Surviving Client/Server: Getting Started With SQL Part 2

by Steve Troxell

As you may recall from the last issue, we took the plunge into SQL and got a crash course in the basic operations of `SELECT`, `INSERT`, `UPDATE` and `DELETE`. You may have noticed that `SELECT` was a pretty powerful statement since over half of the material covered that statement alone. Well, we're going to cover even more about `SELECT` and we still won't have gotten to all of it. This time around we're going to learn about totalling and subtotalling, removing duplicates, grouping data, conditional calculations, and joining unrelated tables. These are some of the more practical tools used to create useful reports out of SQL data.

As before, we will use the sample employee database that ships with Delphi for all our examples. You may want to try the examples with ISQL as you read the text. The sample database can be found in `\IBLOCAL\EXAMPLES\EMPLOYEE.GDB`. To get started with ISQL check out the Local Interbase Server User's Guide, or see the last issue's column for instructions.

Aggregate Functions

Aggregate functions are among the more powerful features of SQL. When used appropriately, they can shift a great deal of reporting work from the client application onto the database server. Aggregate functions return a single value for a given set of rows. For example, the query

```
SELECT SUM(salary) FROM
  employee;
```

returns the total of the salaries for all employees:

```
              SUM
===============
    115522468.00
```

Note that the result set returned by the query consists of a single row with a single column (the `SUM` column). The aggregate function operates on each row specified by the query, but only the final (aggregate) answer is returned by the query. There are five aggregate functions in SQL:

- ➢ `SUM()` The total of the values in the argument,
- ➢ `AVG()` The average of the values in the argument,
- ➢ `COUNT()` The number of non-null values in the argument,
- ➢ `MAX()` The greatest value in the argument,
- ➢ `MIN()` The least value in the argument.

To find the highest salary, lowest salary, and average salary for the company, use the query shown in Figure 1.

```
SELECT MAX(salary),
  MIN(salary), AVG(salary)
  FROM employee;

           MAX            MIN            AVG
============== ============== ==============
   99000000.00       22935.00     2750534.95
```

➤ *Figure 1*

Aggregate functions work on any set you can query with a `SELECT` statement. For example, if you wanted to see the salary figures for a single department instead of the whole company, just add a `WHERE` clause to the query, as shown in Figure 2.

```
SELECT MAX(salary), MIN(salary), AVG(salary) FROM employee
  WHERE dept_no = 621;

           MAX            MIN            AVG
============== ============== ==============
     975000.00       36000.00       69184.88
```

➤ *Figure 2*

The argument to an aggregate function is usually a table column, but it can also be an expression. For example, the following query returns the average unit price for all orders in the sales table:

```
SELECT AVG(total_value /
  qty_ordered) FROM sales;

                    AVG
======================
     4563.823426619132
```

In addition, the `COUNT()` function can include a special argument. `COUNT(columnname)` returns the number of values in the column *excluding nulls*. However, you can use `COUNT(*)` to return the total number of rows in the result regardless of any nulls in any column.

The `SUM()`, `AVG()` and `COUNT()` aggregate functions can include the `DISTINCT` keyword within the parentheses to force the function to ignore any duplicate values in the column. If you wanted to know the number of unique job codes in the employee table, a regular `COUNT(job_code)` would return the total number of employees (each one has a value in the `job_code` column). But `COUNT(DISTINCT job_code)` would return the correct number of different job codes regardless of how many employees were assigned to the same code:

```
SELECT COUNT(job_code),
  COUNT(DISTINCT job_code)
  FROM employee;

      COUNT       COUNT
=========== ===========
         42          13
```

Eliminating Duplicate Rows

Consider the problem of creating a report of all the customers who have not yet paid for their orders. To do this, we must look at the sales table to see which orders have not been paid, and join that with the customer table to get the name of the customer:

```
SELECT customer FROM customer,
  sales
  WHERE paid = 'n' AND
  sales.cust_no =
  customer.cust_no
  ORDER BY customer;
```

