.
3. In the Criteria cell of the Population column, type:

>[and has a population over:]

4. Run the query.

You'll be presented with three prompts, asking for lower and upper dates and then a lower population limit. Note that this query in essence uses the AND logical operator to extend the query, so it's the equivalent of: Show me countries that joined the UN between date x AND date y AND which have a population over n.



Running a parameter query: A parameter query prompts the user for one or more criteria. In this example, the user specifies a population range and the query returns all matching records. Click the image to see a full-size screenshot.

Avoiding endless dialogs

Try experimenting with parameter queries on your own. You'll find they add enormous flexibility to the types of information you can wheedle out of your database. A tourist bureau, for example, can use them to provide information to the public on local events occurring this week. A small business can find all those customers who placed orders over a specific amount in the previous quarter. You're sure to come up with instances where you can use them with your own data.

The only drawback with these queries is that you can end up peppering your users with endless, rather unattractive prompt boxes as you try to collect all the necessary parameters.

There's good news. With a little fast footwork, you can avoid these dialogs and replace them with a single, good-looking dialog of your own design. I'll show you how in the next article in this series.

Sample Data

The following links let you download a copy of the Countries table used in this tutorial. Choose the sample data appropriate for your version of Microsoft Access.

[Access 2000 sample](#)

[Access 97 sample](#)

Straight on to Part VIII: Parameter queries with custom dialogs

© 2001 Rose Vines



Support geekgirl's

Do you find the tutorials on this site useful? If so, please show your support by kicking in a few bucks to sponsor an orphanage for Afghan refugees. For a small amount, it is possible to make a difference in an area of the world which is hurting badly.



Want to know more? Read this [post on my blog](#).

[top](#)

[home](#)

[database menu](#)

Databasics VIII: Parameter queries with custom dialogs

Customise your dialogs to make your queries more usable.

Topic links:

[the task](#)

[when a box is a form](#)

[querying the form](#)

[what type of data?](#)

[at your command](#)

[how'd we get here?](#)

[rock and roll](#)

[sample data](#)

Related articles:

databasics series

[i: records and queries and keys, oh my!](#)

[ii: creating your first database](#)

[iii: data entry design](#)

[iv: streamlining data entry](#)

[v: getting information out](#)

[vi: exploring query types](#)

[vii: parameter queries](#)

In the previous tutorial, we explored how to create simple parameter queries in Microsoft Access which prompt the user for criteria when run. These are real bread-and-butter queries you'll find yourself using over and over. The only drawback with such queries is that you have no control over the look of the prompt dialog boxes displayed and, with complex queries, you may end up presenting your users with prompt after prompt after prompt.

Your database application will be much more enjoyable to use if you eliminate this dialog box proliferation and spruce up the dialogs. To do so, you'll need to create a custom dialog box which consolidates all the prompts.

The task

We're going to use the Countries database we've used in past tutorials (you'll find a link to download a copy at the end of this article) and design a dialog box for the parameter query with which we ended the previous column.

That query was designed to elicit the following information:

Show me countries that joined the UN between date x AND date y AND which have a population over n.

If we create such a query without using a custom dialog box, we'll have to badger our users with three consecutive dialogs, the first asking *Joined the UN after what date?*, the second asking *And joined before which date?*, and the final prompting *And has a population over?* With our custom dialog, we can consolidate these questions and also gain control over the appearance of the dialog box itself.

We'll build the form first, then recreate a modified version of the query which takes advantage of the custom dialog box.

I'll describe the process using Access 2000. Access 97's interface is a little different, but you should be able to adapt these instructions with very few changes.

