Surviving Client/Server: More Data Processing With SQL

by Steve Troxell

Last month, we saw how we can use subqueries to get SQL to do some of the data manipulation work we might otherwise be tempted to code into the client application. This month, we'll go further by examining some SQL programming techniques to perform even more complex data computations.

Frequency Distributions With GROUP BY

If you recall from the last issue, many of the examples showed counts of various items. However, the raw data you have to work with may show discrete transactions rather than summarized data, as we had assumed for the web page examples in the last issue.

You may have to show total customers by region from a table of customers. You may have to show sales by month from a table of orders. You may have to show contacts per zip code from a table of mailing addresses. Or, you may even have to show web hits by month from a table of individual HTTP transactions. Showing quantities of different categories of units is a basic statistical analysis method called a frequency distribution.

Producing these basic frequency distributions from raw data is actually quite simple with SQL's `GROUP BY` clause. For example, the query in Figure 1 shows the breakdown of customers by state. Notice that the `ORDER BY` clause gives us a list of states starting with the highest concentration of customers down to the lowest.

We can also employ the `HAVING` clause to filter the data shown in our frequency distributions. If we were only interested in states with more customers than California, we could use the query shown in Figure 2.

Business Statistics

Figure 3 shows a typical business report displaying departmental budgets for an organization (this data comes from the example InterBase database `EMPLOYEE.GDB`). This entire report can be obtained through one SQL statement (with the nifty formatting of the currency and percentage fields coming from an external reporting tool such as Excel or Crystal Reports). We'll examine each computation one at a time until we've built up to the full report query.

The `Department` and `Budget` columns in our report simply come from the corresponding fields in the `Department` table, no computation is involved yet. The `Budget Share` column shows each department's budget as a proportion of the total budget.

➤ *Figure 1*

```
SELECT State, COUNT(*)
  FROM Customers
  GROUP BY State
  ORDER BY 2 DESC

STATE COUNT
===== =====
NY    25545
PA    17882
CA    11567
FL    8773
CO    2308
```

➤ *Figure 2*

```
SELECT State, COUNT(*)
  FROM Customers
  GROUP BY State
  HAVING COUNT(*) > (SELECT
COUNT(*) FROM Customers
    WHERE State = 'CA')

STATE COUNT
===== =====
NY    25545
PA    17882
```

➤ *Listing 1*

```
SELECT Department, Budget,
  (Budget / (SELECT SUM(Budget) FROM Department)) AS Share
  FROM Department
  ORDER BY Budget DESC, Department ASC
```

➤ *Listing 2*

```
SELECT Department, Budget,
  (Budget / (SELECT SUM(Budget) FROM Department)) AS Share,
  (SELECT SUM(Budget) FROM Department
    WHERE (Budget > D.Budget) OR (Budget = D.Budget AND
      Department <= D.Department)) AS CumulativeTotal
  FROM Department D
  ORDER BY Budget DESC, Department ASC
```

➤ *Listing 3*

```
SELECT Department, Budget,
  (Budget / (SELECT SUM(Budget) FROM Department)) AS Share,
  (SELECT SUM(Budget) FROM Department
    WHERE (Budget > D.Budget) OR (Budget = D.Budget AND
      Department <= D.Department)) AS CumulativeTotal,
  (SELECT SUM(Budget) FROM Department
    WHERE (Budget > D.Budget) OR (Budget = D.Budget AND
      Department <= D.Department)) /
  (SELECT SUM(Budget) FROM Department) AS CumulativePercent
FROM Department D
ORDER BY Budget DESC, Department ASC
```

