Take advantage of SQL improvements!

Recent versions of VFP add significant functionality to VFP's SQL commands. Using these abilities can improve your code.

Tamar E. Granor, Ph.D.

When SQL commands were added in FoxPro 2, it didn't take me long to see that they could make writing code easier. SQL SELECT, in particular, was very appealing since it made it possible to retrieve the data I needed by specifying what I wanted rather than how to find it.

Over the years, VFP's SQL commands have become an important part of my programming arsenal. The list of commands supported and the functionality of the individual commands has increased over the years.

FoxPro 2 included CREATE TABLE, CREATE CURSOR, SELECT and INSERT. VFP 3 added ALTER TABLE, SQL DELETE and UPDATE commands.

After VFP 3, the SQL commands stayed pretty much the same until VFP 8, which introduced a few new clauses, and some new rules, as well. VFP 9 added significant functionality to the SELECT, DELETE and UPDATE commands.

In this article, I'll show you how some of the recent changes can simplify your code.

Adding query data to a table or cursor

VFP 8 added two new ways to insert data into a table or cursor with the SQL INSERT command. One is a standard SQL capability, while the other is a VFP-specific extension.

The standard item lets you run a query and add the query results directly to a cursor or table. The syntax is shown in Listing 1.

Listing 1. VFP 8 added the INSERT INTO SELECT syntax that lets you gather a set of data and add it to an existing table in one step.

```
INSERT INTO <table> [(<field list>])
   SELECT <field list>
     FROM <tables and join conditions>
     <rest of query as usual>
```

Previously, this had to be done in two steps, first running the query, and then using APPEND FROM to add the data.

For example, one of my responsibilities for the Southwest Fox conference is compiling the conference evaluation results and producing the various reports we draw from them. The first step, though, is data entry. Doug Hennig, Rick Schummer, and I each take a third of the evaluation booklets and enter the data; it's stored, of course, in VFP tables. Then, Doug and Rick send me their tables. I need to combine the three sets of tables to form one complete set for analysis.

Because the tables have an auto-incrementing primary key, I can't use APPEND FROM to simply take the tables Doug and Rick send and add them to my table; auto-incrementing fields are read-only. So, prior to VFP 8, this task would have required code like Listing 2 (note that you must have SET FULLPATH ON for this code to work).

Listing 2. Before VFP 8, grabbing data from one table and adding it to another was a two-step process.

```
SELECT iTimeslot, iSpeaker, iTopic, ;
       mLikes, mDislikes, nPrepared, ;
       nKnowledge, nInteresti, nMatchDesc, ;
       nValuable, nRelevant, nAgain, ;
       mComments ;
  FROM DougSessionEval ;
  INTO CURSOR csrDougsEvals NOFILTER

SELECT SessionEval
APPEND FROM DBF("csrDougsEvals")
```

With the new syntax, I can pull this into a single command, as in Listing 3.

Listing 3. In VFP 8 and 9, you can collect the data and insert it in a single step.

```
INSERT INTO SessionEval ;
   (iTimeslot, iSpeaker, iTopic, ;
    mLikes, mDislikes, nPrepared, ;
    nKnowledge, nInteresti, nMatchDesc, ;
    nValuable, nRelevant, nAgain, ;
    mComments) ;
  SELECT iTimeslot, iSpeaker, iTopic, ;
         mLikes, mDislikes, nPrepared, ;
         nKnowledge, nInteresti, nMatchDesc, ;
         nValuable, nRelevant, nAgain, ;
         mComments ;
    FROM DougSessionEval
```

As in the old version, I have to list all the fields I want in order to avoid trying to write to the auto-incrementing primary key field. However, this version avoids opening an intermediate cursor and leaving it open.

The second change to INSERT is the addition of the FROM NAME clause. This allows you to add data to a cursor or table directly from an object. The object's properties are mapped to same-named fields. The syntax is shown in Listing 4.

Listing 4. The new FROM NAME clause for INSERT lets you move data directly from objects into a cursor or table.

```
INSERT INTO <table> FROM NAME <object>
```

I tend to use this approach more in manipulation of data from the Command Window than in applications (though I have used it there). I find it most useful when I need to make a near-copy of an existing record, as in Listing 5, or when I want to copy a record from one table to another with the same structure, as in Listing 6.

Listing 5. INSERT INTO FROM NAME is very handy for making near-copies of existing records.

```
USE MyTable

* Find the right record somehow, then
SCATTER NAME oRec MEMO
WITH m.oRec
   * Change some properties/fields
ENDWITH

* Add the modified record
INSERT INTO MyTable FROM Name m.oRec
```

Listing 6. INSERT INTO FROM Name is also useful for moving data from one table to another.

```
USE Source
SCATTER NAME oRec MEMO

INSERT INTO Dest FROM Name m.oRec
```

Create cursors on the fly

VFP 9 introduced a number of SQL changes. The one I've found most useful is the ability to create *derived tables* in SQL commands, especially queries.

