Session E-SQL

Solving Common Problems with Visual FoxPro's SQL

Tamar E. Granor, Ph.D.
Tomorrow's Solutions, LLC
tamar@tomorrowssolutionsllc.com

Introduction

Visual FoxPro's SQL language lets you solve a variety of problems in database applications. This session will explore common problems, and see how to use SQL to solve them. It focuses on working with VFP data, and looks at many of the SQL features new to VFP 8 and 9.

All of the examples in this white paper are included in the materials for this session. The name of the file for each is included in parentheses in the listing caption.

Preparing for reporting

One of the most common uses for SQL, particularly for SELECT, is preparing data for reports. This section looks at some of the trickier items that come up in massaging data before running a report.

Reporting on hierarchical data

Certain kinds of hierarchical data are best stored by including multiple levels of the hierarchy in a single table. Typical examples include organizational charts for a business, where each employee record includes a pointer to the employee's supervisor, who is also an employee; and family relationships, where each person may be related to other people in one of several ways.

While this structure is handy for storing the data, reporting on it is trickier. In fact, if there are more than two levels of data, you can't use a single query to extract the whole hierarchy.

To explore this problem, we'll use a simple table called Employee. The structure is shown in Listing 1. iID is the primary key and iSuperviso is the supervisor's primary key.

Listing 1. This Employee table is hierarchical, with each record pointing to another record in the same table, the employee's supervisor. (Employee.DBF)

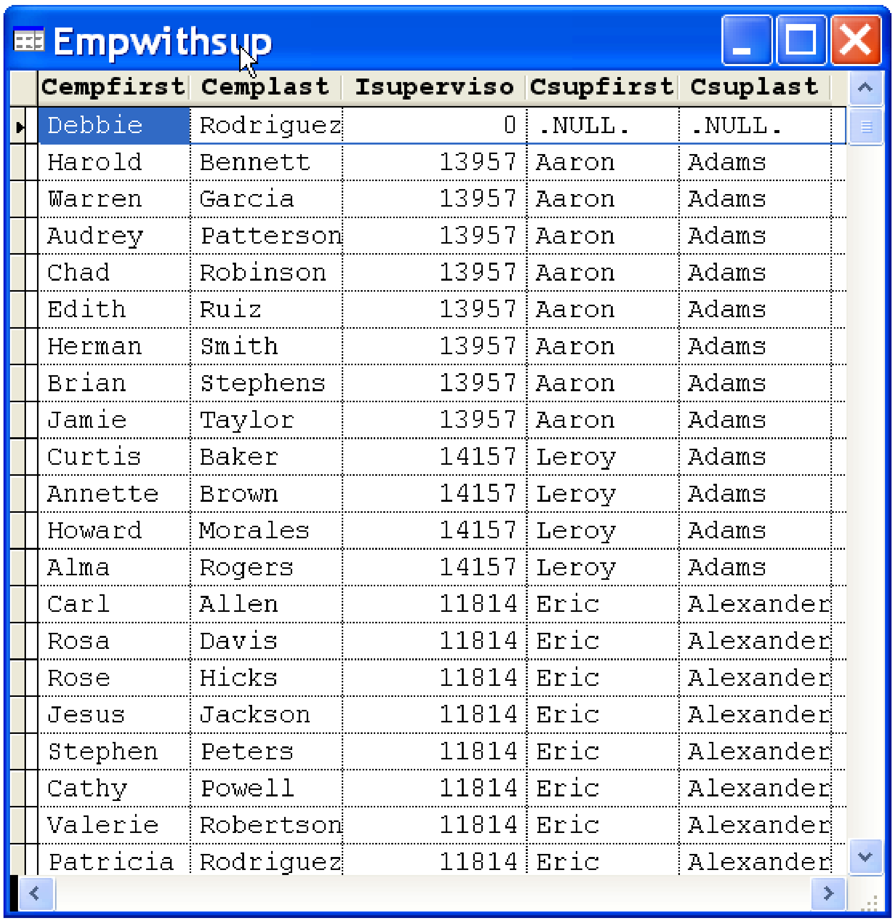
```
Field Name          Type                 Width
IID                 Integer (AutoInc)        4
CFIRST              Character               15
CLAST               Character               20
ISUPERVISO          Integer                  4
```

We'll start with the simplest problem. How do we get the name of each employee and his or her immediate supervisor? It takes a self-join, but is fairly straightforward. In a self-join, one instance of a table is joined to another instance of itself. Local aliases in the FROM clause distinguish the two instances from each other; once a local alias is assigned to a table, that alias must be used throughout the query. The AS clause in the field list renames fields in the result to avoid duplication.

The query in Listing 2 uses a left outer join so that all employees, including the head honcho (the person at the top of the hierarchy), are included. The first few records of the results are shown in Figure 1.

Listing 2. Finding each employee and his or her supervisor requires a self-join. (EmpWithSup.PRG)

```
SELECT Emp.cFirst AS cEmpFirst, Emp.cLast AS cEmpLast, Emp.iSupervisor;
       Sup.cFirst AS cSupFirst, Sup.cLast AS cSupLast;
   FROM Employee Emp ;
     LEFT JOIN Employee AS Emp
       ON Emp.iSuperviso = Sup.iID ;
   ORDER BY cSupLast, cSupFirst, cEmpLast, cEmpFirst ;
   INTO CURSOR EmpWithSup
```



Cempfirst	Cemplast	Isuperviso	Csupfirst	Csuplast
Debbie	Rodriguez	0	.NULL.	.NULL.
Harold	Bennett	13957	Aaron	Adams
Warren	Garcia	13957	Aaron	Adams
Audrey	Patterson	13957	Aaron	Adams
Chad	Robinson	13957	Aaron	Adams
Edith	Ruiz	13957	Aaron	Adams
Herman	Smith	13957	Aaron	Adams
Brian	Stephens	13957	Aaron	Adams
Jamie	Taylor	13957	Aaron	Adams
Curtis	Baker	14157	Leroy	Adams
Annette	Brown	14157	Leroy	Adams
Howard	Morales	14157	Leroy	Adams
Alma	Rogers	14157	Leroy	Adams
Carl	Allen	11814	Eric	Alexander
Rosa	Davis	11814	Eric	Alexander
Rose	Hicks	11814	Eric	Alexander
Jesus	Jackson	11814	Eric	Alexander
Stephen	Peters	11814	Eric	Alexander
Cathy	Powell	11814	Eric	Alexander
Valerie	Robertson	11814	Eric	Alexander
Patricia	Rodriguez	11814	Eric	Alexander

Figure 1. The query in Listing 2 matches each employee with his or her supervisor.

Finding the individual at the top of the hierarchy is no problem; just find the employee with no supervisor. Listing 3 shows the way.

