Surviving Client/Server: Data Processing With Subqueries

by Steve Troxell

Database programmers just coming into the world of SQL and client/server may overlook the subtle power of the SQL language. You may be tempted to look on the SELECT statement as a simple "record read" I/O function which will return a specified set of columns and rows to be further manipulated by the Delphi client program. In reality, SELECT packs a pretty powerful data processing punch of its own. We skimmed the surface of what SELECT can do back in Issue 4 (November 1995). This month we're going to expand our knowledge further and take a look at just how we can leverage the power of SELECT by making use of subqueries.

Subqueries, sometimes referred to as nested queries, can be thought of as chaining two or more queries together with the results of one query feeding into or participating in the processing of the other query. I should say at the outset that the 16-bit BDE with Delphi 1.0 does not support subqueries in local SQL (ie, with Paradox and dBase tables etc), only with database servers.

The usefulness of subqueries might best be explained with an example. Suppose you want a list of stores having above average sales. Let's say we have a table called SalesByStore containing one row per store with the total sales for that store. First, we must obtain the average store sales as shown in Figure 1. Then, we would use this value to create the list of above average stores in Figure 2. We would be tempted to have a Delphi client program run the first query, then substitute the average value obtained into the second query.

However, with a little ingenuity, we can accomplish the same thing by changing Figure 2 to use a subquery as shown in Figure 3. In this case, the inner query (SELECT AVG) is run first to obtain the average value which is then substituted into the outer query. The outer query then runs with a constant value in place of the inner query. All of this happens within the context of a single SQL query sent to the server and a single result set passed back to the client, instead of the two-stage process we had before.

Subqueries don't have to return just a single value either. Suppose you wanted a list of stores with any employee making under $5.00 an hour? Figure 4 shows how to do this. The inner query finds all employees making less than $5.00 an hour and returns a list of Store_IDs where those employees work. Notice that we use the DISTINCT keyword to get a non-duplicating list of store numbers. Otherwise, if a store had more than one employee matching the criteria we would get duplicate Store_IDs in our return set.

```
SalesByStore contains:
Store_ID TotalSales
-------- ----------
6380         132.80
7066       1,821.25
7067       1,486.30
7131       1,400.15
7896         604.40
8042       1,232.00
SELECT Average = AVG(TotalSales) FROM SalesByStore
Average
----------
1,112.82
```

➤ *Figure 1: Obtain the average sales figure*

```
SELECT Store_ID, TotalSales FROM SalesByStore
  WHERE TotalSales > 1112.82
  ORDER BY TotalSales DESC
Store_ID TotalSales
-------- ----------
7066       1,821.25
7067       1,486.30
7131       1,400.15
8042       1,232.00
```

➤ *Figure 2: Stores with above average sales*

```
SELECT * FROM SalesByStore
  WHERE TotalSales > (SELECT AVG(TotalSales)
                      FROM SalesByStore)
  ORDER BY TotalSales DESC
Store_ID TotalSales
-------- ----------
7066       1,821.25
7067       1,486.30
7131       1,400.15
8042       1,232.00
```

➤ *Figure 3: Above average sales with a subquery*

Correlated Subqueries

What if you wanted to find all the stores whose sales are above the average sales for the same city? This is a little different because it's not a simple two-stage process where we can obtain the result of one query and plug that directly into a second query. In this case, the query that determines the average for the city changes as we examine stores in different cities. Figure 5 shows how to make a correlated subquery where the values from the outer query affect the execution of the inner query.

In this query we cannot evaluate the inner query first and then use its results to evaluate the outer query. Here the outer query executes and, as it examines each row in `SalesByStore`, the inner query executes once per row using the `City` value for the current store. Notice that we must use a table alias on the outer query in order to reference it from within the inner query. In its entirety, this query logically states, "For each row in `SalesByStore (S1)`, calculate the average for all stores in the same city as the current store (from `S1`), and return only those stores whose `TotalSales` are greater than the average."

Ranking Data

Let's use a real world example to see how much more we can do with subqueries. Every time someone accesses a page on TurboPower's web site, a "hit" record is written to a transaction table that posts, amongst other things, the date and time of the hit and the page that was hit. In addition a summary file of total hits by page (see Figure 6) is updated for each month via an insert trigger on the transaction table.

Since analysis by month is one of our key purposes in collecting this data, and since the size of the table is relatively small (50 or so web pages by 12 months for a year's worth of data), it is worthwhile to make a separate table for these summary statistics. This also gives us the flexibility to purge the transaction table periodically while still retaining the distilled historical statistics. Retaining historical information like this may not be as important for web page hits, but for product sales, customer order activity, and other similar events, historical summary data may be very important.

Obviously, our marketing folks would very much like to evaluate the effectiveness of the web site by seeing an ordered list of "Top Page Hits" for any given month. With web page hits summarized by month, it is very straightforward to produce this list. We simply run the query shown in Figure 7 and print off the output for the Marketing Guy.

Unfortunately, Marketing Guy returns and asks "How simple is it to put a ranking on these from 1 to whatever so I can easily pull off the top 20?" We can actually get the ranking within the result set itself by using a subquery as shown in Figure 8.

