Animal magic

Mark Whitehorn casts a beady eye over hierarchy in the zoo and shows a way of handling categorisation data. In addition, he passes on some improved solutions to past problems.

A reader called Andrew writes: "First of all, may I say how much I have enjoyed your column in *PCW*" *(ah, I love flattery — MW)*. "But unfortunately it has got me thinking about a problem which I have never satisfactorily resolved for myself." *(Sounds like work! — MW)*. "It involves a recursive link field in a table:

Field	Name	Type	Description
1	Code	Inc	Primary Key
2	Parent	LongInt	Parents Key
3	Category	String	Category Title

"The idea is to provide a hierarchy by using the table to emulate a tree structure which could be used for categorising animals, or departments in an organisation.

"I have been searching for an efficient way of finding all the categories that are below the current item. My best effort has been to run a query that finds all records whose parent is current code, add these to a result table and then query the answer to find all the records where there is a match on any of the codes. I keep this going until there are no records returned (when I've reached the bottom of the tree). Although this works, it does not seem very efficient. Do you have any ideas?"

What I have done is to look at ways of handling data from a table like the one Andrew describes. Fig 1 shows such a table. It describes a hierarchy which is used to name the animals in a zoo.

Each record simply represents a single link in the structure, so the data is stored in a reasonably economical manner. However, you need to look at several records to trace back up the structure.

So, for example, record 12 tells us that Harry's parent record is number three, which in turn tells us that Harry is a King. This in turn leads us to record two, and hence to record one, by which time we know that Harry is a King Penguin, which is a species of bird. The storage is economical, but a little unwieldy.

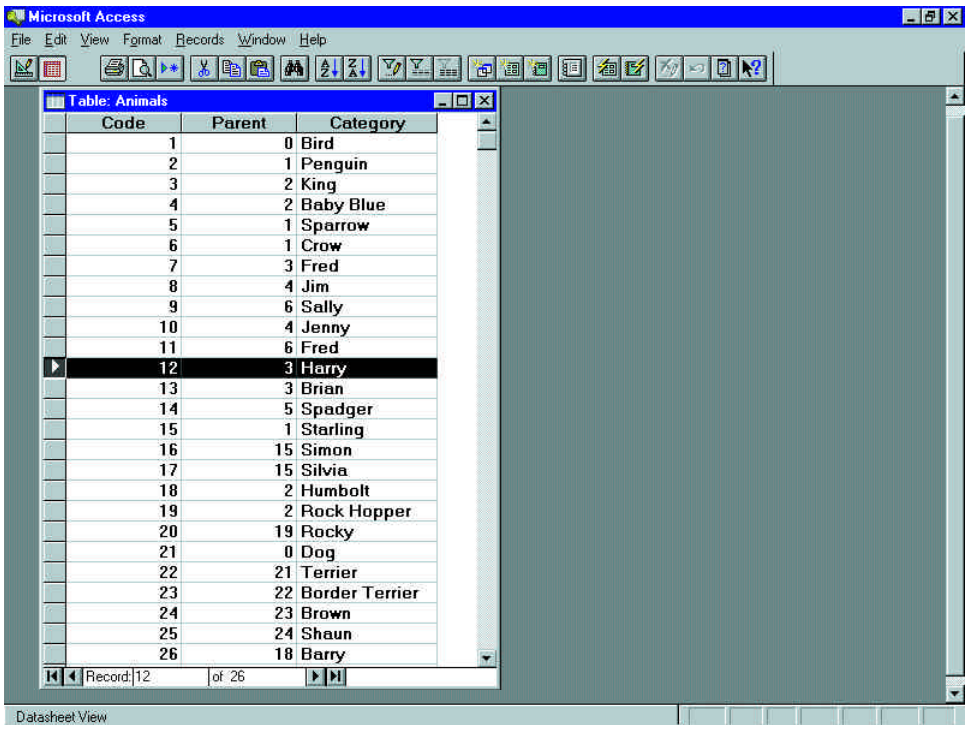
Fig 2 illustrates the information in a more legible state, and Fig 3 shows the query. Note that the answer table is not showing one record for each record in the base table. This is because it takes several records to define one "branch" of the tree. The SQL is a little tortuous, but essentially it is merely an expanded version of this:

```
SELECT DISTINCTROW Animals.Category
AS First, Animals_1.Category AS
Second
FROM Animals AS Animals_1
RIGHT JOIN Animals ON
Animals_1.Parent = Animals.Code
WHERE ((Animals.Parent=0))
ORDER BY Animals.Category,
Animals_1.Category;
```

which works for two levels of hierarchy.