Listing 3. With this structure, finding the ultimate boss is simple. (HeadHoncho.PRG)

```
SELECT cFirst, cLast ;
   FROM Employee ;
```

```
      WHERE EMPTY(iSuperviso) ;
      INTO CURSOR HeadHoncho
```

Finding those employees who are at the lowest level is fairly easy, too. The query in Listing 4 does the trick; it uses a subquery to figure out which employees are supervisors and then keeps only the rest.

Listing 4. Figuring out which employees are worker bees, those who supervise no one else, involves a subquery. (WorkerBees.PRG)

```
SELECT cFirst, cLast ;
   FROM Employee ;
   WHERE iID NOT IN ;
     (SELECT iSuperviso FROM Employee) ;
   ORDER BY cLast, cFirst ;
   INTO CURSOR WorkerBees
```

Figuring out which employees are supervisors is almost identical, but the subquery uses IN rather than NOT IN; Listing 5 shows the query. Although this query could be done with a self-join rather than a subquery, the subquery ensures that each supervisor is listed only once in the results, avoiding the use of DISTINCT, which can be slow.

Listing 5. Finding supervisors is almost the same as finding worker bees. (Supervisors.PRG)

```
SELECT cFirst, cLast ;
   FROM Employee ;
   WHERE iID IN ;
     (SELECT iSuperviso FROM Employee) ;
   ORDER BY cLast, cFirst ;
   INTO CURSOR AllSupervisors
```

After this, things get trickier. You can write queries to find supervisors at a specific level, but starting with a single employee and tracing the hierarchy all the way to the top can't be done with a single query, unless you know how many levels are involved. (There are more complex ways to store hierarchical data that allow you to trace the hierarchy with a single query. The price of doing so is that maintaining the data is more complex.)

One solution is to combine SQL with a little Xbase code, as in Listing 6. A query finds the next level, and a loop keeps the process going until you reach the top.

Listing 6. You can build a hierarchy from one employee to the top of the heap using a query inside a loop. (EmpHierarchy.PRG)

```
LPARAMETERS iEmpID

LOCAL iCurrentID

CREATE CURSOR EmpHierarchy (cFirst C(15), cLast C(20))

iCurrentID = iEmpID
DO WHILE NOT EMPTY(iCurrentID)

   SELECT cFirst, cLast, iSuperviso ;
      FROM Employee ;
       WHERE iID = m.iCurrentID ;
       INTO CURSOR NextEmp

   INSERT INTO EmpHierarchy ;
      VALUES (NextEmp.cFirst, NextEmp.cLast)

   iCurrentID = NextEmp.iSuperviso
ENDDO
```

In VFP 8 and later, an alternative is to use INSERT INTO … SELECT to combine the two operations inside the loop. To do so, we either have to add the supervisor's ID to the result cursor or search the data twice on each pass through the loop. The version in Listing 7 takes the first approach, putting the supervisor id in the results. In my tests, this version is about 30% faster than using a separate SELECT followed by INSERT.

Listing 7. Using INSERT INTO … SELECT to build the hierarchy is faster, at the cost of an extra field in the results. (EmpHierarchy2.PRG)

```
LPARAMETERS iEmpID
```

```
LOCAL iCurrentID

CREATE CURSOR EmpHierarchy (cFirst C(15), cLast C(20), iSuperviso I)

iCurrentID = iEmpID
DO WHILE NOT EMPTY(iCurrentID)

   INSERT INTO EmpHierarchy ;
      SELECT cFirst, cLast, iSuperviso ;
         FROM Employee ;
          WHERE iID = m.iCurrentID ;

   iCurrentID = EmpHierarchy.iSuperviso
ENDDO
```

Of course, using a query to select a single record is unnecessarily complex. The version in **Listing 8** combines SEEK with INSERT to get the same results. This version is slightly faster (about 6%) than the previous version; the downside is having to open a table explicitly, and manage its work area.

Listing 8. The fastest solution for building the hierarchy is to combine the Xbase SEEK with SQL's INSERT. (EmpHierarchyProc.PRG)

```
LPARAMETERS iEmpID

LOCAL iCurrentID

CREATE CURSOR EmpHierarchy (cFirst C(15), cLast C(20))

USE Employee IN 0 ORDER iID

iCurrentID = iEmpID
DO WHILE NOT EMPTY(iCurrentID)

   SEEK iCurrentID IN Employee

   INSERT INTO EmpHierarchy ;
      VALUES (Employee.cFirst, Employee.cLast)

   iCurrentID = Employee.iSuperviso
ENDDO

USE IN Employee
SELECT EmpHierarchy
```

Dealing with Complex Hierarchies

Some hierarchies can't be represented by a single table. For example, a bill of materials is typically handled with two tables. One table (Items) lists all the items, while the second (Contents) shows which items are combined to make other items. The Contents table indicates which container item (iContID) you’re building, which item (iItemID) goes into that container item, and how many you need (nQuantity). Figure 2 shows a simple database to represent this situation. Figure 3 and Figure 4 show the contents of the two tables.

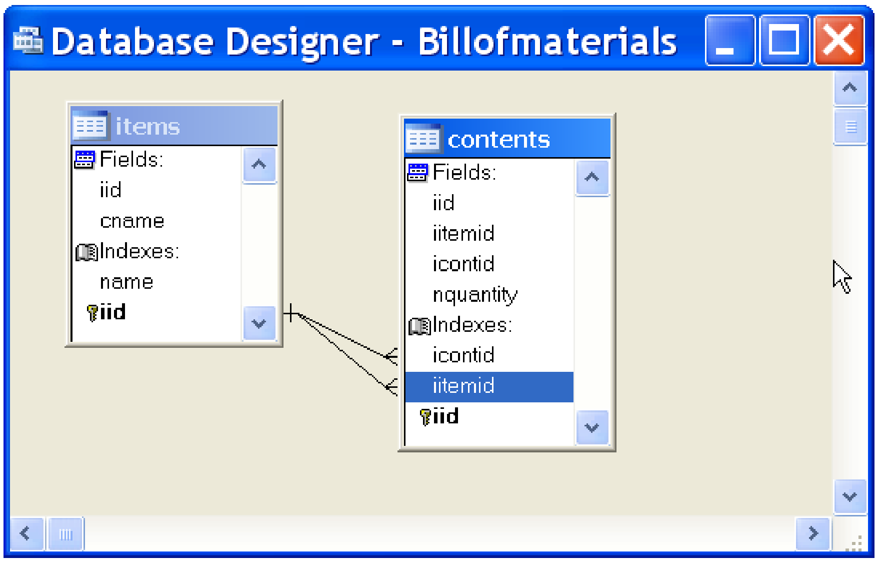
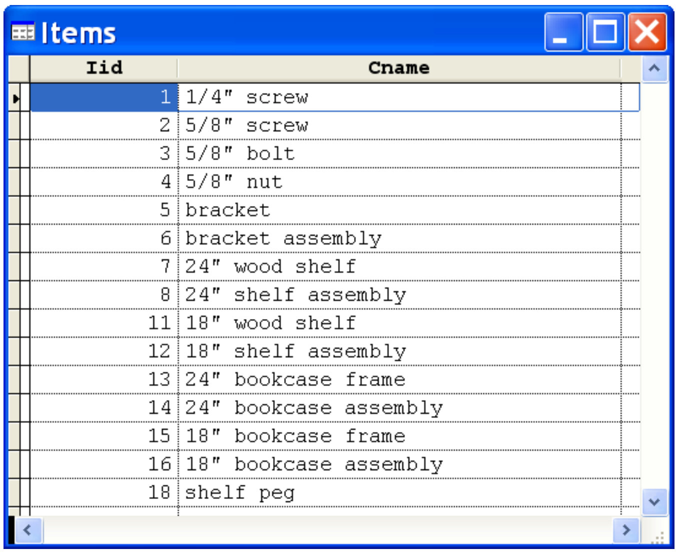