This is a bit different from the subqueries we've looked at so far.

```
SELECT StoreName FROM Stores
  WHERE Store_ID IN (SELECT DISTINCT Store_ID FROM Employees
                       WHERE PayRate < 5.00)

StoreName
------------------------
Videos R Us
Big Bag O' Videos
Videos To Go
Superstar Video Rental
```

➤ *Figure 4*

```
SELECT Store_ID, TotalSales FROM SalesByStore S1
  WHERE TotalSales > (SELECT AVG(TotalSales) FROM SalesByStore
                        WHERE City = S1.City)
  ORDER BY Store_ID

Store_ID TotalSales
-------- ----------
7066     1,821.25
7131     1,400.15
```

➤ *Figure 5*

```
CREATE TABLE WebMonthly(
  Page          varchar(50),
  YearMonth     char(6),      /* YYYYMM */
  TotalHits     int,
  PRIMARY KEY (Page, YearMonth))
```

➤ *Figure 6*

```
SELECT TotalHits, Page FROM WebMonthly
  WHERE YearMonth = '199607'  /* July 1996 */
  ORDER BY TotalHits DESC

TotalHits   Page
----------- ----------
2143        default.htm
773         download.htm
436         products.htm
368         pressrel.htm
296         apd.htm
233         sleuth.htm
199         ordering.htm
185         orpheus.htm
140         systinfo.htm
134         orp21.htm
112         newsletr.htm
85          about.htm
85          orderup.htm
81          order.htm
```

➤ *Figure 7*

Here we are not using a subquery to restrict the rows returned by the outer query, but instead we are using the subquery to compute one of the columns in the result set. This is still a correlated subquery, so the subquery will run for each row processed by the outer query. It is critical that a subquery used in this sense return exactly one row and exactly one column or it wouldn't make sense in the main result set.

The outer query retrieves all rows from the `WebMonthly` table for July 1996, which equates to all the web pages hit in that month. For each row retrieved, the inner query (`SELECT COUNT(TotalHits)`) counts the number of rows in the same month (`YearMonth = W.YearMonth`) having `TotalHits` the same as or higher than the `TotalHits` of the current row of the outer query (`TotalHits >= W.TotalHits`). Therefore, the row with the highest `TotalHits` gets ranked as number 1 because there is only one row with the same or greater `TotalHits` (the row itself). The row with the second highest `TotalHits` gets ranked as number 2 because there are exactly two rows with the same or greater `TotalHits` (the row itself is equal, and the row ranked number 1 is higher).

Finally, we order the list by specifying the column number to order by rather than the column name. Some database server vendors do not allow ordering by a computed column name, but for some reason, it's still legal to order by that same column using the column number. Go figure...

```
SELECT Rank = (SELECT COUNT(TotalHits) FROM WebMonthly
                 WHERE TotalHits >= W.TotalHits AND
                 YearMonth = W.YearMonth),
       TotalHits, Page
  FROM WebMonthly W
  WHERE YearMonth = '199607'   /* July 1996 */
  ORDER BY 1
Rank        TotalHits   Page
----------- ----------- ------------
1           2143        default.htm
2           773         download.htm
3           436         products.htm
4           368         pressrel.htm
5           296         apd.htm
6           233         sleuth.htm
7           199         ordering.htm
8           185         orpheus.htm
9           140         systinfo.htm
10          134         orp21.htm
11          112         newsletr.htm
13          85          about.htm
13          85          orderup.htm
14          81          order.htm
```

➤ *Figure 8*

```
SELECT Rank = (SELECT COUNT(DISTINCT TotalHits) FROM WebMonthly
                 WHERE TotalHits >= W.TotalHits AND
                 YearMonth = W.YearMonth),
       TotalHits, Page
  FROM WebMonthly W
  WHERE YearMonth = '199607'   /* July 1996 */
  ORDER BY 1
Rank        TotalHits   Page
----------- ----------- ------------
1           2143        default.htm
2           773         download.htm
3           436         products.htm
4           368         pressrel.htm
5           296         apd.htm
6           233         sleuth.htm
7           199         ordering.htm
8           185         orpheus.htm
9           140         systinfo.htm
10          134         orp21.htm
11          112         newsletr.htm
12          85          about.htm
12          85          orderup.htm
13          81          order.htm
```

➤ *Figure 9*

Duplicate Values In Rankings

In Figure 8, note that `ABOUT.HTM` and `ORDERUP.HTM` have the same hit count, and therefore have the same ranking (there is a tie for this position). However, the fact that we skip from rank 11 to rank 13 is very disheartening. Why this happens is straightforward. When the query processes the row for `ABOUT.HTM`, there are 11 rows with a higher `TotalHits` and 2 rows with the same `TotalHits` (`ABOUT.HTM` and `ORDERUP.HTM`). Therefore, the rank is 13. The exact same logic applies when `ORDERUP.HTM` is processed. Again, the rank is 13.

