DBCJan02

One aspect of building databases that often gives people trouble is deciding what tables to create and what relationships to create between them – in other words, finding the correct design of database to support the application they have in mind. A good example is one I received recently from Peter Forty whose email address is the gloriously named "Forty Family" GothicHouse@ukgateway.net.

He and I exchanged several emails which are too long to detail here, but the basis of his initial question is on the CD-ROM as PETER.TXT for anyone who wants more detail about the problem. One reason we exchanged more than one email is that it is often difficult to describe an entire problem in one go. What often happens, particularly with more complex problems, is a form of iterative development. As you try to create a database to match a set of requirements, the very act of creating the database brings out into the open some exceptions that mean that the database structure has to be modified.

The ‘challenge’ of database design has, of course, been around for years and all sorts of mechanisms and techniques, such as ER (Entity Relationship) modelling, have evolved to help with the process. These should be, of course, the first tools that spring to mind when you get involved in database design. However, prototyping is also a valuable addition to your armoury.

What do I mean by prototyping? Well, suppose that you are getting hopelessly bogged down in detail when trying to design a database. Instead of trying to fight the fact that creating the database will highlight exceptions, you get this fact to work for you. Take a step back, forget the detail for a moment and rough out a quick prototype that solves the basic problem. Then see where it fails to cope with some more complex aspect of the problem, and modify it. Then add some more detail and modify it again. Repeat this process until the design is capable of dealing with all of the problems that you throw its way.

As a further bonus, at least from my perspective in this column, is that this iterative form of database can be used to illustrate how the very structure of a database can be made to implement some of the business rules of the organisation.

This may sound all to abstract, so let’s have a look at an example. This is loosely based on Peter’s question but I have adapted his problem for illustrative purposes.

Suppose that you collect items such as stamps. Sometimes you buy single stamps, sometimes you buy collections that consist of individual stamps. Sometimes you buy individual stamps and then later assemble them into collections. So a stamp may be a member of a collection when you buy it, or not. It may later become a member of a collection, or not. Each stamp might be unique within your collection, or you might have multiple copies of the same stamp. Hmmm, sounds like a good case for iterative development. We’ll start with the simple case – two tables, one for collections and one for stamps (Screen shot 1).

This database is available on the CD-ROM as dbcJan02a.mdb. My apologies to philatelists, I know nothing about stamps, but that's OK, we're more interested in the structure. This one allows me to define collections and to assign stamps to each collection and the structure itself imposes some business rules.

A I can have a collection which contains no stamps e.g. Collection 3
B I can have a stamp that doesn't belong to a collection e.g. Stamp 2

Suppose that I only ever buy stamps as collections, and so I want the second business rule to be "I **can't** have a stamp that doesn't belong to a collection".

No problem, I simply amend the structure of the second table and make the value in the field Stamps.CollectionID a required one (Screen shot 2). At the same time I would remove the default value of zero. If (unlikely, but possible) it is a business rule that the default collection is, say, number 4, then amend this value to suit.

Do I also want to set the database so that I can't have a collection with no stamps? The answer is probably 'no', because if I set both of these rules up, I will find it very difficult to enter a new collection. (I can't enter a new stamp without a collection to place it in, and I can't enter a new collection until it has stamps!).

Q So far, so good; but suppose that I do buy stamps singly (i.e. I buy stamps that don't belong to a collection)?
A Well, you just remove the requirement for the value in Stamps.CollectionID. This field will remain a foreign key, meaning that any value entered in here will have to point to an existing collection. However, this change means that you aren't **required** to assign any particular stamp to any particular collection.

Q. OK, but that doesn't allow me to record information about stamps that are bought singly, such as where they were bought, price paid etc. Currently that information is in the Collection table. If I move it into the Stamp table, then it works fine for single stamps but very badly for stamps bought as part of a collection because a lot of information would be duplicated.
A. Fine, no problem, what you need is a structure as shown in dbcJan01b.mdb. (Screen shot 3)

Here we still have a table for collections and stamps, as before, but look at the information about stamps. I have split it so that the information which is appropriate for an individually bought stamp is in a new table – the date upon which it was bought, the price paid and so on. The information that needs to be stored about all stamps, irrespective of how I came to own them (face value, date of first issue etc.) is still in the original Stamp table. I have done essentially the same for Collections: pulled out the information that is appropriate for a collection that was actually purchased. This allows me to have collections that were never purchased, in other words, which were put together from singletons.