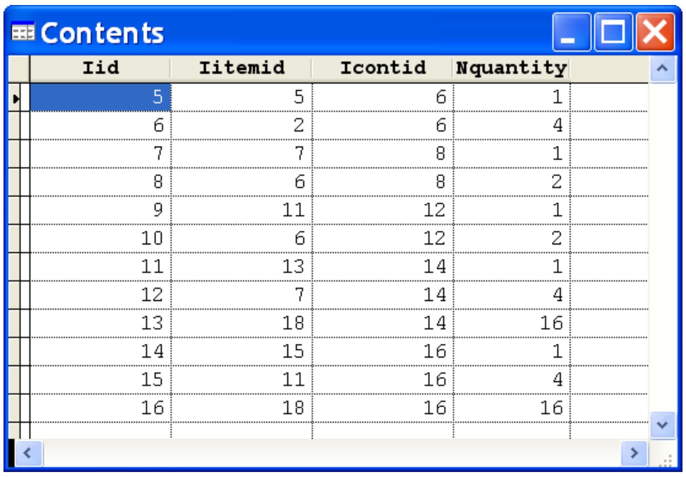
Figure 2. You normally use two tables to represent a bill of materials situation, where items can be constructed out of other items. (BillOfMaterials.DBC)



Items

Iid	Cname
1	1/4" screw
2	5/8" screw
3	5/8" bolt
4	5/8" nut
5	bracket
6	bracket assembly
7	24" wood shelf
8	24" shelf assembly
11	18" wood shelf
12	18" shelf assembly
13	24" bookcase frame
14	24" bookcase assembly
15	18" bookcase frame
16	18" bookcase assembly
18	shelf peg

Figure 3. The Items table lists all the parts that can be used. (Items.DBF)



Contents

Iid	Iitemid	Icontid	Nquantity
5	5	6	1
6	2	6	4
7	7	8	1
8	6	8	2
9	11	12	1
10	6	12	2
11	13	14	1
12	7	14	4
13	18	14	16
14	15	16	1
15	11	16	4
16	18	16	16

Figure 4. The Contents table shows which items go into building other items. (Contents.DBF)

You're most likely to want to start from the top and find all the components of an item in order to build a part list. As you move downward, you have to check each item you find to see whether it contains additional items. Again, you can't do this with a single query. In this case, though, you can't just adopt the last item id found as the next one to search. You have to keep track of which items you've checked for components. Listing 9 shows one solution. It creates a cursor to hold the parts found so far, then loops through that cursor, checking each item to see whether it's composed of smaller items.

Listing 9. To build a part list, you need to drill down checking each item you find to see whether it's composed of additional parts. (PartList.PRG)

```
LPARAMETERS iItemID, lDeleteAssemblies

LOCAL iCurrentID, nCurPart, lDeleteCurRec

iCurrentID = m.iItemID

SELECT iItemID, cName, nQuantity, m.iCurrentID AS iAssemblyID, ;
       000 AS nSortOrder ;
   FROM Contents ;
     JOIN Items ;
       ON Contents.iItemID = Items.iID ;
   WHERE Contents.iContID = m.iCurrentID ;
   INTO CURSOR Parts READWRITE

nSortOrder = 1

SCAN

   nCurPart = RECNO("Parts")
   iCurrentID = Parts.iItemID
   nHowMany = Parts.nQuantity
   IF nSortOrder = 0
      REPLACE nSortOrder WITH m.nSortOrder
   ENDIF

   INSERT INTO Parts ;
      SELECT iItemID, cName, m.nHowMany * nQuantity, m.iCurrentID, ;
             m.nSortOrder + 1 ;
         FROM Contents ;
           JOIN Items ;
             ON Contents.iItemID = Items.iID ;
         WHERE Contents.iContID = m.iCurrentID
   lDeleteCurRec = lDeleteAssemblies AND (_TALLY > 0)
   m.nSortOrder = m.nSortOrder + _TALLY + 1

   GO (nCurPart) IN Parts
   IF lDeleteCurRec
      DELETE NEXT 1
   ENDIF

ENDSCAN

SELECT * FROM Parts ;
   INTO CURSOR Parts ;
   ORDER BY nSortOrder
```

There are several interesting things going on in this code. First, if you choose, you can remove the records for any assemblies (items that are composed of additional parts) and keep only the lowest-level records.

Presumably, the part list should tell the total number of each item needed, so the query multiplies the number for a given assembly by the quantity of that assembly.

In addition, the code sorts the results so that if records for assemblies are there, the parts that comprise the assembly are listed immediately after the assembly. That makes it possible to create more meaningful reports.

The other question you may want to ask in this situation is which items use a particular part. As with the employee-supervisor question, if you only need to go back one level, a single query (shown in Listing 10) suffices.

Listing 10. You can find the assemblies that use each item directly with a single query. (UsedDirectly.PRG)

```
SELECT Items.iID, Items.cName, iContID, Assem.cName ;
   FROM Items ;
     JOIN Contents ;
```

```
        ON Items.iID = Contents.iItemID ;
      JOIN Items Assem ;
        ON Contents.iContID = Assem.iID ;
    ORDER BY Assem.iID ;
    INTO CURSOR UsedDirectly
```

However, if you want to know all the final products that use a particular part, you have to trace the hierarchy. As in Listing 9, at each step, there may be several items to check. The version in Listing 11 uses two cursors—one to hold the list of items to check and the other to hold the results.

