Modelling job

Where did you put that data? Where did you put that file? Mark Whitehorn looks at the pros and cons of database modelling.

Last month, we started to look at three "database models" you can employ:

1. Everything on a stand-alone PC;
2. PC front end with data on file server;
3. Client-server using a database server as the back end.

In this context it is useful to think of a database as having four different parts:

1. User interface section;
2. Data processing engine;
3. Conflict resolution section (to deal with conflicts introduced by multiple users accessing the data at the same time);
4. The data itself.

We looked, last month, at the pros and cons of the first model. As I wrote at the time, I will give size estimates where appropriate but please don't take them as gospel. (See last month's *Hands On Databases* for other qualifications).

PC front end with data on file server

If you have a need for multiple users to access the same data, then it is not beyond the realms of possibility that you already have a network. Given a network, you have the option of moving to our second database model.

In this model, much is left the same as before. You would still run Access, Paradox, or the RDBMS of your choice on your PC so the data processing engine bit of the database stays there.

Only two things change. One is that the data files are moved to a file server and the second is that the individual RDBMSs running on the individual PCs need to communicate with each other. They need to do this in order to resolve the multitude of potential conflicts which suddenly arise when more than one person accesses the same data simultaneously.

Now is not the time to go into all such conflicts but consider a simple example. You and I both work for the same company and we are trying to update the company's customer records. I open up the record for A Smith to increase his credit rating from £2,000 to £3,000. While I am doing so, you delete his record. What happens to his record when I finish editing it and send it back to the file server?

The answer to this question depends on the RDBMS you are using. Access, for instance, maintains a lock file in the same directory as the data file and this is used to store information about who is doing what at a particular time. Thus, if I had opened the record to update it before you tried to delete it you would receive a message saying that the record was in use by "Mark" and that you wouldn't be able to update the record until I had finished with it. Other RDBMSs use other mechanisms, some less efficient then others, for dealing with these potential conflicts.

The important point is that with this database model the control of the user interface and the data processing engine remain on the PC while the data and conflict resolution are moved to the file

Delphi Programming for Dummies

by Neil J. Rubenking

Delphi is a multi-purpose programming tool and is commonly used for the generation of user interfaces to databases. This book is one of a series and if you've seen any of the "** for Dummies*" books you'll already have a good idea of what this one is like.

It has lots of diagrams, some very silly cartoons and lots of icons in the margins to identify Technical Stuff, Warnings, Tips and so on. It is also written in a style which doesn't display the correct reverence for what is a serious programming tool — which is just fine by me. It's not that I don't respect Delphi but books without a sense of humour can be really tedious. This one is fun.

The database section is pretty skimpy, comprising a mere 22 of the 376 pages. It would be impossible to recommend this book to a database-naïve user who wanted to learn how to create databases in Delphi but that description doesn't fit most of the readers of this column. If you are happy that you understand the database end of things, this is an excellent introduction to Delphi.

One of the examples used in the book is a function called Hailstone. Given a seed number, this function generates a series of numbers which bounce up and down "like hailstones in storm winds" before converging on the number 1. Apparently there is no way of either predicting how long a given seed number will take to reach 1, or proving that all numbers will eventually reach 1. I was so intrigued that I modified it to run in Access and it is included in the main part of my column for your amusement (see *Hailstones*).

By the way, I saw one of this series in a bookshop in France: "*Windows pour les Nuls*"; a gift of a translation for any of the series connected with databases!

■ *Delphi Programming for Dummies*. IDG Books £18.99 (ISBN 1-56884-200-7).

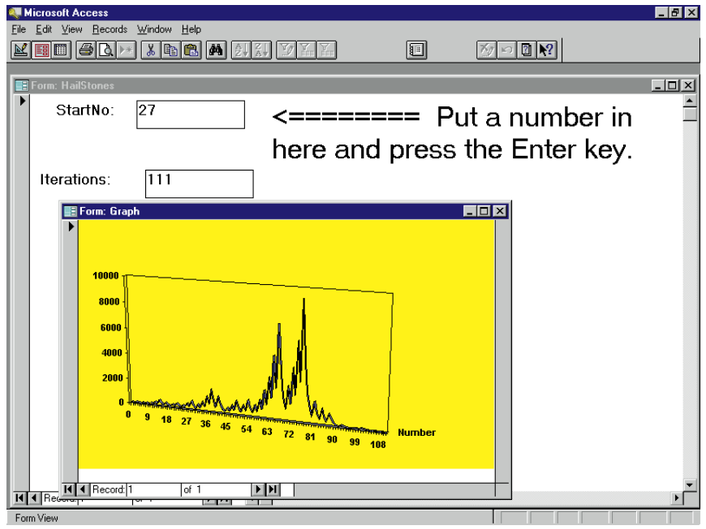


Fig 1 **The "Hailstones" algorithm in action**

server.