Department	Budget	Budget Share	Cumulative Total	Cumulative Percent
Sales and Marketing	$2,000,000	13.0%	$2,000,000	13.0%
Marketing	$1,500,000	9.7%	$3,500,000	22.7%
Software Products Div.	$1,200,000	7.8%	$4,700,000	30.5%
Consumer Electronics Div.	$1,150,000	7.5%	$5,850,000	38.0%
Engineering	$1,100,000	7.1%	$6,950,000	45.1%
Corporate Headquarters	$1,000,000	6.5%	$7,950,000	51.6%
Customer Services	$850,000	5.5%	$8,800,000	57.1%
European Headquarters	$700,000	4.5%	$9,500,000	61.6%
Customer Support	$650,000	4.2%	$10,150,000	65.9%
Pacific Rim Headquarters	$600,000	3.9%	$10,750,000	69.8%
Field Office: Canada	$500,000	3.2%	$11,250,000	73.0%
Field Office: East Coast	$500,000	3.2%	$11,750,000	76.2%
Field Office: Japan	$500,000	3.2%	$12,250,000	79.5%
Field Office: Switzerland	$500,000	3.2%	$12,750,000	82.7%
Research and Development	$460,000	3.0%	$13,210,000	85.7%
Field Office: France	$400,000	2.6%	$13,610,000	88.3%
Field Office: Italy	$400,000	2.6%	$14,010,000	90.9%
Finance	$400,000	2.6%	$14,410,000	93.5%
Software Development	$400,000	2.6%	$14,810,000	96.1%
Field Office: Singapore	$300,000	1.9%	$15,110,000	98.1%
Quality Assurance	$300,000	1.9%	$15,410,000	100.0%

➤ *Figure 3*

We can use the query shown in Listing 1 to produce the first three columns of our report. This query uses a subquery to compute the total budget and then uses that as the divisor for the percent of total calculation. Bear in mind that in this case, most SQL servers will execute the inner query once, substitute the resulting value as a constant into the outer query, then run the outer query.

Although the overall list is ordered by the amount of the budget, we include the department name in the ordering to ensure that departments with the same budget amount are sequenced consistently. This is not only important for the user, but becomes critical for the enhancements we'll add in the following sections.

The cumulative total is simply a running total of all the department budgets, ending with the total budget for the entire organization on the last line. Listing 2 shows how we can calculate this value by adding a correlated subquery.

Because the cumulative total is defined as the sum of all rows prior to and including the current row, our query is now sensitive to the order of the rows in the result set. For the budget report in our example, the order is generally determined by the `Budget` column, so the coarse definition for our calculation would be "the sum of all budgets greater than or equal to the current department's budget." The correlated subquery that corresponds to this definition would be:

```
(SELECT SUM(Budget)
   FROM Department
 WHERE (Budget >= D.Budget)
```

But this doesn't properly account for the case where more than one department has the same budget. With this subquery, the cumulative total for each of the four field offices with a $500,000 budget would be $12,750,000.

To handle this case, we must define some order to the duplicating rows. The `ORDER BY` clause in Listing 2 shows that when the budgets are the same, the rows are ordered alphabetically by the department name. Knowing this, we arrive at the correct subquery to produce the cumulative total. The initial part of the subquery's `WHERE` clause (`Budget > D.Budget`) finds all budgets greater than the current department's budget. The next part handles the case where there are other departments with the same budget (`Budget = D.Budget`) and includes only those departments that already appear in the result set (`Department <= D.Department`). Notice the direction of the relational operators for `Budget` (>) and `Department` (<). This is because the order of the overall result set is descending on `Budget` and ascending on `Department`.

To calculate the cumulative percent for any given row, we simply divide the cumulative total for that row by the overall total. Listing 3 shows the query we use to accomplish this. All we have to do here is repeat the subquery that calculates the cumulative total and divide it by a subquery to obtain the overall total. We already have subqueries to calculate both of these values, but unfortunately most SQL servers won't allow us to simply use the column names of calculated fields to produce another calculated field. So we are forced to repeat our subquery definitions for the terms in our calculation of cumulative percent.

```
Corporate Headquarters
  Engineering
    Consumer Electronics Div.
      Customer Services
      Research and Development
    Software Products Div.
      Customer Support
      Quality Assurance
      Software Development
  Finance
  Sales and Marketing
    European Headquarters
      Field Office: France
      Field Office: Italy
      Field Office: Switzerland
    Field Office: Canada
    Field Office: East Coast
    Marketing
    Pacific Rim Headquarters
      Field Office: Japan
      Field Office: Singapore
```

➤ *Figure 4*

Hierarchical Data

On some occasions you may find need to represent data in hierarchical form. For example, the menu structure of a typical Windows application is hierarchical in nature. Basically, any type of data that can be represented in Delphi's TOutline control is hierarchical.