A derived table is a query in the FROM clause of another SQL command. It works like any other query, except that instead of saving the results in a cursor or table, they're used in the containing command and then discarded. The basic structure of a query with a derived table is shown in Listing 7.

Listing 7. A derived table is a query in the FROM clause of another SQL command.

```
SELECT <field list> ;
  FROM <table> ;
    JOIN (SELECT <field list> ;
            FROM <table> ;
            <additional clausess ;
         ) <local alias> ;
      ON <join condition> ;
  <rest of query>
```

As with INSERT INTO … SELECT, derived tables let you do in one command what previously required several. I use them extensively in preparing data for the speaker evaluation report. Listing 8 shows one example. Southwest Fox speakers are quite competitive and want to know how they did in comparison with their peers. So the report we provide them includes the ranking of their sessions and themselves overall. The query here prepares the overall speaker rankings. The derived table does the actual computation of the rankings; the main query attaches that data to the speaker information (stored in the Speaker table) and sorts in rank order. (The record numbers in the resulting cursor, SpeakAvgs, are what we actually report to the speakers.)

Listing 8. The derived table in this query computes the average evaluation for each speaker, which is used to rank the speakers.

```
SELECT SessAvg.iSpeaker, Speaker.cFirst, ;
       Speaker.cLast, SessAvg.nTotal ;
   FROM ;
      (SELECT iSpeaker, CNT(*), ;
          AVG(nPrepared + ;
              nKnowledge + ;
              nInteresti + ;
              nMatchDesc + ;
              nValuable + ;
              nRelevant) as nTotal;
          FROM SessionEval ;
          GROUP BY 1 ) SessAvg ;
     JOIN Speaker ;
       ON SessAvg.iSpeaker = Speaker.iID ;
     ORDER BY nTotal DESC ;
   INTO CURSOR SpeakAvgs
```

In VFP 8 and earlier, this task would have required two queries in sequence. The first query is the one that's now the derived table, with the results stored in a cursor. Then, the second query uses those results. Listing 9 shows the older way to do this.

Listing 9. In VFP 8 and earlier, computing the rankings requires two queries.

```
SELECT iSpeaker, CNT(*), ;
       AVG(nPrepared + nKnowledge + ;
           nInteresti + nMatchDesc + ;
           nValuable + nRelevant) as nTotal;
   FROM SessionEval ;
   GROUP BY 1 ;
   INTO CURSOR SessAvg

SELECT SessAvg.iSpeaker, Speaker.cFirst, ;
       Speaker.cLast, SessAvg.nTotal ;
   FROM SessAvg ;
     JOIN Speaker ;
       ON SessAvg.iSpeaker = Speaker.iID ;
   ORDER BY nTotal desc ;
   INTO CURSOR SpeakAvgs
```

The derived table version offers a couple of advantages. First, anything you can write as a single query is easier to use as a view, which may be important for some situations. In addition, VFP cleans up after derived tables. So when you run the query in Listing 8, the only cursor it leaves open is the result, SpeakAvgs. With Listing 9, the SessAvg cursor is open afterward as well.

To use a derived table, wrap the query in parentheses (as you generally do for sub-queries). In addition, you must provide a *local alias*, that is, an alias to use for the derived table result within the containing command. In Listing 8, the local alias is SessAvg. Note also that you can have multiple derived tables in a single query.

Derived tables can be used in SQL DELETE and SQL UPDATE, not just for queries. So you can use queries to figure out which records to delete or update.

For example, for the speaker evaluations, I have several queries like the one in Listing 8. Another computes the rankings for session, and a third computes the overall averages for each category, which we provide in the report as a comparison. I index these cursors and use SET RELATION to connect them to the driving cursor for the report (called SessionAvgs, it contains the average ratings for each evaluation category for each individual session), as shown in Listing 10.

Listing 10. The old way to connect data in two cursors for reporting is via SET RELATION. In my code to report speaker evaluations, these lines follow the query in Listing 8.

```
INDEX ON PADL(iSpeaker, 3) + PADL(iTopic,3) ;
         TAG SpeakTopic
SELECT SessionAvgs
SET RELATION TO ;
   PADL(iSpeaker,3) + PADL(iTopic,3) ;
   INTO SessAvgs ADDITIVE
```

However, I could instead put extra fields to hold the rankings for each session in the driving cursor, and use UPDATE to fill them in. Listing 11 shows how to use the UPDATE command to fill the speaker rankings into SessionAvgs.