The big advantage of this model is that it provides multi-user access to the same data at relatively low cost. The big disadvantage is that the model is inefficient in two main ways. Firstly, it tends to load the network, soaking up bandwidth like a sponge. Remember that the data is at one end of the wire and the processing is at the other. Every time you query the data, it has to be moved to the client since that is where it is crunched. In a badly designed system this can mean that every query against a 100Mb table requires the entire table to be shipped to the client. Intelligent indexing can reduce this considerably (since the indexes can be shipped for searching and only the relevant records sent out to the client) but in practice, the effectiveness of this depends on the particular RDBMS.

Secondly, the processing is at the client end, so each client needs sufficient resources to cope with the data. If you decide that an increase in the database size warrants an increase in memory of 16Mb, you will need to add that to all the clients: given ten clients, that's 160Mb.

These restrictions mean that the number of simultaneous clients and the size of the data are relatively restricted. Think in terms of ten clients and 1Gb.

Client server model

This model is simply a modification of the previous one. The user interface stays on the PC: the data, the processing and the conflict resolution moves to a server. This is typically not a file server but a server dedicated to running applications like RDBMSs, hence they are generically known as application servers. Machines which are dedicated to running RDBMSs are often called database servers or SQL servers.

But wait. Why go to the expense of a dedicated database server when you have that nice NetWare 3.x file server already in place? Can't you run a database engine on that as well? The answer is that you can, but you probably don't want to. The reason lies in the NOS (network operating system).

NOSs like NetWare 3.x are optimised for File+Print. Application servers run a NOS which is optimised for running multi-user applications. This doesn't mean that you can't do it, just that performance will be compromised, so a dedicated database server is generally considered to be better. If you really want to run one server for both application and File+Print, then NT is probably better than NetWare, all things considered.

The Client-Server model is typically *not* limited by bandwidth. Since the processing and data are now snuggled together in one place, queries no longer mean that masses of data have to move across the network. Instead when the GUI, running on the client, is used to construct a question only an SQL description of that query is shipped across the network to the server.

This SQL will typically be a very short ASCII string. The database engine on the server processes the query and simply sends the answer (rather than the entire table) to the client. Conflict resolution is also handled centrally, with associated benefits in terms of speed and sophistication. Centralising the processing means that the whole system is easier and usually cheaper to update. If the database slows down you can throw hardware (memory and processors) at the server. You don't have to add it to the clients because they are simply handling the user interface.

On the ticklish subject of size, this model will typically support as many clients and as much data as you can afford. To put that another way, consider this model seriously if you think you will need more than ten clients or >1Gb of data in the foreseeable future.

Hailstones

The hailstones algorithm is very simple and most easily expressed in pseudo code.

```
Start with a seed number
Repeat
        If the number is even,
divide by two
       Else multiply by 3 and add 1
Until number = 1
```

I can offer no justification for implementing this in Access except that it is fun and that the behaviour is really wacky. Give it a number like 26 and it takes a mere ten iterations to get down to one. But 27 takes a monstrous 111 iterations, and 28 a more reasonable 18. A copy of this is in the MDB file on our cover-mounted disk this month. (See also, the *Delphi Programming for Dummies* panel on p285).

More on meters

The meter problem (which was discussed earlier this year) resulted in my publishing, on our *PCW* cover disk, a database with several of the solutions and a kit which allowed you to test any other solutions against the published ones.

I asked only for solutions which were genuinely faster. None have arrived but Paul Bloomfield, a supplier of one of the original solutions, contacted me to say that he had got very different answers from those published. It turned out that he hadn't got the *PCW* cover disk, so had built his own database for test purposes. I sent him the one I had used and this is his reply:

"Our speed differences are down to one factor: the data. My test database had 2,000 records from 200 meters (i.e. average ten readings per meter), whereas I see yours had only three meters and so 666 per meter, so the query is doing 66 times the work!

Perhaps there are lessons to be learnt here in problems of scaling. There is no such thing as good SQL per se*, the query must be written to fit the data, or the data structured to fit the query!"*

I agree. It is un-nerving to realise how many factors we need to consider when working with databases. Still, it keeps us all in employment!

Case-sensitive joins

Andrzej Glowinski writes: "I have recently started using MS ACCESS (Win 3.1) at work, developing applications in the areas of large medical (clinical) information resources. We also use an Oracle server and my systems work fine using this but as soon as I try to build stand-alone versions all hell breaks loose. This is because some ACCESS designer/implementer, in their wisdom, has forced all internal joins to be case insensitive — and there is NO MECHANISM for altering this behaviour!"

I replied to this as follows: "I hate to be contentious, but Access does provide a mechanism which allows joins to be case sensitive.

