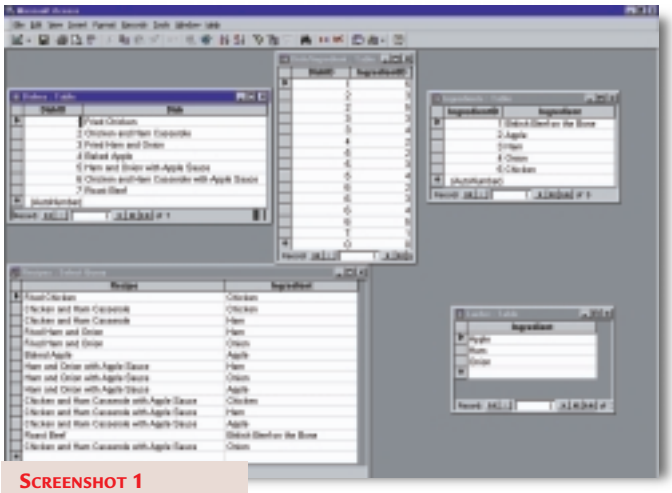
Ready, steady, database

Not even the sacred contents of Mark Whitehorn's larder are safe from his hungry database.

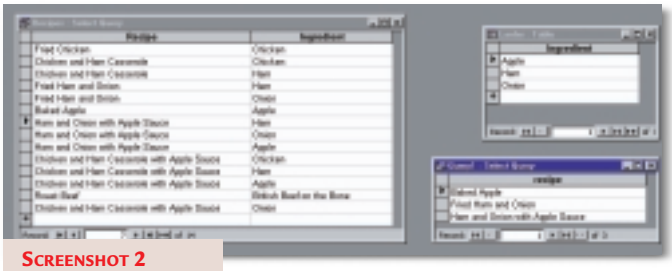
A very long time ago, when I had just started playing with databases, my charming wife Mary and I thought about creating a database which stored recipes, including the ingredients. The idea was that we would be able to input the contents of the larder and the database would list the possible dishes that could be produced. We were young, inexperienced and using (if memory serves) dBASE. We failed. Very occasionally since then the problem has surfaced briefly but I haven't pursued it. Now I don't have to because Ken Sheridan sent in a solution.

One of Mary's many skills is cooking (mine is simply eating) but you may not be interested in recipes. Fair enough, but you may be interested in selecting which engines can be built up from a given list of components, or which courses can be run, given a set of tutors with certain skill sets. In other words, this class of solution is widely applicable.



SCREENSHOT 1
THE TABLE USED FOR COOKING UP THE RECIPES

Imagine that you have the tables set up as shown in screenshot 1. The top three contain the data about the recipes and the information contained in them is redundantly summarised in the query called Recipes shown in the lower half of the screen. There is also another base table called Larder which shows the food available. This table really ought to consist of a set of pointers to the table of ingredients, but making it so simply clutters up the screenshot even more, so I have sacrificed elegance of design for readability.



SCREENSHOT 2
SO, WE CAN FIND THE DISHES THAT CAN BE PREPARED, BUT 'HOW DO THEY DO THAT?'

The SQL that Ken sent in (Fig 1) is deceptively simple and produces the results shown in screenshot 2. The result is correct: given Apple Ham and Onion we can make Baked Apple, Fried Ham and Onion and Ham and Onion with Apple Sauce.

The intriguing question is, how is it working? Experience suggests that simply writing 'This is left as an exercise for the reader' results in a great deal of mail; nevertheless I am of the opinion that you learn more about SQL from trying to work it out. So give it a try ...and read on if necessary.

One way of working out how SQL is performing its magic is to dissect the statement into parts and see how each bit functions alone. This usually doesn't work for all of the parts since they are designed to work together, but it is still a very useful technique.

The first part:

```
SELECT R1.recipe FROM ↙
Recipes AS R1
INNER JOIN Larder AS L1 ON ↙
R1.ingredient = L1.ingredient
```

Can be simplified slightly (if required) to:

```
SELECT Recipes.Recipe FROM ↙
Recipes
INNER JOIN Larder ON ↙
Recipes.Ingredient = ↙
Larder.Ingredient;
```

Either version returns this:

Recipe
Chicken and Ham Casserole
Fried Ham and Onion
Fried Ham and Onion
Baked Apple
Ham and Onion with Apple Sauce
Ham and Onion with Apple Sauce
Ham and Onion with Apple Sauce
Chicken and Ham Casserole with Apple Sauce
Chicken and Ham Casserole with Apple Sauce
Chicken and Ham Casserole with Apple Sauce

Each dish appears once for each occurrence of one of its ingredients in the larder. By that I mean:

FIG 1

```
SELECT R1.recipe FROM Recipes AS R1
INNER JOIN Larder AS L1 ON R1.ingredient = ↙
L1.ingredient
GROUP BY R1.recipe
HAVING Count(R1.ingredient)=(SELECT ↙
COUNT(*)
FROM Recipes As R2
WHERE R2.recipe = R1.recipe);
```

(*Key: ↙ code string continues*)

FIG 2

```
SELECT R1.recipe
FROM Recipes AS R1
LEFT OUTER JOIN Larder AS L1 ON ↙
R1.ingredient = L1.ingredient
GROUP BY R1.recipe
HAVING COUNT(R1.ingredient) = (SELECT ↙
COUNT(ingredient) FROM Larder)
AND COUNT(L1.ingredient) = (SELECT ↙
COUNT(ingredient) FROM Larder);
```

(*Key:* ↙ *code string continues*)

Fried Ham and Onion appears twice because the recipe requires both Ham and Onion and both are in the larder.