In the Department table from EMPLOYEE.GDB, there is a Head_Dept field. This field represents the "parent" for any given department. By threading our way though the Head_Dept and Dept_No fields we can construct the organization's department hierarchy. Figure 4 shows the desired outline (note that departments on the same level are shown alphabetically).

To produce this chart, we can use SQL to derive the correct outline positions, or indentation levels, for each department in the table.

Listing 4 shows a recursive InterBase stored procedure that produces a result set from which we may construct this chart. To run this procedure, we provide the root department number we want to start with and an indentation level of 0 (see Figure 5). The result set shows the indentation level, department number, and department name for each department in the organization. From the indentation level we can easily insert the data into a TOutline control or prepend spaces to the department name to produce the chart shown in Figure 4.

```
CREATE PROCEDURE DeptChart(Head_Dept char(3), Indent smallint)
  RETURNS (Lvl smallint, Dept_No char(3), Department varchar(25))
AS
  DECLARE VARIABLE Child_Dept char(3);
BEGIN
  /* Return info for the dept passed in */
  SELECT :Indent, Dept_No, Department
    FROM Department
    WHERE Dept_No = :Head_Dept
    INTO :Lvl, :Dept_No, :Department;
  SUSPEND;
  Indent = Indent + 1;
  /* Find all depts one level below this one.
     The ORDER BY here determines the sequence of
     departments at the same level. */
  FOR
    SELECT Dept_No FROM Department
      WHERE Head_Dept = :Head_Dept
      ORDER BY Department
      INTO :Child_Dept
  DO
    FOR
      SELECT * FROM DeptChart(:Child_Dept, :Indent)
        INTO :Lvl, :Dept_No, :Department
    DO SUSPEND;
END
```

➤ *Listing 4*

What's handy about this procedure is that you can start with any department number and produce a chart for just that branch of the organization. Figure 6 shows a call to this procedure that produces a chart for just the Software Products Division.

A more effective approach is to eliminate the recursion and use temporary tables. Listing 5 shows a Microsoft SQL Server procedure to solve the same problem. Its output is identical to that shown in Figure 5. This technique comes from the *Microsoft SQL Server Database Developer's Companion*. It uses a temporary table called #Stack to implement a stack structure which keeps track of the departments we are drilling into as we expand their child departments. The output is compiled into another temporary table called #Results which is dumped as the result set for the stored procedure.

With SQL Server, temporary tables are local to the procedure that creates them and can be stored within server RAM, so the overhead is minimal. Also, because the recursive call is eliminated, the caller doesn't need to pass in the initial indentation value.

Cross Tabulations

A cross tabulation (crosstab) is a basic method of comparing two variables in a set of data (in practice, a crosstab can actually be built for n variables, but we'll restrict our discussion here to just two dimensions). The results of a crosstab generally show all the values for one variable listed down the left side as row headers and all the values for the second variable listed across the top as column headers, with the intersecting "cells" representing the number of data elements containing that combination of variables (Figure 7).