Listing 11. To find the actual products that use a particular item, you have to trace up the hierarchy, but it's very similar to tracing down. (UsesItem.PRG)

```
LPARAMETERS iItemID

LOCAL iCurrentID, nRecNo

CREATE CURSOR UsesItem (iItemID I, cName C(25))

iCurrentID = m.iItemID

SELECT iContID ;
   FROM Contents ;
   WHERE iItemID = m.iCurrentID ;
   INTO CURSOR CheckItems READWRITE

SCAN
   iCurrentID = CheckItems.iContID
   nRecNo = RECNO("CheckItems")

   * Look for items containing the current item
   INSERT INTO CheckItems ;
      SELECT iContID ;
         FROM Contents ;
         WHERE iItemID = m.iCurrentID
   GO nRecNo IN CheckItems

   IF _Tally = 0
      * This is the top of the chain
      INSERT INTO UsesItem ;
         SELECT iID, cName ;
            FROM Items ;
            WHERE iID = m.iCurrentID
   ENDIF

ENDSCAN
```

The code to list all items that contain a particular item, not just the final products, is surprisingly similar to the code that produces the part list. It's shown in Listing 12.

Listing 12. Producing a list of all items that use a particular part is almost identical to producing a part list. (ContainerList.PRG)

```
LPARAMETERS iItemID

LOCAL iCurrentID

iCurrentID = m.iItemID

SELECT iContID as iItemID, cName ;
   FROM Contents ;
     JOIN Items ;
       ON Contents.iContID = Items.iID ;
   WHERE Contents.iItemID = m.iCurrentID ;
   INTO CURSOR Containers READWRITE

SCAN

   nCurPart = RECNO("Containers")
```

```
      iCurrentID = Containers.iItemID

      INSERT INTO Containers ;
         SELECT iContID, cName ;
            FROM Contents ;
              JOIN Items ;
                ON Contents.iContID = Items.iID ;
            WHERE Contents.iItemID = m.iCurrentID

      GO (nCurPart) IN Containers

   ENDSCAN
```

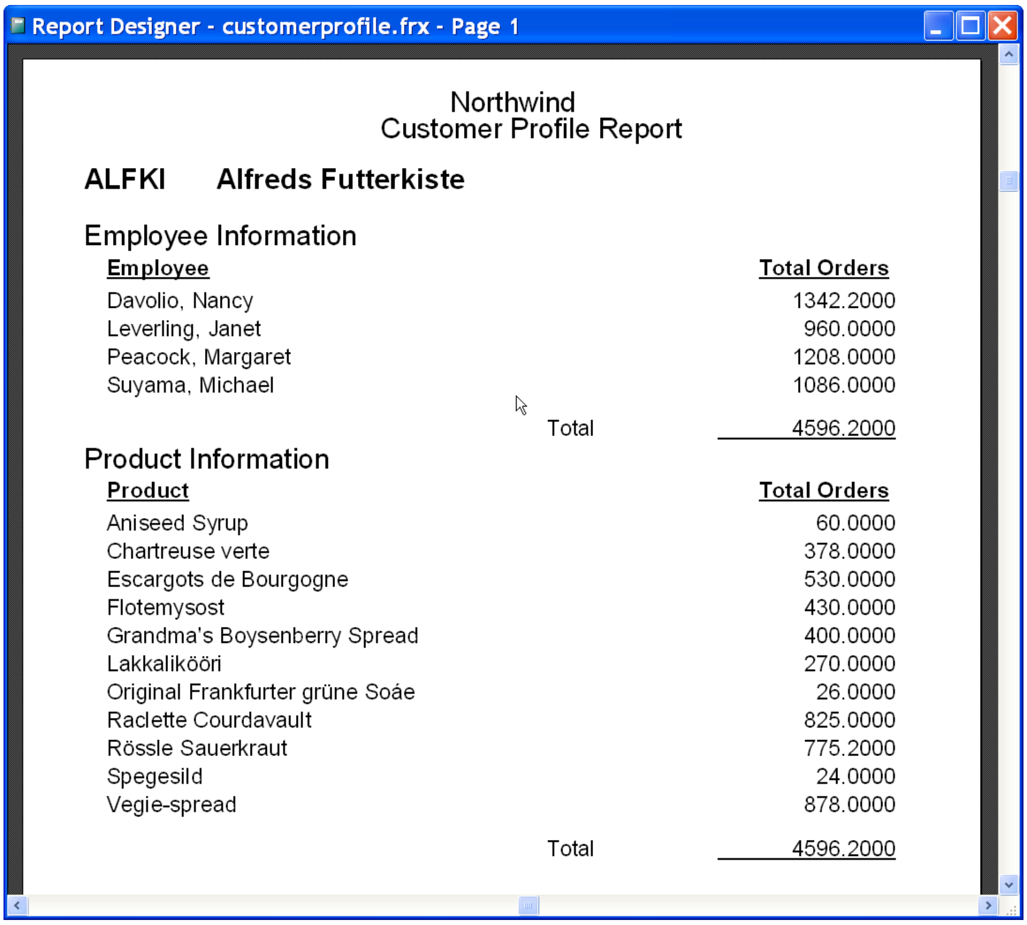
Overall, hierarchical data demonstrates the value of having both SQL's set-orientation and VFP's record orientation available.

Reporting on one parent table and all its child tables

Until VFP 9, FoxPro didn't offer the possibility of putting multiple detail bands into a report. Yet many data sets call for a report that looks as if it has more than one detail band. For example, a customer statement might show invoices and payments in separate sections.

The secret to preparing this kind of report is to get all the data into a single cursor in the order in which you want it to appear in the report. For the customer statement described above, you'd want each customer's data with invoices listed first, then payments. Once you have the data ordered properly, you can use grouping in the Report Designer to give the impression of multiple detail bands.

To demonstrate, consider a "customer profile" report for the Northwind database. The report includes all customers; for each, it shows first a list of employees with whom that customer has placed any orders and then a list of the products ordered by the customer. Figure 5 shows one page of the report.



Northwind
Customer Profile Report

ALFKI   Alfreds Futterkiste

Employee Information

Employee	Total Orders
Davolio, Nancy	1342.2000
Leverling, Janet	960.0000
Peacock, Margaret	1208.0000
Suyama, Michael	1086.0000
Total	4596.2000

Product Information

Product	Total Orders
Aniseed Syrup	60.0000
Chartreuse verte	378.0000
Escargots de Bourgogne	530.0000
Flotemysost	430.0000
Grandma's Boysenberry Spread	400.0000
Lakkalikööri	270.0000
Original Frankfurter grüne Soáe	26.0000
Raclette Courdavault	825.0000
Rössle Sauerkraut	775.2000
Spegesild	24.0000
Vegie-spread	878.0000
Total	4596.2000

Figure 5. The Customer Profile report looks like it has two detail bands, but it actually uses grouping. (CustomerProfile.FRX)

To prepare the data for this report, use two queries joined by UNION. The first query collects the employee information, while the second gathers the product information. Listing 13 shows the query. The additional field, cType, distinguishes employee records from product records in the result. This field is used to put the data in the right order and for grouping in the actual report.