This statement selects all the unpaid orders from sales, determined by the value of the `paid` column. The `cust_no` column is the link we use to join the sales and customers tables. Take a look at the result set for this query shown in Figure 3. Notice that several customers are listed more than once. This is because our query returns a row for every unpaid order, and these customers have more than one order that has not been paid. To make the query return a single row for every customer with an unpaid order, regardless of how many unpaid orders they have, we have to get rid of the duplicate rows. To do this we use the `DISTINCT` keyword to return only distinct (non-duplicating) rows (see Figure 4):

```
SELECT DISTINCT customer FROM
  customer, sales
  WHERE paid = 'n' AND
  sales.cust_no =
  customer.cust_no
  ORDER BY customer;
```

By including the `DISTINCT` keyword in any `SELECT` query, we can

```
SELECT customer FROM customer, sales
  WHERE paid = 'n' AND sales.cust_no = customer.cust_no
  ORDER BY customer;

CUSTOMER
=========================
3D-Pad Corp.
3D-Pad Corp.
Anini Vacation Rentals
Anini Vacation Rentals
Buttle, Griffith and Co.
DT Systems, LTD.
Dallas Technologies
Dallas Technologies
GeoTech Inc.
Lorenzi Export, Ltd.
Max
Signature Design
Signature Design
Signature Design
```

➤ *Figure 3*

```
SELECT DISTINCT customer FROM customer, sales
  WHERE paid = 'n' AND sales.cust_no = customer.cust_no
  ORDER BY customer;

CUSTOMER
=========================

3D-Pad Corp.
Anini Vacation Rentals
Buttle, Griffith and Co.
DT Systems, LTD.
Dallas Technologies
GeoTech Inc.
Lorenzi Export, Ltd.
Max
Signature Design
```

➤ *Figure 4*

automatically eliminate any duplication of rows. Duplication is defined as *identical values for all columns* in the select list, not just the first one. For example, suppose we wanted to add the amount of each order that has not been paid to our report:

```
SELECT DISTINCT customer,
  total_value FROM
  customer, sales
  WHERE paid = 'n' AND
  sales.cust_no =
  customer.cust_no
  ORDER BY customer;
```

Take a look at the results shown in Figure 5. We have duplication of the customer name again, even though we used the `DISTINCT` keyword. That's because `DISTINCT` operates on all the columns of the select list (in this case customer and `total_value`). In fact, if it happened that the same customer had two or more unpaid orders with the same total amount, then those entries would be combined into one.

Be sure to note the difference in `DISTINCT` when used in the select list (as shown above) versus when used within an aggregate function. In a select list, `DISTINCT` eliminates duplicate rows from the display. In an aggregate function, `DISTINCT` eliminates duplicate values (not the entire row) from the aggregate computation.

Grouping Data

Since the query shown above does not work, how do we obtain a non-duplicating list of customers and the total of their unpaid orders? Well, "total of unpaid orders" implies a calculation and for that we need to use an aggregate function. But until now, we've only used aggregate functions to compute a result for the entire set returned by the `SELECT`. In this case we need to compute a new result for each set of rows for a given customer. To accomplish this, we need to use a GROUP BY clause.

GROUP BY is another mechanism for removing duplicate rows. We could have reworked our query in Figure 4 to use GROUP BY to produce the same result set:

```
SELECT customer FROM
   customers, sales
   WHERE paid = 'n' AND
   sales.cust_no =
   customer.cust_no
   GROUP BY customer;
```

Like DISTINCT, GROUP BY will return only a single row for each group of rows that match on the fields listed after GROUP BY. But GROUP BY is far more useful than this. We can use this clause in combination with aggregate functions to perform basic calculations on the set of rows represented by each row in the result set. To get our list of outstanding customers, we simply use the query shown in Figure 6.

Notice that although only one row is shown for each customer, the aggregate function SUM() still uses all the rows for each customer to make the calculation (that is, all the rows the WHERE clause permitted).

Another example of how this might be useful to us would be a sales report showing number of sales, total sales, and average sale by salesman. The query in Figure 7 produces such a report.

Notice how the output of the aggregate functions in this query differs from what we had when we used them in a regular SELECT statement (without a GROUP BY clause). The GROUP BY clause creates sets of rows with matching values, but only shows one row for each set in the final output. When an aggregate function is used in this type of SELECT statement, it returns a value for each group of rows, instead of a value for the entire result set. In fact, GROUP BY and aggregate functions are most often used together.