There are a number of SQL techniques to produce crosstab result sets, but one particularly elegant approach I'd like to share with you comes from Joe Celko's *SQL for Smarties: Advanced SQL Programming*. The crosstab shown in Figure

```
SELECT * FROM DeptChart('000', 0)

LVL DEPT_NO DEPARTMENT
=== ======= ========================
  0 000     Corporate Headquarters
  1 600     Engineering
  2 670     Consumer Electronics Div.
  3 672     Customer Services
  3 671     Research and Development
  2 620     Software Products Div.
  3 623     Customer Support
  3 622     Quality Assurance
  3 621     Software Development
  1 900     Finance
  1 100     Sales and Marketing
  2 120     European Headquarters
  3 123     Field Office: France
  3 125     Field Office: Italy
  3 121     Field Office: Switzerland
  2 140     Field Office: Canada
  2 130     Field Office: East Coast
  2 180     Marketing
  2 110     Pacific Rim Headquarters
  3 115     Field Office: Japan
  3 116     Field Office: Singapore
```

➤ *Figure 5*

```
SELECT * FROM DeptChart('620', 0)
LVL    DEPT_NO DEPARTMENT
====== ====== ====================
     0 620     Software Products Div.
     1 623     Customer Support
     1 622     Quality Assurance
     1 621     Software Development
```

➤ *Figure 6*

```
Type          0736   0877   1389      Total
business      1      0      3         4
mod_cook      0      2      0         2
popular_comp  0      0      3         3
psychology    4      1      0         5
trad_cook     0      3      0         3
UNDECIDED     0      1      0         1
Total         5      7      6         18
```

➤ *Figure 7*

```
CREATE PROCEDURE DeptChart (@Current char(3))
AS
BEGIN
  DECLARE @Indent smallint
  /* Create stack and initialize it with the root department */
  CREATE TABLE #Stack (Dept_No char(3), Indent smallint)
  INSERT INTO #Stack VALUES (@Current, 0)
  CREATE TABLE #Result (Indent smallint, Dept_No char(3),
    Department varchar(30))
  /* Initial indentation level is 0 */
  SELECT @Indent = 0
  WHILE @Indent >= 0
  BEGIN
    /* If any departments are at this level */
    IF EXISTS(SELECT * FROM #Stack WHERE Indent = @Indent)
    BEGIN
      SELECT @Current = Dept_No FROM #Stack WHERE Indent = @Indent
    /* Return info for the current dept */
    INSERT INTO #Result
      SELECT @Indent, @Current, SPACE((@Indent - 1) * 2) + Department
        FROM Department WHERE Dept_No = @Current
    /* Remove current dept from the stack */
    DELETE FROM #Stack WHERE Indent = @Indent AND Dept_No = @Current
    /* Find all the depts one level below this dept */
    INSERT INTO #Stack SELECT Dept_No, @Indent + 1 FROM Department
      WHERE Head_Dept = @Current
      ORDER BY Department DESC
    /* If any found, increase the indentation level. @@ROWCOUNT
       returns the # of rows affected by the immediately previous
       SQL statement. */
    IF @@ROWCOUNT > 0 SELECT @Indent = @Indent + 1
  END
  ELSE
    SELECT @Indent = @Indent - 1
END
SELECT * FROM #Result /* Return the accumulated results */
END
```

➤ *Listing 5*

```
SELECT Type, Pub_ID, COUNT(*)
  FROM Titles
  GROUP BY Type, Pub_ID

TYPE         PUB_ID COUNT
============ ====== =======
business     0736   1
business     1389   3
mod_cook     0877   2
popular_comp 1389   3
psychology   0736   4
psychology   0877   1
trad_cook    0877   3
UNDECIDED    0877   1
```

➤ *Figure 8*

7 gives book titles by publisher from the Titles table in the Microsoft example database Pubs. The raw data for this report can be shown with a simple SELECT GROUP BY statement (Figure 8). A crosstab takes this one dimensional data and presents it in the two dimensional grid format which is shown in Figure 7.

To get to Figure 7 from Figure 8, we can use the query shown in Figure 9 (on the next page). The first SELECT statement produces a list of rows of unique book types. The columns for these rows are defined as correlated subqueries computing the sum of all titles of that type for each of the three publishers (the PUB_xxxx notation at the end of the subqueries is how we assign a name to a computed column in SQL Server). The final subquery computes the total for each given book type to provide the row total in the crosstab.

