I design good databases, you design good databases, so there is no point in discussing bad design, because we never have to cope with it. Right?

Wrong, as an email from Mark Schaefer Schaefer@gunn-jcb.co.uk illustrates

His company sells a large number of items - a separate company answers the phones and takes the orders. Most of the items are VATable; a few aren't. The phone answerers structure their database with a single row for each telephone call and the items ordered are added to this row rather than to a separate table. That is, each row contains the customers info and then Item 1, Item 2, Item 3 .. .. Item 20 all in a single row. An additional table called ITEMS contains the vat / price / description and part number.

Mark has to generate two figures from this, the total value of the non-vat rated items and the total value of the vat rated items. The only way he can see of doing this is with a huge IF - THEN -ELSE statement along the lines of - If item 1 = (0RatedItem1 or 0RatedItem2 or .. .) then ThisRow0VatTotal = ThisRow0VatTotal + (QtyItem1 * ItemPrice)

This seemed like a good topic for the column because I suspect that many people have to cope with badly structured data and such data can be very difficult to query.

Possible solutions include:

- writing code to generate the answer that you want (as in, the big IF-Then-Else)
- Biting the bullet once and for all and converting the data to a normalised structure

The problem with the first solution is that every time you want to ask another question you will again have difficulties extracting the information. The second solution means that all further questions should be much easier to answer. So, how can we turn non-normalised data like this into normalised?

First we need to work out exactly what is wrong with the current data. Sample data is in table Orders in DBCFEB02a.MDB. I have simplified the data is a little in Order2, just to make the actual problem easier to visualise. People are ordering goods from Mark's company. Each record in the table contains information about the customer and every different item that they have ordered. If they order two different items then two of the available 20 columns will have information in them; if they order four then four columns will have data and so on. (One of the reasons why this table is poorly designed is that it limits the number of different items that a person can order.)

The original table, Orders, has three columns for each item - one for the item identifier, one for the item size (which only applies to certain items) and one for the number ordered. In direct contrast, a normalised structure would contain one table for the customers, one for the items (already provided), one for orders and a fourth for details of each order.

(screen shot 1)

OK, now we understand the problem, how do we solve it?

Customers

The first job is to create the Customer table. This is an easy one, something like:

```
SELECT ContactID, ContactFirstName, ContactLastName, ContactCardType,
ContactCardNumber
FROM Orders;
```

I've kept the number of fields small to reduce the size of the SQL statement, but you can add whichever fields you think are appropriate. If customers place multiple orders, you'll end up with multiple rows for some customers. If you add the word DISTINCT after the word SELECT then the SQL statement will remove duplicate entries from the answer table (see the query called Customers Unique). In practice, you may have to do more cleaning work because any mis-spelling in any of the fields that you extract will give duplicate entries for the same customers. Once you are happy with the result, add to the SQL so that it reads:

```
SELECT DISTINCT ContactID, ContactFirstName, ContactLastName,
ContactCardType, ContactCardNumber
INTO Customer
FROM Orders;
```

This creates a new table called Customer, the second of our four-table normalised structure. In the new table, make the ContactID field the primary key. (Screen shot 2).

Orders

We can create the Orders table in much the same way, extracting the information that's unique for the order itself from the current Orders table – for example:

```
SELECT OrderID, ContactID, ContactUser, ContactDate, OrderProcessed INTO
TblOrder
FROM Orders;
```

(For information, the table isn't called Order because that's a reserved word in SQL, so I've added Tbl as a prefix.) Make the field called OrderID the primary key.

OK, that's all the easy stuff, and we only have one more table to generate (screen shot 3). That's the one for the order details – the detailed information about the items that make up each order.

Order-details.

