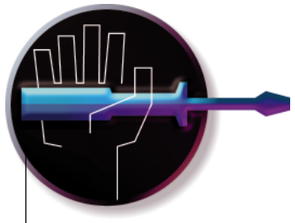


Group therapy

Having dealt with the basics, Mark Whitehorn delves deeper into SQL and shows you how to organise your records logically, in part II of our four-part tutorial.

Last month, we looked at the basic building-blocks of SQL and the ways in which they can be put together to elicit information from a database. With those commands alone you could pose an almost infinite series of queries, but SQL still has a whole range of tricks up its sleeve.

(*Last month's sample tables reappear in the screen shots here, so you won't have to fight with two magazines at once).*

Built-in Functions

SQL includes several simple statistical functions:

Function	
SUM	Total
COUNT	The number of occurrences
AVG	Average
MIN	Minimum
MAX	Maximum

Thus, it is possible (although not normal practice) to write SQL statements such as:

```
SELECT SUM(Amount)
FROM SALES;
```

Some systems will actually accept this. Access, for instance, generates a "dummy" field name (Expr1000) and yields the following table:

Expr1000
£5,117.57

It is common to explicitly name the field into which to place the output. For example:

```
SELECT SUM(Amount) "Sum of Amount"
FROM SALES;
```

or:

```
SELECT SUM(Amount) AS SumOfAmount
FROM SALES;
```

or even:

```
SELECT DISTINCTROW
SUM(SALES.Amount) AS SumOfAmount
FROM SALES;
```

which is how it appears in the Access dialect of SQL. All three of the above yield a table like this:

SumOfAmount
£5,117.57

The AS followed by a field name simply tells the SQL statement to put the data into a field of that name in the answer table.

It is permissible to mix two or more functions, for example:

```
SELECT SUM(Amount) AS SumOfAmount,
COUNT(Amount) AS CountOfAmount,
AVG(Amount) AS AvgOfAmount,
MIN(Amount) AS MinOfAmount,
MAX(Amount) AS MaxOfAmount
FROM SALES;
```

which yields the table shown in Fig 1.

It's also perfectly permissible to mix fields like this:

```
SELECT COUNT(Customer) AS
 CountOfCustomer,
AVG(Amount) AS AvgOfAmount
FROM SALES;
```

giving:

CountOfCustomer	AvgOfAmount
7	£731.08

These functions will even operate on fields which contain no data. If we amend the base table (for the sake of this example only) to be as shown in Fig 2, then the SQL statement above will give:

CountOfCustomer	AvgOfAmount
5	£282.76

The COUNT function finds only five values and AVG sums the values that it finds and then divides the result by six (i.e.the number of values in that particular field) rather than seven (the number of records).

However, these functions are designed to yield only a single figure each. Thus, SQL statements such as:

```
SELECT Customer,
AVG(SALES.Amount) AS AvgOfAmount
FROM SALES;
```

are illegal because SELECT Customer can (and in this case, would) have an output consisting of multiple records, while the second:

```
SELECT AVG(SALES.Amount) AS
AvgOfAmount
```

can only have an output of a single record.

Several SQL implementations provide more than the basic functions.

For example, Access also provides:

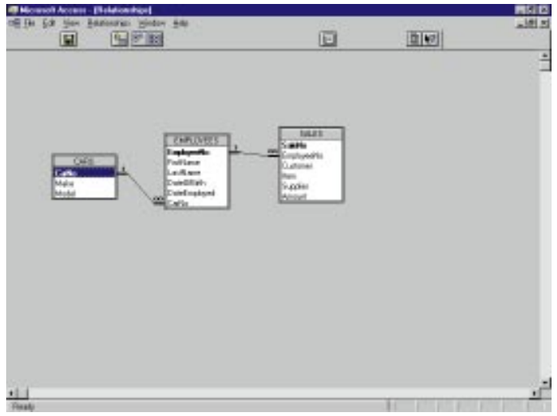
Function	
StDev	Standard Deviation
Var	Variance

Fig 1

SumOfAmount	CountOfAmount	AvgOfAmount	MinOfAmount	MaxOfAmount
£5,117.57	7	£731.08	£82.78	£3,421.00

Fig 2

SaleNo	EmployeeNo	Customer	Item	Supplier	Amount
1	1	Simpson	Sofa	Harrison	£235.67
2	1	Johnson	Chair	Harrison	£453.78
3	2		Stool	Ford	£82.78
4	2	Jones	Suite	Harrison	
5	3	Smith	Sofa	Harrison	£235.67
6	1		Sofa	Harrison	£235.67
7	1	Jones	Bed	Ford	£453.00



The relationship editor, showing the joins between the tables

It is just this kind of variation from the standard which demonstrates that SQL is still a fairly fluid standard.