This naming hierarchy is a little anarchistic in that names can appear at several levels. Suppose we have a more defined structure which must have, say, four levels. This will use the same base-table structure, but it provides a little more

Fig 1 Table of animals' names



p296 ➢

scope when it comes to defining forms. Two possibilities, one of which makes use of combo boxes, are shown in Fig 4, but neither is entirely satisfactory.

If anyone wants to try improving them, be my guest. The work so far is in the MDB file called DBCMAR97 on our CD-ROM.

Finally, it is possible to count those categories which exist under any given one, using a GROUP BY query based on the query (in this case called Project) which displays the data in the manner seen in the top right of Fig 4. So

```
SELECT DISTINCTROW Project.Group,
Count(Project.Priority) AS
CountOfPriority
FROM Project
GROUP BY Project.Group
ORDER BY Count(Project.Priority)
DESC;
```

produces the answer table for the data shown:

Group	CountOfPriority
Prototyping	4
Design	4
Repair	2
Year 2000	1
Modification	1
Consultancy	1

Rounding out

On to a little housekeeping. Over the past few months I've published several problems with solutions and asked for comments and/or improvements. This month, we'll tidy these up. Although most of the problems originate in Access, the solutions are usually applicable in most RDBMSs.

Last November I published an algorithm for rounding in Access. In last month's column I used several answers (of differing efficiency!) from readers. James Talbut came back with a modified version of his: "I'm probably too late, but anyway, I think I have fixed that rounding code:

```
Function Round(dNumber As Double,
iNumDigits As Integer) As Double
Dim dFactor As Double
Dim dTemp As Double
  dFactor = 10 ^ iNumDigits
  dTemp = dNumber * dFactor
  If 2 * dTemp = Int(2 * dTemp) And
Int(dTemp) <> dTemp Then dTemp =
dTemp + dTemp / 2 /
  Abs(dTemp) Round = (dTemp \ 1) /
dFactor
End Function
```

"The solution was simple. I just had to stare at it until my eyes ached. My fudge factor was either 1 or -1, depending on the sign of dTemp. Unfortunately, this was compounding the way in which Access alternates between rounding up or down. The solution was simply to fudge by 0.5 or -0.5. It's still hideous, for something as simple as a rounding function. But as a technical exercise I think it's kind of neat. And it's probably still quicker than converting to and from a string."

I agree. It's ridiculous that we have to write something like this just to round a number. Why doesn't Access simply include a rounding function? Microsoft, please note for future versions.