When a box is a form

As far as Access is concerned, a dialog box is simply another type of form, like the data entry forms you encountered early on in this series. So that's where you'll start – by creating a form:

databases from scratch series

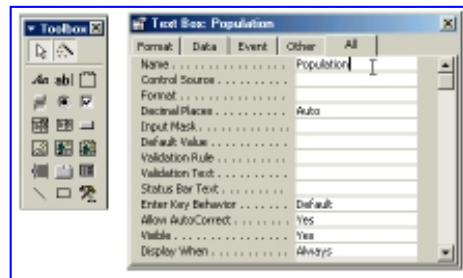
i: introduction

ii: simple database design

iii: the design process

a database dictionary

1. Open the Countries database and, in the main database window, click the Forms button, then double-click Create Form In Design View. You'll see a blank form titled Form1: Form.
2. If it's not already displayed, display the Toolbox by clicking the Toolbox icon (a crossed hammer and spanner) on the toolbar. Make sure the Control Wizards are enabled (the Control Wizards icon at the top of the Toolbox should be depressed). Drag the toolbox to the side so it's easily accessible but doesn't obscure the form window. Take some time to run your cursor over each of the Toolbox icons so you can see what it contains.



Access provides an assortment of tools with which to customise forms. Click the image above to see a full-sized image and detailed caption.

3. We want to add a data entry box for each of our three prompts. So click the Text Box tool in the Toolbox, then click-and-drag on the form to create a rectangular text box (position it a little down from the top and about two-thirds of the way from the left edge). You'll see that a label is created to the left of your form. Click the label to select it and resize it by dragging the resizing handle at its left edge towards the left of the form. Click within the label, delete the existing text and type *Show countries which joined the UN after:*, then click the form outside the label. You'll see that the Text Box contains the word *Unbound*. This indicates that the text box is not connected (bound) to data in a field, as is the case with our data entry forms. In fact, the whole form is called an *unbound form* because it is not based on any existing data or data structure.
4. Repeat step 3 for the remaining two prompts, lining them up one beneath the other. Place the text *and joined before:* in the second prompt and *and whose population exceeds:* in the third prompt.
5. Each text box needs an easy-to-remember name so we can refer to it easily in our query, so right-click the first text box and choose Properties from the pop-up menu. This displays the Properties box, which lets you adjust every facet of your text box. Position the Properties box to the side, so you can see it and the Toolbox while also viewing your form.
6. In the Properties box, click the All tab and change the value of the Name property to *FirstJoined*.
7. Now do the same for the other two text boxes. Click the second text box (the Properties box will now reflect the settings for that box) and change the Name property to *LastJoined*. Click the third box and change its Name property to *Population*.

Save your form at this point giving it the name *Criteria*.

Querying the form

So far, all we've done is cosmetic. We've designed a rather boring looking form that will display three prompts. You can preview your form by clicking the View button on the toolbar. This form needs a Command Button that will tell it to run our date and population query, but before we can do that, we have to have a query to run!

To create a query that grabs its parameters from our custom form:

1. In the main database window, click the Queries button and double-click Create Query In Design View.
2. Select the Countries table and click Add to add it to the query, then close the Show Tables dialog.
3. Drag the fields Name and JoinedUN onto the query grid, then drag the Population field onto the grid to the right of the JoinedUN column.
4. In the Criteria cell of the JoinedUN column, type:

```
Between [Forms]![Criteria]![FirstJoined] AND
[Forms]![Criteria]![LastJoined]
```

Eek! I know this looks scary, but that *[Forms]!* *[Criteria]!* *[FirstJoined]* stuff simply tells Access to go check its collection of forms, locate one called Criteria, and then find the value entered into the FirstJoined text box on that form. You can think of it as the equivalent of telling Access the address where it can find the data, with each part of the address enclosed in square brackets and separated by an exclamation mark. Note that instead of using greater than (>) and less than (<) symbols, we've used the shorthand BETWEEN...AND operator, which you can use when searching for dates.

5. You should be able to guess what's coming next. In the Criteria cell of the Population column, type:

```
>[forms]![Criteria]![Population]
```

What type of data?

Because each answer is being entered into a *text* box on our form, we need to tell Access whether the input is a date, number or something else by identifying its data type. That way, Access will know how to deal with the data:

1. Right-click in the grey area above the query grid and choose Parameters from the pop-up menu.
2. In the first Parameter cell, type:

```
[Forms]![Criteria]![FirstJoined]
```

then tab over to the Data Type cell and select Date/

Time from the drop-down list.

3. In the second Parameter cell type:

[Forms]![Criteria]![LastJoined]

and select Date/Time data type.

4. In the third Parameter cell type:

[Forms]![Criteria]![Population]

and select Long Integer in the data type list.