GROUP BY — collecting information

So far, our generic SELECT statement looks like this:

```
SELECT field name(s)
FROM table name
WHERE condition(s)
ORDER BY field name(s)
```

We can expand it with:

```
SELECT field name(s)
FROM table name
WHERE condition(s)
GROUP BY Field name(s)
ORDER BY field name(s)
```

Last month we looked at the command ORDER BY, which provides a way of presenting information in ascending or descending order. Further control over your answer data is given by GROUP BY. The syntax is:

```
GROUP BY Field name(s)
```

To illustrate its usefulness, we'll consider the simple statement below:

```
SELECT AVG(Amount) AS AvgOfAmount
FROM SALES;
```

AvgOfAmount
£731.08

This averages the values found in the [Amount] field for all records in the SALES table. Suppose you want to examine the records which refer to customer "Simpson"? You'd use WHERE, as follows:-

```
SELECT AVG(Amount) AS AvgOfAmount
FROM SALES
WHERE Customer = "Simpson";
```

AvgOfAmount
£235.67

Now, suppose you want to do this for each customer. An inelegant, brute-force solution would be to run the query multiple times, once each for each customer. A clever solution is to get the SQL statement to group the records together by the name of the customer and then apply the AVG function to the values in the groups.

We can visualise the process as follows: going from the data shown in Fig 3, to that shown in Fig 4; and then to this, which is a full but compact summary of the required information:

Customer	AvgOfAmount
Johnson	£453.78
Jones	£1,937.00
Simpson	£235.67
Smith	£159.23

The SQL statement required to perform this magic is impressive:

```
SELECT Customer, AVG(Amount) AS
AvgOfAmount
FROM SALES
GROUP BY Customer
Order BY Customer;
```

The GROUP BY clause can be used more simply than this. For example:

```
SELECT Customer
FROM SALES
GROUP BY Customer;
```

produces:

Customer
Johnson
Jones
Simpson
Smith

At first, it appears that this is the same as:

```
SELECT DISTINCT Customer
FROM SALES;
```

which yields the same answer table, but adding another field demonstrates the difference. Thus:

```
SELECT DISTINCT Customer, Amount
FROM SALES;
```

produces:

Customer	Amount
Johnson	£453.78
Jones	£453.00
Jones	£3,421.00
Simpson	£235.67
Smith	£82.78
Smith	£235.67

whereas:

```
SELECT Customer, Amount
FROM SALES
GROUP BY Customer;
```

fails to run. Why? To answer this, we must look at what the SQL clauses are trying to achieve. The command:

```
SELECT Customer
FROM SALES
GROUP BY Customer;
```

essentially says "Sort the records in the SALES table so that identical values in the Customer field are together. Then 'crush together' the records with identical Customer values so that they appear to be one record." Thus:

```
SELECT Customer, Amount
FROM SALES
GROUP BY Customer;
```

Fig 3

SaleNo	EmployeeNo	Customer	Item	Supplier	Amount
1	1	Simpson	Sofa	Harrison	£235.67
2	1	Johnson	Chair	Harrison	£453.78
3	2	Smith	Stool	Ford	£82.78
4	2	Jones	Suite	Harrison	£3,421.00
5	3	Smith	Sofa	Harrison	£235.67
6	1	Simpson	Sofa	Harrison	£235.67
7	1	Jones	Bed	Ford	£453.00

Fig 4

SaleNo	EmployeeNo	Customer	Item	Supplier	Amount
2	1	Johnson	Chair	Harrison	£453.78
7	1	Jones	Bed	Ford	£453.00
4	2	Jones	Suite	Harrison	£3,421.00
6	1	Simpson	Sofa	Harrison	£235.67
1	1	Simpson	Sofa	Harrison	£235.67
5	3	Smith	Sofa	Harrison	£235.67
3	2	Smith	Stool	Ford	£82.78

p248

fails because there's a conflict (real in this case, potential in others) between the number of records that should be output.

```
SELECT Customer
FROM SALES
GROUP BY Customer;
```

will output four records:

Customer
Johnson
Jones
Simpson
Smith

while:

```
SELECT  Amount
FROM SALES;
```

will output seven records:

Amount
£235.67
£453.78
£82.78
£3,421.00
£235.67
£235.67
£453.00

Combining these two incompatible requests is impossible and SQL engines will refuse the statement. As you can see from the above, there is no obligation to combine GROUP BY with one or more of the functions. However, it is commonly done because we often only want to group records in order to be able to perform some type of manipulation on selections of records. It is perfectly possible to GROUP BY more than one field.