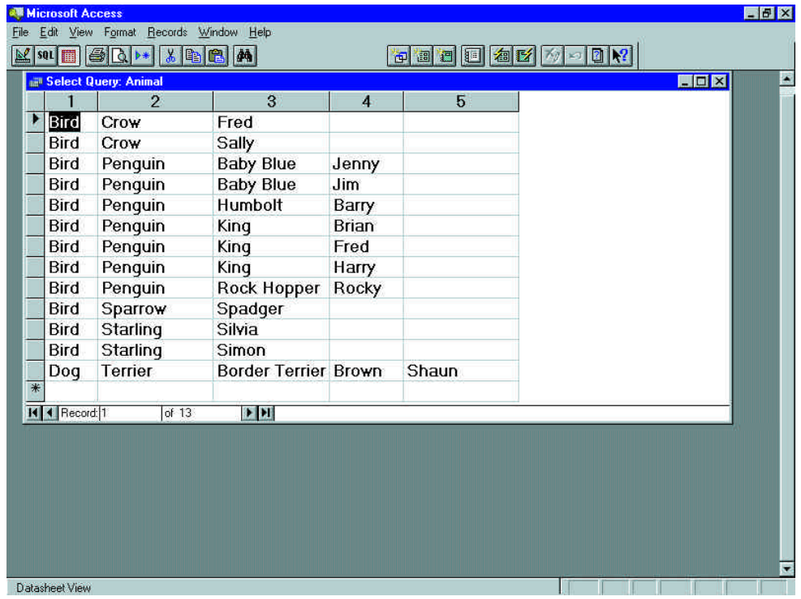


Fig 2 The data is stored in a reasonably economical manner

Case-sensitive joins

In last December's edition of my column, I published an email from Andrzej Glowinski who bemoaned the lack of case-sensitive joins in Access. I published a solution but he found it too slow, so I asked for other ideas as well as solutions in other RDBMSs. I was impressed with the intriguing answers, reproduced here.

On the subject of case sensitivity and the lost ninth data type, Stephen Parry writes: "There is another solution to Andrzej's problem with case sensitivity in joins, indices and sorting. It does, however, rely on a partially documented feature of access, data type 9 (aka BINARY). The BINARY field data type has been in Access since its earliest beginnings but has been omitted from key places in the package and its documentation throughout. Various system table fields use it and it appears as the data type for certain field types, in tables attached from other database products.

"It behaves much like TEXT in all respects except any comparison operations (joins, indexing, sorting) operate on ASCII code value, i.e. case sensitively. Very useful; but how do you create one? 'Not easily' is the answer to that. BINARY does not appear as an entry in the field type list in table design view. If you try to use DB_BINARY with the CreateField method in A.B. you get an error. Not very helpful. The way around this is to use a CREATE TABLE data definition query:

```
CREATE TABLE(MyField BINARY (30))
```

"This creates a table with a single field (column) called MyField of type BINARY and a length of 30. Type this into the SQL view of an empty data definition query, execute it and it will create such a table. Remember to refresh the table list in the database window to show the new table. You can then edit the table definition in table design view as per normal. You can even change the properties of the binary column(s). The data type column correctly indicates a field type of binary. You can change this, of course, but once changed you can't get it back!

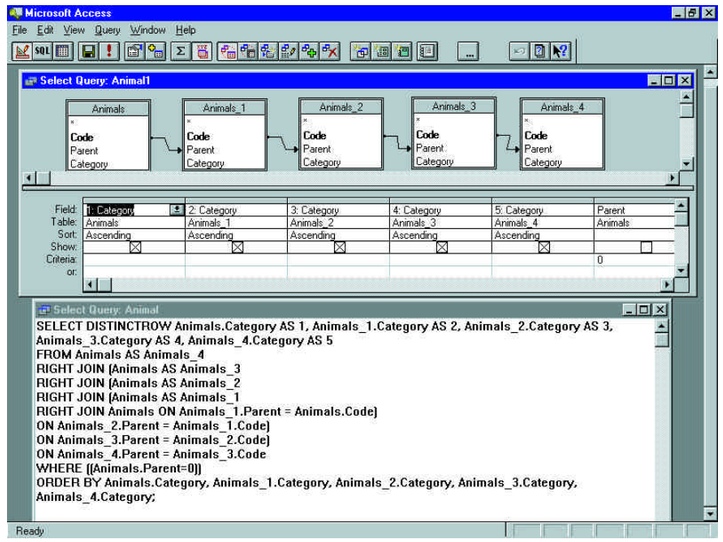


Fig 3 This SQL looks a bit complex, but it is just an expanded version of the code listing on p295

"To add a binary column to an existing table, I suspect an ALTER TABLE statement should work, although I have not tried it. This would allow you to add a new binary column to an existing table. You could then use code or query to copy the data from an existing column and then remove the original column. Thus, you have a long-winded means of changing a column's data type to binary.