This result set is unioned with another SELECT statement which uses subqueries to compute the column totals. The UNION ALL syntax means the rows from the two result sets are merged together without regard to duplicate rows between them. This prevents the union from imposing an implicit order to the rows and leaves our column totals as the last row in the overall result set, instead of it appearing alphabetically within the book types column.

CASE Expressions

ANSI SQL-92 provides a CASE expression that can be used to return conditional values. Unlike most CASE structures in procedural languages which control program flow, CASE in SQL is a type of expression macro that returns a value (like a function result) based on the logic within the CASE function. Therefore, CASE can be used to conditionally return a value anywhere an expression is allowed in SQL. Typically, you'll find CASE used to manipulate results returned by a SELECT statement as shown in Figure 10 (the examples for CASE are for Microsoft SQL Server, CASE is not supported by InterBase).

In this example, the third column is defined by the CASE function and the column has been given a name of Label (just as we can rename any column in a result set). Here, the CASE function examines the Country field and returns a character value based on the contents of the that field. The WHEN clause translates to "when Country equals 'USA'" and the THEN clause defines the value to

```
/* First query: */
SELECT Emp_No, Dept_No, Salary,
  Salary * 1.20 NewSalary
    FROM Employee
    WHERE Dept_No = '621'
UNION
SELECT Emp_No, Dept_No, Salary,
  Salary * 1.05 NewSalary
    FROM Employee
    WHERE Dept_No = '180'

/* Second query: */
SELECT Emp_No, Dept_No, Salary,
    CASE
      WHEN Dept_No = '621' THEN
        Salary * 1.20
      WHEN Dept_No = '180' THEN
        Salary * 1.05
    END NewSalary
  FROM Employee
  WHERE Dept_No = '621' OR
    Dept_No = '180'
```

➤ *Listing 6*

```
SELECT DISTINCT Type,
  (SELECT COUNT(*) FROM Titles
    WHERE Type = TO.Type AND Pub_ID = '0736') 'PUB_0736',
  (SELECT COUNT(*) FROM Titles
    WHERE Type = TO.Type AND Pub_ID = '0877') 'PUB_0877',
  (SELECT COUNT(*) FROM Titles
    WHERE Type = TO.Type AND Pub_ID = '1389') 'PUB_1389',
  (SELECT COUNT(*) FROM Titles
    WHERE Type = TO.Type) 'RowTotal'
  FROM Titles TO
UNION ALL
SELECT 'Column Total',
  (SELECT COUNT(*) FROM Titles WHERE Pub_ID = '0736'),
  (SELECT COUNT(*) FROM Titles WHERE Pub_ID = '0877'),
  (SELECT COUNT(*) FROM Titles WHERE Pub_ID = '1389'),
  (SELECT COUNT(*) FROM Titles)

Type         PUB_0736    PUB_0877    PUB_1389    RowTotal
============ =========== =========== =========== ===========
business     1           0           3           4
mod_cook     0           2           0           2
popular_comp 0           0           3           3
psychology   4           1           0           5
trad_cook    0           3           0           3
UNDECIDED    0           1           0           1
Column Total 5           7           6           18
```

➤ *Figure 9*

```
SELECT Pub_Name, Country,
    CASE Country
      WHEN 'USA' THEN 'Domestic'
      ELSE 'Overseas'
    END Label
  FROM Publishers

Pub_Name                  Country      Label
========================= ============ =======
New Moon Books            USA          Domestic
Binnet & Hardley          USA          Domestic
Algodata Infosystems      USA          Domestic
Five Lakes Publishing     USA          Domestic
Ramona Publishers         USA          Domestic
GGG&G                     Germany      Overseas
Scootney Books            USA          Domestic
Lucerne Publishing        France       Overseas
```