GROUP BY can begin to cause problems when you have more than one table column in the select list. Although there are four columns in the output of the query shown above, only one of them, full_name, is a table column, the other three are aggregate function calculations. Take a look at this query:

```
SELECT sales_rep, qty_ordered
  FROM sales
  GROUP BY sales_rep;
```

This doesn't work (Interbase produces an "invalid column reference" error). The rows are arranged by sales_rep and, like DISTINCT, the rows must match on all of the columns in the select list. In this case, each salesman is likely to have different order quantities

```
SELECT DISTINCT customer, total_value FROM customer, sales
  WHERE paid = 'n' AND sales.cust_no = customer.cust_no
  ORDER BY customer;

CUSTOMER                   TOTAL_VALUE
========================= ===========

3D-Pad Corp.                   999.98
3D-Pad Corp.                 10000.00
Anini Vacation Rentals        9000.00
Anini Vacation Rentals       16000.00
Buttle, Griffith and Co.         0.00
DT Systems, LTD.              9000.00
Dallas Technologies          14850.00
Dallas Technologies          20000.00
GeoTech Inc.                  1500.00
Lorenzi Export, Ltd.          2693.00
Max                            490.69
Signature Design              3399.15
Signature Design             60000.00
Signature Design            422210.97
```

➤ *Figure 5*

```
SELECT customer, SUM(total_value)
  FROM customer, sales
  WHERE paid = 'n' AND sales.cust_no = customer.cust_no
  GROUP BY customer;

CUSTOMER                           SUM
========================= ===========

3D-Pad Corp.                 10999.98
Anini Vacation Rentals       25000.00
Buttle, Griffith and Co.         0.00
DT Systems, LTD.              9000.00
Dallas Technologies          34850.00
GeoTech Inc.                  1500.00
Lorenzi Export, Ltd.          2693.00
Max                            490.69
Signature Design            485610.12
```

➤ *Figure 6*

```
SELECT full_name,
       COUNT(*) AS num_sales,
       SUM(total_value) AS total_sales,
       AVG(total_value) AS avg_sale
  FROM sales, employee
  WHERE sales.sales_rep = employee.emp_no
  GROUP BY full_name;

FULL_NAME                     NUM_SALES TOTAL_SALES    AVG_SALE
=========================== =========== =========== ===========

Ferrari, Roberto                      2   122693.00    61346.50
Glon, Jacques                         5   462600.49    92520.10
Leung, Luke                           5    37475.69     7495.14
Osborne, Pierre                       1     1980.72     1980.72
Sutherland, Claudia                   3   960008.00   320002.67
Weston, K. J.                         8   139450.50    17431.31
Yamamoto, Takashi                     3    24190.40     8063.47
Yanowski, Michael                     6   502192.23    83698.71
```

➤ *Figure 7*

on each row, making them unique rows that cannot be grouped together. You must use GROUP BY sales_rep, qty_ordered to make this query work. However, this returns a row for each unique salesman/quantity combination, not each unique salesman. That may or may not be what you were looking for.

Some SQL servers require that all the table columns in the select list (excluding aggregate functions) appear in the GROUP BY clause as well, even if there is a one-to-one relationship in the values of all the selected columns. For example, the query:

```
SELECT sales_rep, full_name,
  COUNT(*) as num_sales
  FROM sales, employee
  WHERE sales.sales_rep =
  employee.emp_no;
  GROUP BY sales_rep;
```

is illegal in Interbase, even though each instance of duplication in the sales_rep column is matched exactly by duplication in the full_name column (thereby conceptually allowing them to be grouped together). Interbase requires all non-aggregate function items in the select list also appear in the GROUP BY clause.

Limiting Groups

Given our sales report shown in Figure 7, suppose we wanted to restrict the report to salesmen having more than $250,000 in sales. Our first instinct might be to add the clause WHERE SUM(total_value) > 250000 to our query, but this produces an error because aggregate functions are not allowed within a WHERE clause. This is sensible because WHERE prevents rows that don't match the criteria from being processed, yet aggregate functions must process the rows to get an aggregate answer. It's a chicken-or-egg paradox: you can't determine which rows to process based on a function that requires processing the rows.