Thus:

```
SELECT Customer, Supplier,
 AVG(Amount) AS AvgOfAmount
FROM SALES
GROUP BY Customer, Supplier;
```

produces more groups than the SQL statement above that grouped by one field, because it is grouping those records which share the same value in Customer and Supplier. The answer table is this:

Customer	Supplier	AvgOfAmount
Johnson	Harrison	£453.78
Jones	Ford	£453.00
Jones	Harrison	£3,421.00
Simpson	Harrison	£235.67
Smith	Ford	£82.78
Smith	Harrison	£235.67

which raises another interesting question: how can you tell how many records are actually contributing to each group? One answer (but by no means the only one) is:

```
SELECT Count(*) AS NumberInGroup,
Customer, Supplier, AVG(Amount) AS
AvgOfAmount
FROM SALES
GROUP BY Customer, Supplier;
```

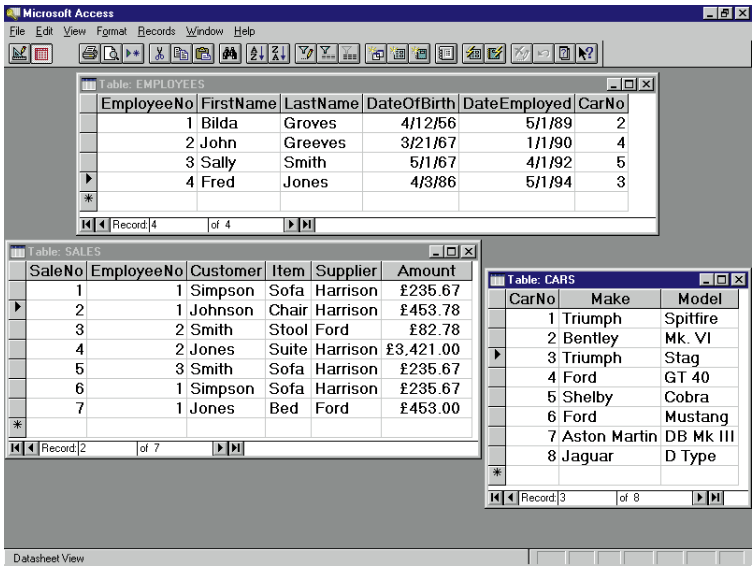
The only addition is the "Count(*) AS NumberInGroup" bit which simply says that the number of records in each group should be counted (Fig 5).

We could equally well use:

```
SELECT Count(Customer) AS
NumberInGroup, Customer, Supplier,
AVG(Amount)
AS AvgOfAmount
FROM SALES
GROUP BY Customer, Supplier;
```

which returns the same answer table.

GROUP BY is an incredibly powerful tool



The tables used in my examples

and it can be made even more so with the addition of HAVING.

GROUP BY and HAVING — Collecting information together

Whereas the GROUP BY clause puts records into logical groupings, the HAVING clause allows you to select the groups that you want to see based on values which appertain to that group. Consider the example given above.

```
SELECT Customer, Supplier,
AVG(Amount) AS AvgOfAmount
FROM SALES
GROUP BY Customer, Supplier;
```

Customer	Supplier	AvgOfAmount
Johnson	Harrison	£453.78
Jones	Ford	£453.00
Jones	Harrison	£3,421.00
Simpson	Harrison	£235.67
Smith	Ford	£82.78
Smith	Harrison	£235.67

Suppose, now the records are grouped in this way, that we are only interested in the groups where the average amount is £250 or more? The foolish solution is:

```
SELECT Customer, Supplier,
AVG(Amount) AS AvgOfAmount
FROM SALES
GROUP BY Customer, Supplier
ORDER BY AVG(Amount);
```