➤ *Figure 10*

```
SELECT *,
    CASE
      WHEN RoyaltyPer <= 33 THEN "Bottom Third"
      WHEN RoyaltyPer > 33 AND RoyaltyPer <= 66 THEN "Middle Third"
      WHEN RoyaltyPer > 66 THEN "Top Third"
      ELSE "Undefined"
    END
  FROM TitleAuthor

au_id       title_id au_ord royaltyper
=========== ======== ====== =========== ===========
172-32-1176 PS3333   1      100         Top Third
213-46-8915 BU1032   2      40          Middle Third
213-46-8915 BU2075   1      100         Top Third
238-95-7766 PC1035   1      100         Top Third
267-41-2394 BU1111   2      40          Middle Third
267-41-2394 TC7777   2      30          Bottom Third
274-80-9391 BU7832   1      100         Top Third
409-56-7008 BU1032   1      60          Middle Third
427-17-2319 PC8888   1      50          Middle Third
472-27-2349 TC7777   3      30          Bottom Third
```

➤ *Figure 11*

return if the WHEN clause is true. As with most CASE structures, you can have as many WHEN THEN clauses as you have cases to examine. Any unaccounted for cases can be captured by an optional ELSE clause which simply defines the value to return if the case is not explicitly handled by one of the WHEN THEN clauses.

CASEs also can be used to check non-discrete values. Figure 11 shows an alternate way to use CASE to examine ranges of values. Here, the TitleAuthor table links one or more authors to a book title and defines the royalty percentage for each author. The CASE function simply compares the range of the royalty percentage and returns a character value labeling the segment of the range that author happens to fall in.

CASE functions don't have to return just character values, but any SQL data type: integers, floats, dates, even nulls if that's what you need. Also, a subquery could be used in the WHEN expression as long as it results in a single value that could be compared against.

Using CASE may allow you to avoid UNIONing multiple SELECT statements together to produce a single result set with calculated fields that vary based on row content. For example, Listing 6 shows two equivalent queries which produce a projected salary increase report for two departments, each department using a different calculation to increase salaries.

CASE is not restricted to manipulating column values returned by SELECT. Actually, CASE can be used anywhere an expression is allowed. A CASE function can be used in an UPDATE to modify a column with values varying based on the CASE logic, or you could even use a CASE function to affect a WHERE clause.

The COALESCE Function

Related to the CASE function is the COALESCE function, which accepts any number of arguments and returns the first argument that is not null, or returns null if all arguments are null. The example shown in Figure 12 illustrates this (this example comes directly from the SQL Server on-line help). Given a table of employees, which may be hourly, salaried, or paid on commission, compute the total annual compensation for each employee.

```
SELECT *, COALESCE(Hourly_Wage * 40 * 52,
     Salary,
     Commission * Num_Sales) TotalComp
  FROM Wages

Emp_ID Hourly_Wage Salary      Commission  Num_Sales   TotalComp
====== =========== =========== =========== =========== ===========
1      10          (null)      (null)      (null)      20800
2      20          (null)      (null)      (null)      41600
3      30          (null)      (null)      (null)      62400
4      40          (null)      (null)      (null)      83200
5      (null)      10000       (null)      (null)      10000
6      (null)      20000       (null)      (null)      20000
7      (null)      30000       (null)      (null)      30000
8      (null)      40000       (null)      (null)      40000
9      (null)      (null)      15000       3           45000
10     (null)      (null)      25000       2           50000
11     (null)      (null)      20000       6           120000
12     (null)      (null)      14000       4           56000
```

➤ *Figure 12*

Conclusion

As you can see, there is quite a bit you can accomplish within SQL, which usually provides a more compact, reliable solution.

When embedded within stored procedures or triggers, an SQL-based algorithm is re-usable by other applications, even if written in different languages. You have to be careful though, some of the subquery processing can be a performance problem for very large, poorly organized tables. On the other hand, processing like the crosstabs might be more efficiently handled by the server than by downloading all the raw data to a client application or report writer across a network.

As with everything, good judgment must prevail in deciding the appropriateness of any technique for your particular circumstances.

However, deciding which tool is the right tool is better if you have lots of tools to choose from.

Steve Troxell is a software engineer with TurboPower Software where he is developing Delphi client/server applications for the casino industry. Steve can be contacted at stevet@tpower.com or on CompuServe at 74071,2207