The solution is to use the HAVING clause. Whereas WHERE determines which *rows* to include in the *processing*, HAVING determines which *groups* to include in the *output*. The query in Figure 8 uses a HAVING clause to produce the report we want.

```
SELECT full_name,
       COUNT(*) AS num_sales,
       SUM(total_value) AS total_sales,
       AVG(total_value) AS avg_sale
 FROM sales, employee
 WHERE sales.sales_rep = employee.emp_no
 GROUP BY full_name
 HAVING SUM(total_value) > 250000;

FULL_NAME                       NUM_SALES TOTAL_SALES    AVG_SALE
=============================== ========= ===========  ===========

Glon, Jacques                           5   462600.49    92520.10
Sutherland, Claudia                     3   960008.00   320002.67
Yanowski, Michael                       6   502192.23    83698.71
```

➤ *Figure 8*

```
SELECT full_name,
       COUNT(*) AS num_sales,
       SUM(total_value) AS total_sales,
       AVG(total_value) AS avg_sale
  FROM sales, employee
  WHERE sales.sales_rep = employee.emp_no AND
        paid = 'y'
  GROUP BY full_name
  HAVING SUM(total_value) > 250000;

FULL_NAME                       NUM_SALES TOTAL_SALES    AVG_SALE
=============================== ========= ===========  ===========

Glon, Jacques                           2   450100.51   225050.26
Sutherland, Claudia                     3   960008.00   320002.67
```

➤ *Figure 9*

In this case the WHERE clause only serves to make the join between the sales and employee tables; it does not filter any rows. The HAVING clause limits the groups that are returned by the query,

Let's do the same sales report again, but this time let's only include paid orders. The query and results are shown in Figure 9.

Notice in the results that the aggregate values are different. That's because the WHERE clause threw out some rows before they got grouped (and hence aggregated) by GROUP BY. Also notice that we now have one less row because one salesman's new aggregate total falls below our threshold of $250,000.

Unions

Let's say it's annual performance review time and you want a report to project the planned salary increases for two departments. Everyone in the Software Development department (dept 621) will receive a 20% increase and everyone in the Marketing department (dept 180) will receive a 5% increase *[Now **that** looks like a company with real vision! Editor]*. Recall from last issue's column that we can easily include a calculation within a SELECT statement to compute the new salary for each employee. But in this case, we need to apply one of two different calculations to the same column depending upon which department the employee is in. We could do this with two separate queries, but we'd rather have a nice, clean, single output result set for employees from both departments. To do this, we can combine the results from our two queries in a union by connecting the SELECT statements with the UNION keyword, as in Figure 10.

Take a look at the output for this query. As you can see, the salary calculations were performed as desired depending on department. A union simply combines the output from multiple SELECTs into a single result set. And unions are not limited to just two SELECTs; you can string several SELECTs together with a UNION between each. However, be wary of using unions on large tables. Each SELECT within the union is performed independently and the results combined, so you could take a performance hit due to repetitive table traversal, particularly if the SELECT cannot take advantage of an index.

The most significant thing to notice about union output is that it does not contain all the rows from the first SELECT followed by all the rows from the second SELECT (as shown by the dept_no column in Figure 10). In fact, the rows have been ordered on the first column even though we did not include an ORDER BY clause. Interbase automatically orders results from a union; some other SQL servers do not. If you want to force the ordering of a union, you simply add an ORDER BY clause to the last SELECT in the union. The ORDER BY will be applied to the entire result set. In Interbase, when the ORDER BY clause is used within a union, you must specify the columns to order on by number, as in ORDER BY 2. Other SQL servers allow column names to be used as well.

One more thing to notice in Figure 10 is that we did not give a column heading to the new salary calculation. In a regular non-union SELECT statement we would use a column alias such as salary * 1.20 as new_salary. Unfortunately, for some peculiar reason, Interbase ignores column aliases in unions. Also, I have seen cases of unions containing more than two SELECTs where *all* of the column names are mysteriously dropped from the output.