Customer	Supplier	AvgOfAmount
Smith	Ford	£82.78
Smith	Harrison	£235.67
Simpson	Harrison	£235.67
Jones	Ford	£453.00
Johnson	Harrison	£453.78
Jones	Harrison	£3,421.00

which, although it renders the desired values easy to find, nevertheless still leaves the job of actually locating them, up to the user. A much better solution would be:

```
SELECT Customer, Supplier,
AVG(Amount) AS AvgOfAmount
FROM SALES
GROUP BY Customer, Supplier
HAVING AVG(Amount) >= 250;
```

Customer	Supplier	AvgOfAmount
Johnson	Harrison	£453.78
Jones	Ford	£453.00
Jones	Harrison	£3,421.00

You can, of course, still order the groups:

```
SELECT Customer, Supplier,
AVG(Amount) AS AvgOfAmount
FROM SALES
GROUP BY Customer, Supplier
HAVING AVG(Amount) >= 250
ORDER BY AVG(Amount);
```

Customer	Supplier	AvgOfAmount
Jones	Ford	£453.00
Johnson	Harrison	£453.78
Jones	Harrison	£3,421.00

Working with multiple tables

So far, we have looked at using the SELECT statement with a single table. Clearly, since the relational model encourages us to split complex data into separate tables we will often find it necessary to recover data from two or more tables. To do this, we have to use the SELECT statement to draw data from both and the WHERE clause to form the joins.

Before we do, let's try querying the tables without using the WHERE clause.

```
SELECT SALES.Customer,
EMPLOYEES.LastName, SALES.Amount
FROM SALES, EMPLOYEES;
```

produces the data shown in Fig6.

Note that this SQL statement includes, for the first time, the table names when fields are being specified. Up to this point our SELECT statements have referred to single tables. Since field names within a single table must be unique, the field name alone allowed us to unambiguously identify the fields. However, field names can (and often are) shared by different tables. For example, both SALES and EMPLOYEES have a field called EmployeeNo. Therefore, the only way to identify a precise field uniquely is to use the table name as well. SQL syntax typically has the table name first in upper case, followed by a dot, followed by the field name in lower case.

SQL allows you to substitute temporary synonyms for table names:

```
SELECT S.Customer, E.LastName,
S.Amount
FROM SALES S, EMPLOYEES E;
```

which can shorten statements considerably but also tends to makes them less readable.

Note that the synonyms are defined in the FROM clause, but can still be used in the SELECT clause which tells you something about the way in which the SQL statement is read by the RDBMS.

To return to the multiple table query, if we were to add a WHERE clause as shown here:

```
SELECT SALES.Customer,
EMPLOYEES.LastName, SALES.Amount
FROM SALES, EMPLOYEES
WHERE SALES.EmployeeNo =
EMPLOYEES.EmployeeNo;
```

we get:

Customer	LastName	Amount
Simpson	Groves	£235.67
Johnson	Groves	£453.78
Simpson	Groves	£235.67
Jones	Groves	£453.00
Smith	Greeves	£82.78
Jones	Greeves	£3,421.00
Smith	Smith	£235.67

Referring to the base tables shows that this is a more useful answer table than the previous one.

How it works, and what it's doing, will be revealed next month. ■

Fig 6

Customer	LastName	Amount
Simpson	Groves	£235.67
Johnson	Groves	£453.78
Smith	Groves	£82.78
Jones	Groves	£3,421.00
Smith	Groves	£235.67
Simpson	Groves	£235.67
Jones	Groves	£453.00
Simpson	Greeves	£235.67
Johnson	Greeves	£453.78
Smith	Greeves	£82.78
Jones	Greeves	£3,421.00
Smith	Greeves	£235.67
Simpson	Greeves	£235.67
Jones	Greeves	£453.00
Simpson	Smith	£235.67
Johnson	Smith	£453.78
Smith	Smith	£82.78
Jones	Smith	£3,421.00
Smith	Smith	£235.67
Simpson	Smith	£235.67
Jones	Smith	£453.00
Simpson	Jones	£235.67
Johnson	Jones	£453.78
Smith	Jones	£82.78
Jones	Jones	£3,421.00
Smith	Jones	£235.67
Simpson	Jones	£235.67
Jones	Jones	£453.00

Fig 5

NumberInGroup	Customer	Supplier	AvgOfAmount
1	Johnson	Harrison	£453.78
1	Jones	Ford	£453.00
1	Jones	Harrison	£3,421.00
2	Simpson	Harrison	£235.67
1	Smith	Ford	£82.78
1	Smith	Harrison	£235.67