"Of course, this all begs a fundamental question: why the (*beep!*) is something as useful as this so well hidden? I suspect that with sustained use, various 'holes' might be found in the functionality of this data type.

"Microsoft obviously intended a more complete but as yet unrealised solution to problems like this, but needed a quick solution to the Paradox and dBase connectivity problems, as well as a quick fix for some internals. For now, however, I will certainly continue to use it.

"By the way, I have noticed that in Access 7, nasty ol' Microsoft has encrypted its Wizards and other .MDA add-in files in this release. I have derived many useful programs from hacked versions of the Access 2.0 wizards, e.g. a database object directory comparison tool and a scripting tool based on the documentor, which just outputs a mammoth text file without generating a report or a temporary table. Hence, I was miffed to find that A7 has all its wizards compiled/encrypted in some way. Have you seen any way around this? I vaguely recall seeing an MSDN article on securing add-in code and possibly one giving the unsecured contents of certain wizards, but I have not been able to find either again."

I knew of the binary data type but hadn't thought of it as a solution to this problem. As to decrypting the Access 7.0 wizards, has anyone else found a way?

On the subject of case sensitivity, the following came from Neil Howie: "A bell rang at the back of my mind, so I set about creating a similar query in Paradox 7 but taking advantage of the fact that if you have maintained secondary indexes, you can set a case-sensitive option.

"Adding an auto primary index to each table and setting up secondaries on the Names fields let me set up the join in the query window and display the required result with the greatest of ease.

"For further investigation I took a long doc, converted it to text and wrote a little routine to extract the words to create two files, 12,000 records long. I modified the second to capitalise the last letter of nine words out of ten at random, then pushed these into Paradox. Because of repetitions, it produced 15,000 matches in 18 seconds, so I guess on your enquirer's 200,000++ records it would still not be a practical proposition if he had to do it too often.

"However, what is now worrying me is that executing the same query in Delphi (using Paradox's SQL) takes almost 50 seconds. What am I doing wrong?"