The easiest way is to explain how we do this is to split the problem into bite-sized chunks. If we write a query like this (query1):

```
SELECT OrderID, ContactItemCode1, ContactSize1, ContactQty1
FROM Orders;
```

then it returns:

OrderID	ContactItemCode1	ContactSize1	ContactQty1
1	JCB 46	L/XL 38" – 46"	1
2	JCB 116		1
3	JCB 46	L/XL 38" – 46"	2
4	JCB 34		1

In other words, we have extracted the OrderID and the Item details from the first of 20 possible entries. We can use a slightly altered version to pick up the 'second' item from each order (query2):

```
SELECT OrderID, ContactItemCode2, ContactSize2, ContactQty2
FROM Orders;
```

We get:

OrderID	ContactItemCode2	ContactSize2	ContactQty2
1	JCB 221		1
2	JCB 221		1
3	JCB 221		3
4	JCB 116		1

Sadly, the third query reveals a problem:

```
SELECT OrderID, ContactItemCode3, ContactSize3, ContactQty3
FROM Orders;
```

OrderID	ContactItemCode3	ContactSize3	ContactQty3
1			
2	JCB 34		1
3	JCB 116		4
4			

Oops. Orders 1 and 4 only contained two different items so they don't have any values in this third slot (or any of the other 17). We only want rows that contain values in the ContactItemCode field, so we'll just ask for rows where the value in ContactItemCode3 is not null – query 4.

```
SELECT OrderID, ContactItemCode3, ContactSize3, ContactQty3
FROM Orders
```

WHERE ContactItemCode3 Is Not Null;

I ran this query expecting it to work and was gob-smacked when it returned exactly the same set of data as query3. Strange. The rows in question look as if they contain no data, and seem to behave as if they contain no data but they don't contain null values. Which leaves us with a question.

Q What looks like a null, smells like a null, but isn't a null?

A A zero-length string.

Given that we are trying to solve the broader problem, I'll leave the explanation about the distinction between nulls and zero-length strings until next month; however, it's fair to say that you don't normally expect to find them in places like this. The good news is that you can get a query to filter out zero-length strings, for example query5:

```
SELECT OrderID, ContactItemCode3, ContactSize3, ContactQty3
FROM Orders
WHERE Not ContactItemCode3="";
```

or you could alter the last line to:

```
WHERE ContactItemCode3<>"";
```

Just to be really safe we might amend it to:

```
WHERE ContactItemCode3<>"" And ContactItemCode3 Is Not Null
```

to catch any nulls as well.

So, we can write all our 20 queries using this WHERE clause to make sure we don't get any meaningless rows in our Order-Details table. However, that will still leave us with 20 queries to run which is tedious. Happily, to our rescue comes the UNION query. This will essentially bolt answer tables together. So, if we use query 6:

```
SELECT OrderID, ContactItemCode1 as Item, ContactSize1 as Size, ContactQty1 as
Quantity
FROM Orders
WHERE ContactItemCode1<>"" And ContactItemCode1 Is Not Null

Union

SELECT OrderID, ContactItemCode2, ContactSize2, ContactQty2
FROM Orders
WHERE ContactItemCode2<>"" And ContactItemCode2 Is Not Null
```

And so on down to….

```
UNION SELECT OrderID, ContactItemCode20, ContactSize20, ContactQty20
FROM Orders
WHERE ContactItemCode20<>"" And ContactItemCode20 Is Not Null;
```

we get our final table (screen shot 4). Access isn't happy about this being a 'make table' query as well, so cut and paste the answer into the empty table (one that I prepared earlier using a toilet roll and some sticky-backed plastic) called OrderDetails. Once all that's been done, you can set referential integrity between the tables and you have a normalised database. It is left as an exercise for the reader to create the queries that Mark needs…

We have had some problems with the MDB files becoming corrupted on the CD-ROM, so the sample files are on my web site:
www.penguinsoft.co.uk

DBCFeb02a.mdb is the starting set of data
DBCFeb02b.mdb has the completed set of normalised data.

Screen shot1
A normalised structure which would hold the data more elegantly.

2
Two tables down, two to go.

3
Three down, only one to go!

4
The final table.