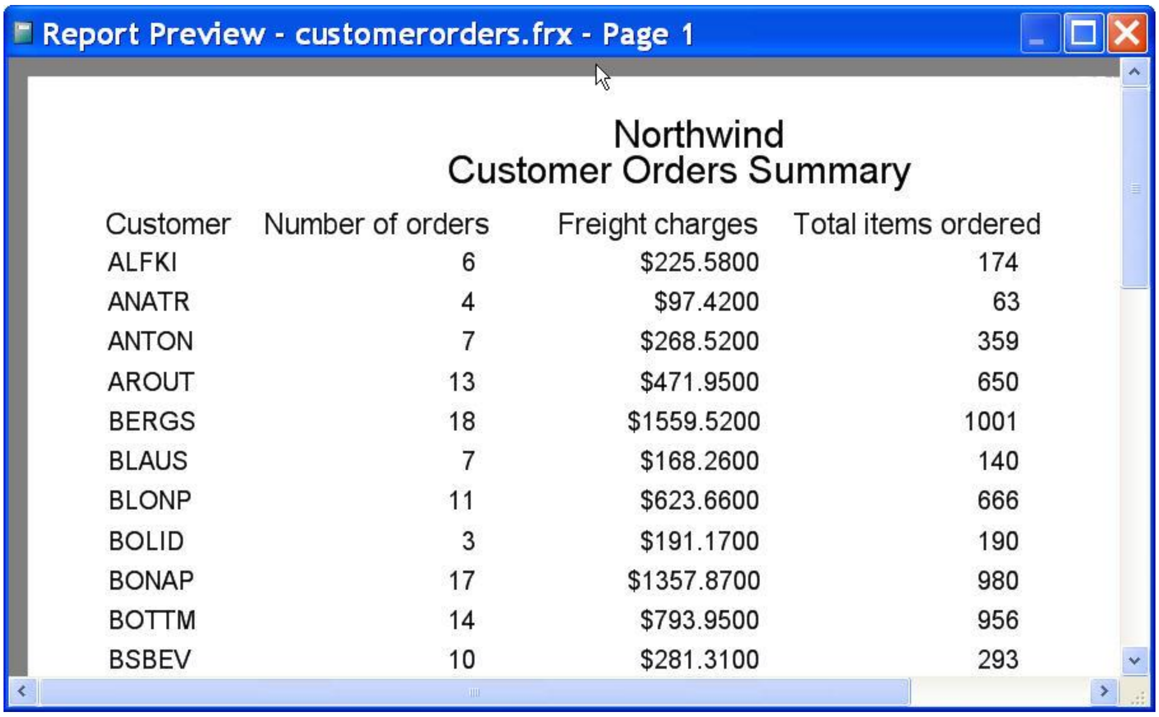
Listing 13. Use UNION to combine queries for the different children into a single result. (CustomerProfile.PRG)

```
SELECT Customers.CustomerID, CompanyName, ;
       PADR(TRIM(LastName) + ", " + FirstName, 40) AS cName, ;
       SUM(Quantity * UnitPrice) AS nTotal, ;
       "E" AS cType ;
    FROM Customers ;
      JOIN Orders ;
        JOIN OrderDetails ;
          ON Orders.OrderID = OrderDetails.OrderID ;
        ON Customers.CustomerID = Orders.CustomerID ;
      JOIN Employees ;
        ON Orders.EmployeeID = Employees.EmployeeID ;
    GROUP BY 1, 2, 3, 5 ;
UNION ALL ;
SELECT Customers.CustomerID, Customers.CompanyName, ;
       Products.ProductName AS cName, ;
       SUM(Quantity * OrderDetails.UnitPrice) AS nTotal, ;
       "P" AS cType ;
    FROM Customers ;
      JOIN Orders ;
        JOIN OrderDetails ;
          JOIN Products ;
            ON OrderDetails.ProductID = Products.ProductID ;
          ON Orders.OrderID = OrderDetails.OrderID ;
        ON Customers.CustomerID = Orders.CustomerID ;
    GROUP BY 1, 2, 3, 5 ;
  ORDER BY CompanyName, cType, cName ;
  INTO CURSOR CustomerProf
```

Note that the same approach can be used to collect detail and summary information into a cursor for reporting. This is useful in the situation where you want to show all the detail and then include a detailed summary (such as showing all orders for a year and then totals by month).

Aggregating both parents and children

SELECT's GROUP BY clause makes it easy to compute aggregate data, but when you want to aggregate both parent and child data, it can be tricky. Suppose you want to look at each customer in Northwind and figure out how many orders the customer has placed, the total shipping (freight) cost for those orders and the total number of items ordered, as in Figure 6. The first query in Listing 14 computes the number of orders and the shipping cost, while the second computes the total number of items ordered.



Northwind
Customer Orders Summary

Customer	Number of orders	Freight charges	Total items ordered
ALFKI	6	$225.5800	174
ANATR	4	$97.4200	63
ANTON	7	$268.5200	359
AROUT	13	$471.9500	650
BERGS	18	$1559.5200	1001
BLAUS	7	$168.2600	140
BLONP	11	$623.6600	666
BOLID	3	$191.1700	190
BONAP	17	$1357.8700	980
BOTTM	14	$793.9500	956
BSBEV	10	$281.3100	293

Figure 6. This summary report includes data aggregated from both the Orders and OrderDetails tables. (CustomerOrders.FRX)

Listing 14. Aggregating data from a single table is easy. (AggregateParentAndChild.PRG)

```
SELECT CustomerID, COUNT(OrderID) as OrderCount, ;
       SUM(Freight) As FreightTotal ;
   FROM Orders ;
   GROUP BY CustomerID ;
   INTO CURSOR CustomerData

SELECT CustomerID, SUM(Quantity) as QuantityOrdered ;
   FROM Orders ;
     JOIN OrderDetails ;
       ON Orders.OrderID = OrderDetails.OrderID ;
   GROUP BY CustomerID ;
   INTO CURSOR QuantityData
```

To compute all three values with a single query, you might try something like Listing 15. However, when you examine the results, you'll find that the number of orders and the freight charges are too high. To understand what happens in this case, you need to look at the way aggregation is performed. The engine first does the joins and the filtering, producing an intermediate result set. Then, it finds all the unique values for the combination of items in the GROUP BY list—all records with the same value are compressed into a single result record, computing aggregates as you go.