Not only can a union be used to perform conditional calculations on a single table, as demonstrated in our previous example, but we can also use a union to pull rows from different tables into the same result set. This is not the same as a join because there is not necessarily any relationship between the separate tables. For example, let's suppose our company is throwing a product launch party. We're going to invite employees and customers to the party and we need an invitation list. This will entail a SELECT from both the employee table and the customer table (see Figure 11):

```
SELECT first_name, last_name,
  'empl' FROM employee
UNION
SELECT contact_first,
  contact_last, 'cust'
  FROM customer
```

This query returns all the rows from employee and all the rows from customer combined into a single result set. There is no linkage of the rows between tables as you'd have in a join. The only requirements are that each of the SELECTs must have the same number of items in the select list and those items must be of compatible data types. Notice that the actual column names do not have to be the same; the column names from the first SELECT are used as the column names for the entire output of the union.

In this query we included a literal in each SELECT to identify whether the person on the invitation list is an employee or a customer. Although you can place literals in any SELECT statement, when used in a union, you must make sure that the literals in each SELECT are the same length. If not, Interbase issues a "data type unknown" error; the same error that occurs if you use columns of different data types.

Ordering Clauses

Finally, I'd like to point out a minor but very important detail. When

```
SELECT full_name, dept_no, salary, salary * 1.20
  FROM employee
  WHERE dept_no = 621
UNION
SELECT full_name, dept_no, salary, salary * 1.05
  FROM employee
  WHERE dept_no = 180;

FULL_NAME             DEPT_NO          SALARY
===================== ======= ============== =====================

Bishop, Dana          621           62550.00                 75060
Green, T.J.           621           36000.00                 43200
Johnson, Leslie       180           64635.00              67866.75
Nordstrom, Carol      180           42742.50             44879.625
Ramanathan, Ashok     621           80689.50      96827.39999999999
Young, Bruce          621           97500.00                117000
```

➤ *Figure 10*

```
SELECT first_name, last_name, 'empl' FROM employee
UNION
SELECT contact_first, contact_last, 'cust' FROM customer

FIRST_NAME      LAST_NAME
=============== ==================== =====

Andreas         Lorenzi              cust
Ann             Bennet               empl
Ashok           Ramanathan           empl
Bill            Parker               empl
Bruce           Young                empl
Carol           Nordstrom            empl
Chris           Papadopoulos         empl
Claudia         Sutherland           empl
Dale J.         Little               cust
Dana            Bishop               empl
Elizabeth       Brocket              cust
Glen            Brown                cust
Greta           Hessels              cust
Jacques         Glon                 empl
James           Buttle               cust
Janet           Baldwin              empl
Jennifer M.     Burbank              empl
John            Montgomery           empl
K. J.           Weston               empl
K.M.            Neppelenbroek        cust
```

➤ *Figure 11 (partial listing of rows)*

multiple clauses are used in a SELECT statement, they are required to appear in a specific order as follows:

```
 SELECT <select-list>
   FROM <table(s)>
   WHERE <search-condition>
   GROUP BY <column-list>
   HAVING <search-condition>
   UNION <select-statement>
   ORDER BY <column-list>
```

If you start seeing "unknown token" errors complaining about a seemingly legal keyword, make sure you have your clauses in the right sequence.

Summary

As you can see, SQL provides more than just basic input/output operations. It includes some powerful features to manipulate, categorize, and compute the data to provide concise and meaningful answers. Although for a formal report you will generally turn to a report writer such as ReportSmith or Crystal Reports to do these tasks, it's still helpful to know to what degree SQL can handle this manipulation itself. First, when evaluating a report writer or the performance of a particular report, it's useful to be able to read the report writer's generated SQL query to determine if it's taking advantage of the full range of SQL statements to off-load as much processing on the server as possible. Secondly, it's important to have the skills to get quick-and-dirty answers when a formal report is overkill.

In the next issue, I'll be outlining some principles of designing client/server systems, highlighting differences in the design of the more conventional software systems most of us are used to.

Later, I plan to return to SQL to see how we can move SQL out of the application entirely and use stored procedures to bind our data manipulation statements directly to the database itself, where any application you develop can take advantage of centralized data processing logic.

Steve Troxell is a Software Engineer with TurboPower Software where he is developing Delphi Client/Server applications using InterBase and Microsoft SQL Server for parent company Casino Data Systems. Steve can be reached on CompuServe at 74071,2207