So, what business rules am I supporting with this structure? Well, for a start, the one about being able to buy stamps singly and also the one about being able to build collections up from singletons. A stamp doesn't have to belong to a collection, but if it does, that collection has to exist. We manage this one with Stamps.CollectionId set to not required (see above) and referential integrity set between Collection and Stamps (screen shot 5).

One relatively unusual point of this database is that it makes use of one-to-one relationships between two sets of tables, CollectionDetail-Collection and Stamps-StampDetail. I want to enforce referential integrity between these. For example, in the case of Stamps-StampDetail, if I have an entry in Stamps-Detail it must also exist in Stamps. This is another way of saying that Stamps is the definitive list of all the stamps I own, some of which were purchased separately and hence are listed both in Stamps and StampDetail.

Most databases will support this, Access included, but it is somewhat picky about how you set it up. You need to set up the relationship in the usual way, but be sure that you drag **from** the table that is the parent **to** the table that is the child e.g. from the Stamps table to the StampDetail table. You can play around with the different relationships that can be created and see the effects that each has on the business rules. For example, you can set up one-to-one relationships where referential integrity is not enforced.

One of the advantages of using a database like this is that I can build queries very easily that pull the data together in a readable way. For example the query:

```
SELECT Collection.CollectionID, Collection.Description, CollectionDetail.PricePaid,
CollectionDetail.[Date Bought]
FROM CollectionDetail
RIGHT JOIN Collection ON CollectionDetail.CollectionID = Collection.CollectionID;
```

Should give you:

CollectionID	Description	PricePaid	Date Bought
1	My First which contains 2 stamps, bought as a collection	£34,534.00	05/07/1989
2	Second collection, not bought, but built up from single purchases		

Which is all the information relating to collections.

While something like:

```
SELECT Stamps.StampID, Stamps.CollectionID, Stamps.FaceValue,
Stamps.DataFirstIssue, StampDetail.[Date Bought], StampDetail.PricePaid,
StampDetail.Notes
FROM StampDetail
RIGHT JOIN Stamps ON StampDetail.StampID = Stamps.StampID;
```

Gives a more stamp-centric view of the world. Finally the rather more verbose:

```
SELECT Stamps.StampID, StampDetail.[Date Bought], StampDetail.PricePaid,
StampDetail.Notes, Stamps.CollectionID, Stamps.FaceValue, Stamps.DataFirstIssue,
Collection.Description, CollectionDetail.[Date Bought], CollectionDetail.PricePaid
FROM CollectionDetail
RIGHT JOIN (Collection
RIGHT JOIN (StampDetail
RIGHT JOIN Stamps
ON StampDetail.StampID = Stamps.StampID)
ON Collection.CollectionID = Stamps.CollectionID)
ON CollectionDetail.CollectionID = Collection.CollectionID;
```

gives an overall view of the data. These three queries are provided with the sample databases.

Now suppose that I find that I have stamps (such as the Penny Black) that turn up again and again in my collection. The data relevant to this stamp (Face value, dateof first issue) will keep on cropping up again and again and will be repeated in the database. As we all know, repeated data is an anathema in a database. So, the question to ponder for next time is, how would you alter the database to take this new development into account?

Screen shots.
1 The first cut of the database.
2 altering the properties of Stamps.CollectionID
3 The next cut of the database.
4 setting up the one-to-many relationship.

****
The text below is to go into PETER.TXT on the CD-ROM.
****
I want to record my collection in a database. So how do I structure the database and how do I query it? It seems obvious to place the limited set of individual objects in one table, the limited set of recognised combinations of these objects in a second, with the complete collection in a third.

Taking the stamp collection example the first table would contain an entry for each unique stamp, its face value, date of first issue, date of last issue, description etc etc, and of course would be given a primary key. The second table would contain an entry for each recognised combination, with a description and the primary keys for each object in the combination from the first table (if the combinations are of variable size a further table with a record for each object in the combination might be preferred but I think the basic issue remains).

The last table wants to contain an entry for each object or recognised combination collected, when it was collected, how much it cost, approximate value, comments etc.

So what is the problem? Well the third table is not very elegant.

A record in this table points either to an individual object in the first table or to a recognised combination in the second table. These end up as separate fields with only one of them valid and another Boolean field to say which is valid.

However the only alternative I can see is also unattractive and that is to list all objects in the second table as combinations of just one object each, so that the third table need only refer to the second.