How do we correct this and achieve a ranking list with no gaps? All we need to do is add the `DISTINCT` keyword within the inner query as shown in Figure 9. Why does `DISTINCT` correct this? Now, all rows having the same value for `TotalHits` are counted only once as a whole. Now when processing `ABOUT.HTM`, there are 11 rows with a higher `TotalHits` and one distinct match with the same value. Therefore, the ranking is 12. Since rows with the same value are counted only once, you are guaranteed to have an unbroken chain of consecutive rankings.

Ranking Multiple Sequences

So you hand off your new statistical report and lean back in satisfaction thinking "what else could our Marketing Guy possibly want?"

"Great!" he says. "Can you stick the ranking for last month in there too? And show the number of hits from last month as well, so I can chart the up or down movement."

Figure 10 shows just how to do this. Since this is a fairly involved query, I've used variables to hold the current and last month values to make it easier to follow. This script is written for Microsoft SQL Server, there are subtle differences in how other servers handle variables.

At first this query looks very intimidating, but fear not! It's actually more of the same thing we've been doing, just a whole lot of it in the same query.

Our main outer query is the same as before, selecting all the web pages for the given month. Only now we're assigning a table alias of M1 to this set of rows.

The RankThisMonth column is the same subquery as in Figure 9. It is a correlated subquery tied to the current row in M1 which computes the ranking for the current month. The logic is therefore "for every web page in the current month (M1), count the number of other web pages in that month (WHERE YearMonth = M1.YearMonth) that have distinct TotalHits greater than or equal to this page (TotalHits >= M1.TotalHits)."

The RankLastMonth column is a subquery that itself contains a subquery. The outer query retrieves last month's WebMonthly record for the current web page. The inner query then uses that to compute last month's ranking. This is essentially the same as the RankThisMonth subquery, except it is correlated to last month's web page row (M2) instead of this month's (M1). The net result of these two nested queries is a single value representing the ranking of that web page for last month. Note that when a particular page was not present last month, we conveniently get a null value for its ranking and hit count.

The HitsThisMonth column is simply copied from the TotalHits from the current month's web page row (M1). The HitsLastMonth column requires a subquery to retrieve last month's row for the current web page.

To recap, for every row in M1:

- A query is launched to count the number of other rows having the same or greater TotalHits for the same month (RankThisMonth);
- A query is launched to retrieve the matching web page row for the previous month and then another query is launched to count the number of other rows having the same or greater TotalHits for last month (RankLastMonth);
- Finally, a fourth query is run to retrieve the matching web page row for the previous month (again) in order to get last month's total hits.

```
DECLARE @vThisMonth char(6)
DECLARE @vLastMonth char(6)
SELECT @vThisMonth = '199607'  /* July 1996 */
SELECT @vLastMonth = '199606'  /* June 1996 */
SELECT
  RankThisMonth = (SELECT COUNT(DISTINCT TotalHits)
                   FROM WebMonthly
                   WHERE YearMonth = M1.YearMonth AND
                         TotalHits >= M1.TotalHits),
  RankLastMonth = (SELECT (SELECT COUNT(DISTINCT TotalHits)
                           FROM WebMonthly
                           WHERE YearMonth = M2.YearMonth AND
                                 TotalHits >= M2.TotalHits)
                   FROM WebMonthly M2
                   WHERE YearMonth = @vLastMonth AND
                         Page = M1.Page),
  HitsThisMonth = M1.TotalHits,
  HitsLastMonth = (SELECT TotalHits FROM WebMonthly
                   WHERE YearMonth = @vLastMonth AND
                         Page = M1.Page),
  M1.Page
  FROM WebMonthly M1
  WHERE YearMonth = @vThisMonth
  ORDER BY 1, 2

RankThisMonth RankLastMonth HitsThisMonth HitsLastMonth Page
------------- ------------- ------------- ------------- ------------
1             1             2143          2066          default.htm
2             2             773           874           download.htm
3             3             436           526           products.htm
4             4             368           509           pressrel.htm
5             5             296           341           apd.htm
6             (null)        233           (null)        sleuth.htm
7             7             199           200           ordering.htm
8             6             185           207           orpheus.htm
9             8             140           185           systinfo.htm
10            9             134           183           orp21.htm
11            10            112           125           newsletr.htm
12            11            85            111           orderup.htm
12            12            85            92            about.htm
13            13            81            77            order.htm
14            (null)        71            (null)        sl-scrn1.htm
15            15            53            54            delphi32.htm
15            15            53            54            gi.htm
16            14            50            61            btree.htm
```

➤ *Figure 10*

Obviously, a query such as this is a good candidate for a stored procedure, with parameters for the two months to compare. Note this algorithm is not limited to comparing consecutive months: any two points in time can be compared (for example, current month and same month last year).

Complicated queries such as this may perform slowly for large sets of data depending on how widely the values are distributed and how well you indexed the tables. But for the type of data we've examined here and similar data summaries (product sales by product and month, for example), performance should not be much of a problem. Also, bear in mind that these types of query are not run very frequently.

Conclusion

We've seen how to use subqueries to add another dimension of processing power to our SQL queries. Subqueries can generally be used anywhere an expression compatible with the result of the subquery can be used. Next month, we'll continue the theme by seeing how to work with case functions, running totals and cross tabulations.

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