5. Click OK and close the query, saving it with the name *UNPopulation*.

At your command

Now that we have our query, we're going to return to our form and put a Command Button on it that runs that query:

1. Open the Criteria form in Design view.
2. Click the Command Button tool in the toolbox then draw a rectangle about three centimetres wide and one centimetre high at the bottom of your form. The Command Button Wizard will be displayed. (If it isn't, it means you haven't enabled Control Wizards in the Toolbox. Select the button you just created and press the Delete key to erase it, then enable the Control Wizards and try again.)
3. In the Categories list select Miscellaneous and in the Actions list select Run Query, then click Next.
4. Select the UNPopulation query from the list and click Next.
5. Select the Text option, type *Display Matches* and click Next.
6. Name the button *RunQuery* and click Finish.
7. Save and close the form.

How'd we get here?

That's all the hard work out of the way. Let's summarise what we've done then take our customised parameter query for a spin:

1. We created a form containing a text box for each parameter in our query.
2. We gave each text box a recognisable name.
3. We created a query containing criteria which derive their parameters from the values typed into the text boxes on our custom form.
4. We assigned each parameter a data type to ensure Access knew how to deal with it.
5. We added to our form a command button which runs the query.

Rock and roll

Now it's time to run the query. In this case, we actually run the *form* which, in turn, calls the query via the command button:

1. Click Forms in the database window and double-click the Criteria form. (If the Properties box opens when you open the form, click its Close button to close it.)
2. Type the following values in the three boxes on the form:

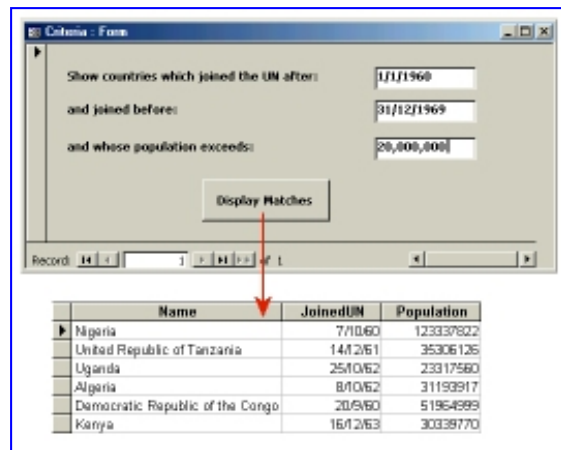
1/1/1960

31/12/1969

20,000,000

and click the Display Matches button. (Note, if you use a different date format on your PC, such as MM/DD/YY, adjust the second value accordingly.)

3. Access will display a table containing the six countries which joined the United Nations during the sixties and whose population is greater than 20 million.



Our custom dialog box and the results of running the query. Click the image to see a full-sized image and detailed caption.

Sample Data

The following links let you download a copy of the Countries table used in this tutorial. Choose the sample data appropriate for your version of Microsoft Access..

[Access 2000 sample](#)

[Access 97 sample](#)

© 2001 Rose Vines



Support geekgirl's

Do you find the tutorials on this site useful? If so, please show your support by kicking in a few bucks to sponsor an orphanage for Afghan refugees. For a small amount, it is possible to make a difference in an area of the world which is hurting badly.



Want to know more? Read this [post on my blog](#).



What's this box about?

[top](#)

[home](#)

[database menu](#)

Which database series should I read?

Not sure which series of database tutorials is right for you? Here's a quick description of their contents.

In the beginning there was a single series of database tutorials on this site. Now there are two. Which one should you read?

If you're looking for a nice, slow approach to learning databases, choose *Databasics*. It will help you learn the lingo, start with simple stuff, and eventually take on something a little more challenging. It's a series in development, so you can munch on some new information, go away and put it to use, then come back in a week or a month and try something new.

If, on the other hand, you want to dive right into the nitty gritty of database design, try the companion series of articles, *Databases from Scratch*. While there's some overlap between the two series, *Databases from Scratch* moves onto advanced topics, such as relational database design, more rapidly and restricts its topic matter to design issues. *Databasics* wanders farther afield, and includes (or will eventually include) tutorials on queries, reports, forms design, application development as well as basic design principles.