Listing 15. You can't aggregate parent and child data in a simple query. (AggregateBad.PRG)

```
SELECT CustomerID, COUNT(Orders.OrderID) as OrderCount, ;
       SUM(Freight) As FreightTotal, ;
       SUM(Quantity) as QuantityOrdered ;
   FROM Orders ;
     JOIN OrderDetails ;
       ON Orders.OrderID = OrderDetails.OrderID ;
   GROUP BY CustomerID ;
   INTO CURSOR AggregateDataBad
```

In the queries in Listing 14, that works perfectly. In the first query, the intermediate result contains one record per order. When you aggregate that data, each customer's orders are combined; since each was listed only once, you get the number of records and the total of the shipping charges for those orders. In the second query, the intermediate result contains one record for each line item of each order. When you aggregate, each customer's line items are combined, and the quantity is totaled.

In Listing 15, though, things are different. The intermediate result contains one record for each line item of each order. When you aggregate, the quantity is totaled correctly, but the count and the freight fields take in all that repeated data.

If you didn't need the shipping total, you could use DISTINCT to ensure that the count was correct, as in Listing 16. But there are two issues with using this approach to solve the stated problem. First, you can use DISTINCT only once in a query; since you'd need it for both order count and shipping total, you're out of luck. But even if you could omit the order count, DISTINCT won't work correctly for the shipping total. While it would eliminate the duplicates introduced by joining the OrderDetails table, it will also eliminate any duplicates that happen to occur in the data. That is, if two orders have the same freight cost, using SUM(DISTINCT Freight) would include that amount only once.

Listing 16. When you only need one parent field aggregated, you may be able to use DISTINCT. (AggregateCountAndQuantity.PRG)

```
SELECT CustomerID, COUNT(distinct Orders.OrderID) as OrderCount, ;
       SUM(Quantity) as QuantityOrdered ;
   FROM Orders ;
     JOIN OrderDetails ;
       ON Orders.OrderID = OrderDetails.OrderID ;
   GROUP BY CustomerID ;
   INTO CURSOR AggregateCountAndQuantity
```

In VFP 8 and earlier, the only solution is to run the two queries in Listing 14 and follow them with another query to combine the results, as shown in Listing 17.

Listing 17. Use a third query to combine the parent aggregates with the child aggregates. (AggregateParentAndChild.PRG)

```
SELECT CustomerData.CustomerID, OrderCount, FreightTotal, ;
       QuantityOrdered ;
   FROM CustomerData ;
     JOIN QuantityData ;
       ON CustomerData.CustomerID = QuantityData.CustomerID ;
   INTO CURSOR AggregatedData
```

VFP 9 offers some additional choices because you can handle the need to join the parent and child outside the main query. The most obvious is to make the two queries from Listing 14 into derived tables (tables created on the fly in the FROM clause); that approach is shown in Listing 18.

Listing 18. Rather than running three queries in a row, you can make the first two into derived tables. (AggregateAllDerived.PRG)

```
SELECT CustomerData.CustomerID, OrderCount, FreightTotal, ;
       QuantityOrdered ;
   FROM ( ;
     SELECT CustomerID, COUNT(OrderID) as OrderCount, ;
             SUM(Freight) As FreightTotal ;
       FROM Orders ;
        GROUP BY CustomerID) CustomerData ;
     JOIN ;
       (SELECT CustomerID, SUM(Quantity) as QuantityOrdered ;
          FROM Orders ;
            JOIN OrderDetails ;
              ON Orders.OrderID = OrderDetails.OrderID ;
          GROUP BY CustomerID) QuantityData ;
       ON CustomerData.CustomerID = QuantityData.CustomerID ;
   INTO CURSOR AggregatedData
```

Another solution uses a derived table only for the child aggregates; Listing 19 shows this technique. The downside of this approach is the need to wrap the field drawn from that query in MAX() (or put it into the GROUP BY clause).

Listing 19. You can use a derived table to compute the child aggregates and then join that result to the parent aggregates. (AggregateDerivedTable.PRG)

```
SELECT Orders.CustomerID, COUNT(OrderID) as OrderCount, ;
       SUM(Freight) as FreightTotal, ;
       MAX(QuantityOrdered) ;
   FROM Orders ;
```

```
      JOIN (SELECT CustomerID, SUM(Quantity) as QuantityOrdered ;
            FROM Orders ;
              JOIN OrderDetails ;
                ON Orders.OrderID = OrderDetails.OrderID ;
              GROUP BY CustomerID) QuantityData ;
        ON Orders.CustomerID = QuantityData.CustomerID ;
    GROUP BY Orders.CustomerID ;
    INTO CURSOR AggregatedData
```

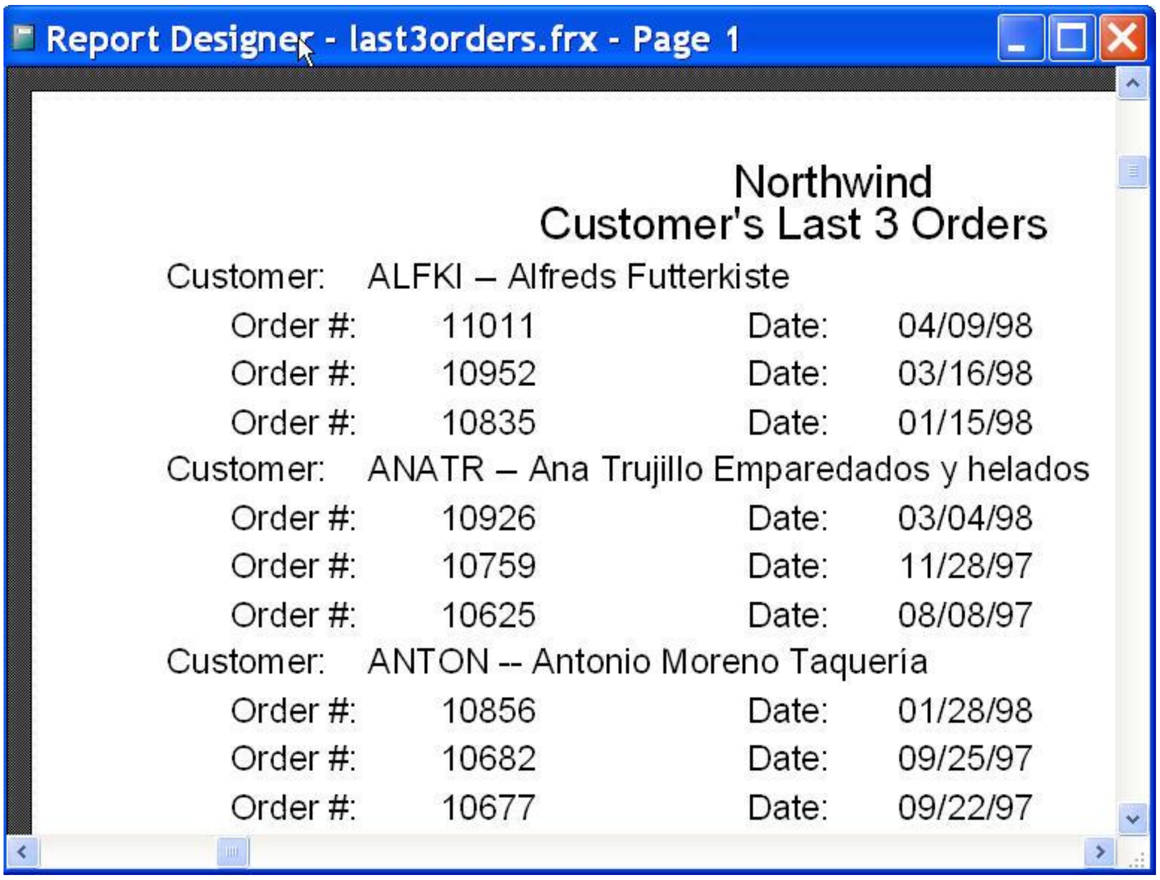
Finally, another possibility in VFP 9 is to compute the child aggregate in the field list and avoid the whole issue of a join in the main query. That solution is shown in Listing 20.