Baked Apple appears once because the sole ingredient is in the larder.

Chicken and Ham Casserole appears once because one of the two ingredients is in the larder.

Roast Beef doesn't appear because the ingredient is not in the larder.

In other words, this part of the query finds all of the dishes which have one or more ingredients in the larder.

The next part of the SQL applies a GroupBy

```
SELECT Recipes.Recipe
FROM Recipes
INNER JOIN Larder ON ↙
Recipes.Ingredient = ↙
Larder.Ingredient
GROUP BY Recipes.recipe
```

This produces:

Recipe
Baked Apple
Chicken and Ham Casserole
Chicken and Ham Casserole with Apple Sauce
Fried Ham and Onion
Ham and Onion with Apple Sauce

This simply ensures that each of the selected recipes is shown only once. So, we now have the name of each recipe that uses one or more of the ingredients in the larder. The trouble is that we can't actually make all of these dishes. We don't, for example, have any chicken; so attempts to produce Chicken and Ham Casserole are doomed.

We need to reduce this list still further. How can we ensure that each dish has not only some of the required ingredients, but all? Well, consider this variation on the query above.

```
SELECT Recipes.Recipe, ↙
Count(Recipes.Ingredient) ↙
AS CountOfIngredient
FROM Recipes
INNER JOIN Larder ON ↙
Recipes.Ingredient = ↙
Larder.Ingredient
GROUP BY Recipes.Recipe;
```

All I have added is a count of the ingredients for each dish which are currently in the larder. The answer table looks like this:

Recipe	CountOfIngredient
Baked Apple	1
Chicken and Ham Casserole	2
Chicken and Ham Casserole with Apple Sauce	4
Fried Chicken	1
Fried Ham and Onion	2
Ham and Onion with Apple Sauce	3
Roast Beef	1

Now, it so happens that we can also find out how many ingredients are required for each dish.

```
SELECT Recipe, ↙
Count(Ingredient) AS ↙
CountOfIngredient
FROM Recipes
GROUP BY Recipe;
```

Recipe	CountOfIngredient
Baked Apple	1
Chicken and Ham Casserole	1
Chicken and Ham Casserole with Apple Sauce	3
Fried Ham and Onion	2
Ham and Onion with Apple Sauce	3

Ah ha! Given the two tables above you can see where this is going. If we take from the list of recipes in the first table only those where the CountOfIngredients matches the value in the second table, we have the list of dishes that can be prepared. So, Baked Apple needs one ingredient, and there is one in the larder, so we can build that dish. The casserole is a no no because we have only one ingredient and need two.

Ken's SQL statement wraps all of this up into one elegant statement. As a further variation he adds: 'A variation on it would be to use exact division, ie returning only recipes where the ingredients exactly match those in the larder, Ham and Onion with Apple Sauce in the above case.'

The intrepid among you can try to work it out. The solution is as shown in Fig 2 above.

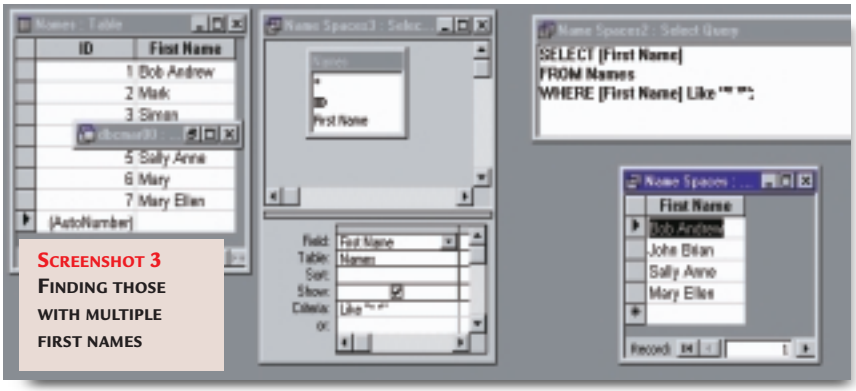
What's in a name?

Hiram Morgan writes to ask if, when you have a column with two forenames in some of the First Name fields, eg Mark John, is there a criterion to select the records containing the double names?

The answer is to use Like "* *" as the criterion, as shown in screenshot 3.

Access nulls

Finally, on the subject of finding nulls in Access, several people have pointed out that Access has an equivalent function to Oracle's NVL(), which is Nz(). Interestingly this doesn't appear in earlier versions of Access, for example Access 2.0, but had appeared by 97. Anyone know about 95?



SCREENSHOT 3
FINDING THOSE WITH MULTIPLE FIRST NAMES

PCW CONTACTS

Mark Whitehorn welcomes your feedback on the Databases column. Contact him via the PCW editorial office, or email: database@pcw.co.uk
The MDB file for this and previous months' columns can be found at www.penguinsoft.co.uk