p298 ➢

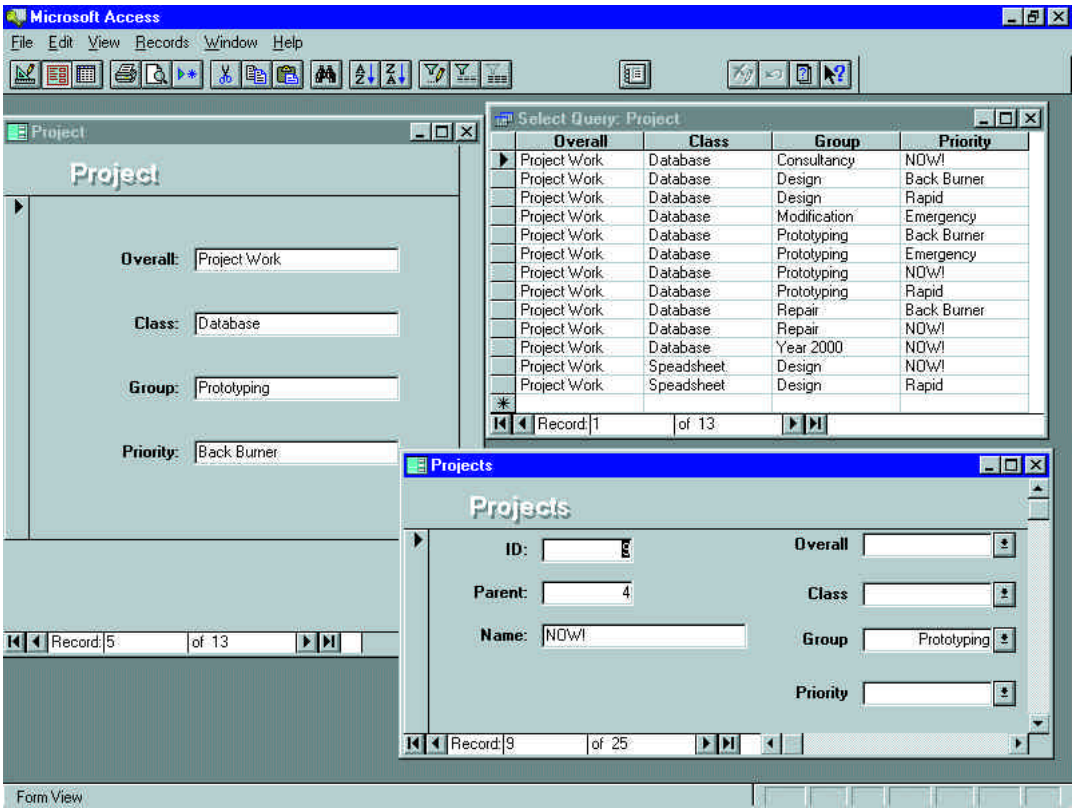


Fig 4 A defined structure with four levels provides more scope when defining forms

Making your mark

Last month I published a problem from Andy, a teacher who was using Access 2.0 to store his pupils' marks. He wanted not only to store the marks that his students achieved in their tests, but also their positions in the various class groups.

I suggested there was a conflict here between relational theory and expediency, and asked for suggestions and restraint (in the hope of avoiding a holy war). I am delighted to report that (nearly) everyone kept their heads and the majority of answers were helpful rather than religious. Paul Mapstone's answer (below) was the most complete. It is also applicable to any situation in which a rank order is required.

"My advice on this is not to store derived data (i.e. the Position column) unless you have to, for performance reasons. This is a good example. It is fairly straightforward to calculate the Position column in standard SQL using a correlated subquery as follows:

```
SELECT A.[Pupil ID], A.[Test ID],
A.Score, A.Position,
  (select Count(*) + 1
  from [Test Scores] AS B
  where B.[Test ID] = A.[Test ID]
and B.Score > A.Score ) AS Rank
FROM [Test Scores] AS A
WHERE A.Score is not null
ORDER BY Score DESC
```

"Column Rank in the above query should correctly return the required Position. This works because the Position is equal to the number of people who have a better position + 1. Alternatively, you can use the SQL'92 outer join syntax (which Access seems to partially support) as follows:

```
SELECT A.[Pupil ID], A.[Test ID],
A.Score, A.Position,
Count(B.Score) + 1 AS
Rank
FROM [Test Scores] AS A LEFT JOIN
[Test Scores] AS B
       ON A.[Test ID] = B.[Test
ID] and A.Score < B.Score
WHERE A.Score is not null
GROUP BY A.[Pupil ID], A.[Test
ID],
A.Score, A.Position
ORDER BY A.Score DESC
```

"We need the outer join, as the inner join will eliminate the top result. Either of the above queries could be used as the basis of any required reports etc, but if your teacher really wants to store the rank in the Position column (and suffer potential update anomalies), simply save one of the above queries with the name 'Rank query' and use it in the following UPDATE query:"

```
UPDATE [Test Scores]
SET Position = dlookup("Rank",
"Rank query",
      "[Pupil ID]=" & [Pupil ID] &
 " and [Test ID]="
& [Test ID])
WHERE Score is not null;
```

An excellent answer. Paul's first paragraph touches on the heart of the conflict. Storing derivable data usually has no benefits and several major disadvantages (such as causing potential update anomalies). However, storing such data can occasionally yield a major performance benefit. Of course, a real purist would never consider mere performance as a justification for breaking one of the central tenets of the relational model. Non-purists, on the other hand, wouldn't even hesitate. I sit uncomfortably on the fence, sticking to the purist line whenever possible and worrying every time expediency forces me to break what I know to be a sensible rule.

Paul's solutions are on this month's cover-mounted CD-ROM as PAUL.MDB and PAUL95.MDB. The first is an Access 2.0 version which crashes Access every time I try to run the update query. There cannot be anything fundamentally wrong with the solution because the problem does not occur in Access 95. So perhaps it is my machine?

PCW Contact

Mark Whitehorn welcomes readers' correspondence and ideas for the Databases column at database@pcw.vnu.co.uk