Listing 20. By computing the child aggregate in the field list, you avoid the whole issue of doing a join in the main query. (AggregateFieldList.PRG)

```
SELECT Orders.CustomerID, COUNT(OrderID) as OrderCount, ;
       SUM(Freight) as FreightTotal, ;
       (SELECT SUM(Quantity) ;
         FROM Orders Ord;
           JOIN OrderDetails ;
             ON Ord.OrderID = OrderDetails.OrderID ;
         WHERE Ord.CustomerID = Orders.CustomerID) as QuantityOrdered ;
   FROM Orders ;
   GROUP BY Orders.CustomerID ;
   INTO CURSOR AggregatedData
```

Finding the top N in a group

SELECT’s TOP n clause isn’t as useful as it may initially seem. It applies to the query results as a whole, returning the top n records based on the ordering criteria. In practice, though, we often want the top n from each of a series of groups. For example, we might want to find the three most recent orders for each Northwind customer, as Figure 7.



Northwind
Customer's Last 3 Orders

Customer: ALFKI – Alfreds Futterkiste

Order #:	11011	Date:	04/09/98
Order #:	10952	Date:	03/16/98
Order #:	10835	Date:	01/15/98

Customer: ANATR – Ana Trujillo Emparedados y helados

Order #:	10926	Date:	03/04/98
Order #:	10759	Date:	11/28/97
Order #:	10625	Date:	08/08/97

Customer: ANTON -- Antonio Moreno Taqueria

Order #:	10856	Date:	01/28/98
Order #:	10682	Date:	09/25/97
Order #:	10677	Date:	09/22/97

Figure 7. SELECT's TOP n clause doesn't help collect data for a report like this, showing the last 3 orders for each customer.

Your first instinct may be to use a subquery involving TOP n. However, you can't use TOP n in a correlated subquery and without correlating, there's no way to limit the subquery to the records for a particular customer.

The solution (which I learned from MVP Fabio Lunardon) is counter-intuitive. It involves joining the table to itself using an unorthodox join condition, then grouping the results and keeping only a subset of the records. Listing 21 shows the solution. The key code is in the join condition; it joins the Orders table to itself using the CustomerID, OrderID and OrderDate fields. After this join, you have one or more records for each customer-order combination; the number of records for each combination is the number of orders for that customer placed on that order's date or earlier. The GROUP BY clause compresses these groups into a single record and then keeps only those that started out as groups of 3 or fewer. In other words, it keeps only the three most recent orders for each customer.

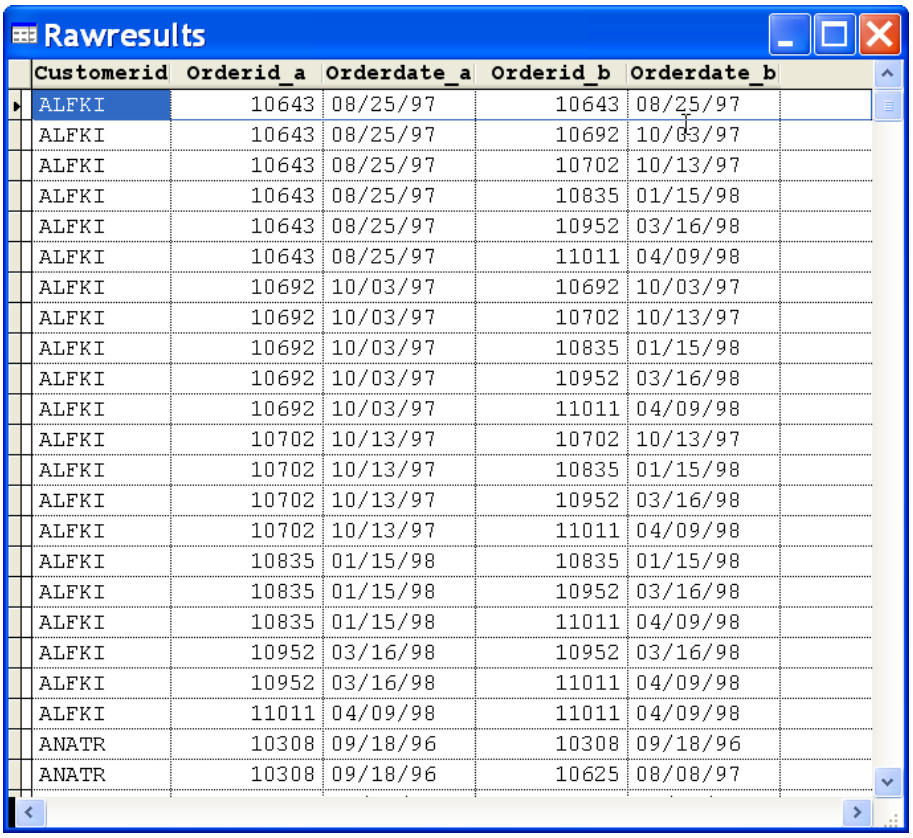
Listing 21. To get the top n for each group, join the table to itself comparing the field you want to order on. (TopNGroupJoin.PRG)

```
SELECT O1.CustomerID, O1.OrderID, O1.OrderDate ;
   FROM Orders O1;
     JOIN Orders O2 ;
       ON O1.CustomerID = O2.CustomerID ;
          AND (O1.OrderDate < O2.OrderDate ;
           OR (O1.OrderDate = O2.OrderDate ;
           AND O1.OrderID < O2.OrderID)) ;
   GROUP BY 1, 2, 3 ;
   HAVING COUNT(*) <= 3 ;
   INTO CURSOR TopItems
```

To get a better feel for how this process works, consider the query in Listing 22. It uses the same join conditions as the solution, but includes some additional fields and orders rather than groups the results. Figure 8 shows part of the result. Note that the oldest order (10643) for ALFKI has six records, while the newest order (11011) has only one. The HAVING clause in Listing 21 keeps only those with three or fewer records in this result.