Listing 11. Instead of storing the speaker ranks in a separate cursor as in Listing 8, we could stuff the results right into fields of an overall result cursor.

```
UPDATE SessionAvgs ;
  SET nSpeakerRank = SpeakRank.nRank ;
  FROM (;
   SELECT iSpeaker, RECNO() as nRank ;
      FROM (SELECT iSpeaker, ;
              AVG(nPrepared + nKnowledge + ;
                  nInteresti + nMatchDesc + ;
                  nValuable + nRelevant) ;
                  AS nTotal;
              FROM SessionEval ;
              GROUP BY 1 ;
              ORDER BY nTotal Desc ;
           ) ComputeSpeakAvg ;
       ) SpeakRank ;
   WHERE SessionAvgs.iSpeaker = ;
         SpeakRank.iSpeaker
```

This command actually uses two derived tables. The inner query calculates the average total ranking per speaker and puts them in descending order with a local alias of ComputeSpeakAvg. The outer query pulls just the speaker ID and the position of the speaker in the list (that is, the speaker's rank) from that result into a derived table with a local alias of SpeakRank. Then, the UPDATE command joins that result to the existing cursor, SessionAvgs, based on matching the speaker ID and fills in SessionAvgs.nSpeakerRank.

Compute fields with subqueries

Starting in VFP 9, you can use a subquery in the field list of a query, in order to compute the field value. This capability is called *projection*.

Projection is most useful when you want to base a column on filtered and/or aggregated data. For example, projection lets me see whether speakers do significantly differently in their first presentation of sessions than in the second. The query in Listing 12 shows how to do this. There are four fields in the field list. The first two are straightforward, but the last two are nearly identical subqueries. The first computes the average score for a given speaker in the initial presentation of sessions; the second computes the average score for the speaker in the repeat. (A few notes about the additional tables used in this query. TimeSlot is a master list of available timeslots at the conference. Each record represents a single timeslot. Schedule links TimeSlot to the actual sessions, with one record for each presentation of each topic. Note that to avoid additional complexity, these queries take a shortcut, assuming that the primary key in the TimeSlot table is in time-sequence order. It would be better, but more complicated, code to actually check the field of TimeSlot that specifies the starting time of the session.)

Listing 12. This complex looking query uses two subqueries in the field list to compute the average rating for each speaker in the initial presentation of a session and the average in the repeat presentation.

```
SELECT iSpeaker, ;
       AVG(nPrepared + nKnowledge + ;
           nInteresti + nMatchDesc + ;
           nValuable + nRelevant) as nTotal, ;
       (SELECT AVG(nPrepared + nKnowledge + ;
                   nInteresti + nMatchDesc + ;
                   nValuable + nRelevant) ;
          FROM SessionEval SE ;
          WHERE SE.iSpeaker = ;
                SessionEval.iSpeaker ;
            AND SE.iTimeSlot = ;
              (SELECT MIN(TimeSlot.iID) ;
                  FROM TimeSlot ;
                    JOIN Schedule ;
                      ON TimeSlot.iID = ;
                       Schedule.iSlot ;
                 WHERE Schedule.iTopic = ;
                       SE.iTopic)) ;
         AS nSlot1, ;
       (SELECT AVG(nPrepared + nKnowledge + ;
                   nInteresti + nMatchDesc + ;
                   nValuable + nRelevant) ;
          FROM SessionEval SE ;
          WHERE SE.iSpeaker = ;
                SessionEval.iSpeaker ;
            AND SE.iTimeSlot = ;
              (SELECT MAX(TimeSlot.iID) ;
                  FROM TimeSlot ;
```

```
                    JOIN Schedule ;
                      ON TimeSlot.iID = ;
                         Schedule.iSlot ;
                 WHERE Schedule.iTopic = ;
                       SE.iTopic)) ;
         AS nSlot2 ;
      FROM SessionEval ;
      GROUP BY 1 ;
INTO CURSOR csrAverageByRepeat
```

But wait, there's more

VFP 9 introduced several other new capabilities for SQL commands, including the ability to use the TOP N clause in some subqueries and allowing GROUP BY in correlated subqueries. If the SQL commands are part of your VFP arsenal, be sure to check the "SQL Language Improvements" section of the VFP 9 help file to see what you've missed. If the SQL commands aren't something you're using regularly, it's time to take a good, hard look at them.

Author Profile

Tamar E. Granor, Ph.D. is the owner of Tomorrow's Solutions, LLC. She has developed and enhanced numerous Visual FoxPro applications for businesses and other organizations. She currently focuses on working with other developers through consulting and subcontracting. Tamar is author or co-author of ten books including the award winning Hacker's Guide to Visual FoxPro, Microsoft Office Automation with VisualFoxPro and Taming Visual FoxPro's SQL. Her latest collaboration is Making Sense of Sedna and SP2, coming out this year. Her books are available from Hentzenwerke Publishing (www.hentzenwerke.com). Tamar is a Microsoft Support Most Valuable Professional. In 2007, Tamar received the Visual FoxPro Community Lifetime Achievement Award. You can reach her at tamar@thegranors.com or through www.tomorrowssolutionsllc.com.