Assume that we have two tables: NAMES and ORDERS. Each table has a field called NAME which contains case sensitive data. We thus have two fields:

```
NAMES.Name
ORDERS.Name
```

which need to be joined.

Create a query containing both tables, join them within the query (not in the relationships editor) and run the query. Precisely as you suggest, the data in the tables is joined in a case insensitive manner.

Now add a field to the query:

```
CaseCompare: StrComp([NAMES].
[Name],[ORDERS].[Name],0)
```

This uses the function StrComp which can be rendered case sensitive by setting the third argument to be 0.

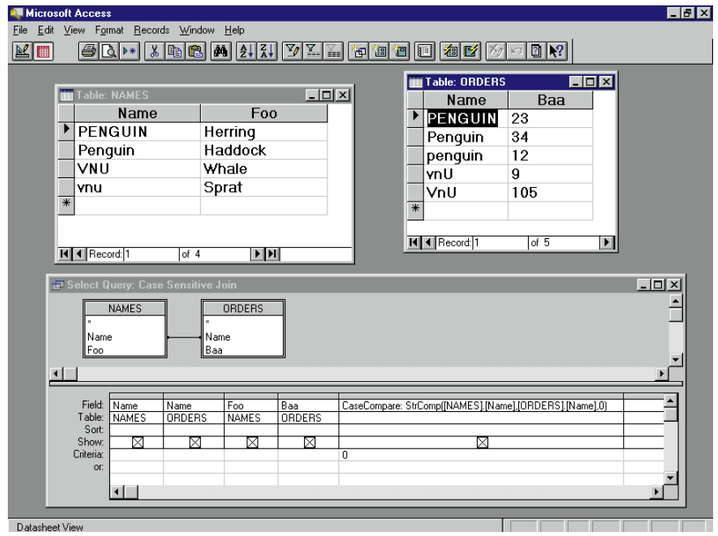
The function returns 0 if the strings match in case as well as letter order, so if you set the Criteria for that field to be 0 and re-run the query, the join is now case sensitive. The SQL version reads as:

```
SELECT DISTINCTROW NAMES.Name,
ORDERS.Name, NAMES.Foo, ORDERS.
Baa, StrComp([NAMES].[Name],
[ORDERS].[Name],0) AS CaseCompare
FROM NAMES INNER JOIN ORDERS ON
NAMES.Name = ORDERS.Name
WHERE (((StrComp([NAMES].[Name],
[ORDERS].[Name],0))=0));
```

I am aware that this doesn't allow you to sort case-sensitive material, although it would almost certainly be possible to write a function to do this. Rather more interestingly, it doesn't allow you to apply referential integrity or set a primary key on material which is case-sensitive (since Access treats 'PENGUIN' and 'Penguin' as equivalent in a primary key). However, it does at least allow you to perform the join." (A copy of this is available in the MDB file on the *PCW* cover-mounted disk).

To which I received Andrew's reply: *"Thanks for your message about ACCESS. We have explored the mechanism you propose quite extensively, both on local (native) ACCESS tables and on attached ones, primarily ORACLE. The performance hit is just unacceptable — I'm talking of joins across tables with 200,000++ items in them, so the total loss of indexing (which, effectively, is what you get) is very significant. You can mess around with the order of execution of the query but it is just too unpredictable to be of much use."*

I can't argue with any of this. Indeed, depending on how large the records are, this database may be getting to the size where a move to another model is inevitable. One question is, do any other PC-based RDBMSs have a better way of supporting



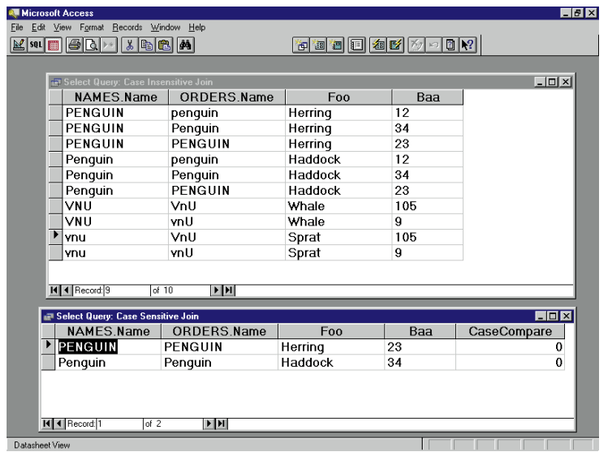


Fig 2 (above) The tables used to demonstrate the case sensitive join, and the query itself

Fig 3 (left) The results of a case insensitive (top) and case sensitive join (bottom)

PCW Contacts

Mark Whitehorn welcomes readers' correspondence and ideas for the Databases column. He's on **m.whitehorn@dundee.ac.uk**