Listing 22. Browsing the results of this query offers a better understanding of how the query in Listing 21 works. (TopNRawResults.PRG)

```
SELECT O1.CustomerID, O1.OrderID, O1.OrderDate, ;
       O2.OrderID, O2.OrderDate ;
   FROM Orders O1;
     JOIN Orders O2 ;
       ON O1.CustomerID = O2.CustomerID ;
          AND (O1.OrderDate <= O2.OrderDate ;
           OR (O1.OrderDate = O2.OrderDate ;
           AND O1.OrderID < O2.OrderID)) ;
   ORDER BY 1, 2, 4 ;
   INTO CURSOR RawResults
```



Customerid	Orderid_a	Orderdate_a	Orderid_b	Orderdate_b
ALFKI	10643	08/25/97	10643	08/25/97
ALFKI	10643	08/25/97	10692	10/03/97
ALFKI	10643	08/25/97	10702	10/13/97
ALFKI	10643	08/25/97	10835	01/15/98
ALFKI	10643	08/25/97	10952	03/16/98
ALFKI	10643	08/25/97	11011	04/09/98
ALFKI	10692	10/03/97	10692	10/03/97
ALFKI	10692	10/03/97	10702	10/13/97
ALFKI	10692	10/03/97	10835	01/15/98
ALFKI	10692	10/03/97	10952	03/16/98
ALFKI	10692	10/03/97	11011	04/09/98
ALFKI	10702	10/13/97	10702	10/13/97
ALFKI	10702	10/13/97	10835	01/15/98
ALFKI	10702	10/13/97	10952	03/16/98
ALFKI	10702	10/13/97	11011	04/09/98
ALFKI	10835	01/15/98	10835	01/15/98
ALFKI	10835	01/15/98	10952	03/16/98
ALFKI	10835	01/15/98	11011	04/09/98
ALFKI	10952	03/16/98	10952	03/16/98
ALFKI	10952	03/16/98	11011	04/09/98
ALFKI	11011	04/09/98	11011	04/09/98
ANATR	10308	09/18/96	10308	09/18/96
ANATR	10308	09/18/96	10625	08/08/97

Figure 8. The unusual join shown in Listing 21 and Listing 22 results in a different number of records for each order for a given customer.

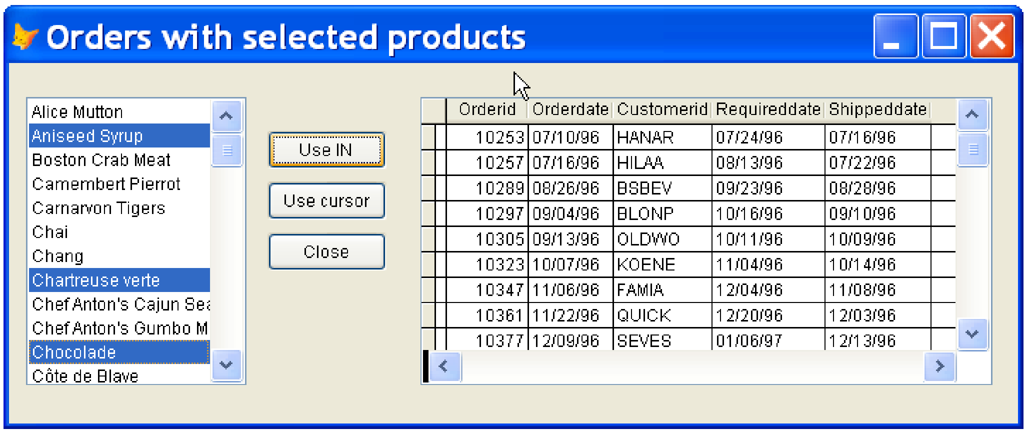
Processing data

While most people think of SQL first for reporting, all of the data manipulation commands (SELECT, INSERT, DELETE and UPDATE) can be quite useful for other parts of an application as well. This section looks at some of the ways you can use SQL in processing data.

Matching user input

A very common need in applications is finding records that match user input. When the user provides a single value, it's easy to write a query that finds all matching records. But when the user can specify multiple values and you want to find records that match any of them, things get a little trickier.

The form shown in Figure 9 demonstrates two solutions. In the form, the user can choose one or more products. After clicking either Use IN or Use cursor, the grid is filled with orders that contain any of those items.



Alice Mutton
Aniseed Syrup
Boston Crab Meat
Camembert Pierrot
Carnarvon Tigers
Chai
Chang
Chartreuse verte
Chef Anton's Cajun Sea
Chef Anton's Gumbo M
Chocolade
Côte de Blave

Orderid	Orderdate	Customerid	Requireddate	Shippeddate
10253	07/10/96	HANAR	07/24/96	07/16/96
10257	07/16/96	HILAA	08/13/96	07/22/96
10289	08/26/96	BSBEV	09/23/96	08/28/96
10297	09/04/96	BLONP	10/16/96	09/10/96
10305	09/13/96	OLDWO	10/11/96	10/09/96
10323	10/07/96	KOENE	11/04/96	10/14/96
10347	11/06/96	FAMIA	12/04/96	11/08/96
10361	11/22/96	QUICK	12/20/96	12/03/96
10377	12/09/96	SEVES	01/06/97	12/13/96

Figure 9. When the user chooses one or more items, then clicks one of the Use buttons, the grid fills with orders that include the chosen items. (ChooseItems.SCX)

One way to handle the user's choices is to build a list of values and then use the IN operator to filter results. Listing 23 shows the code in the Use IN button's Click method.

Listing 23. VFP's string handling abilities make it easy to build a string of values to use with the IN operator.

```
LOCAL cListValues, nIndex

* Build a list of items
cListValues = ""

WITH ThisForm.lstProducts as ListBox
   FOR nIndex = 1 TO .ListCount
      IF .Selected[ nIndex ]
         cListValues = cListValues + "," + .List[ nIndex, 2 ]
      ENDIF
   ENDFOR
ENDWITH

cListValues = SUBSTR(cListValues, 2)

* Run the query
IF NOT EMPTY(cListValues)
   SELECT OrderID, OrderDate, CustomerID, RequiredDate, ShippedDate ;
      FROM Orders ;
      WHERE OrderID IN ;
         (SELECT OrderID ;
            FROM OrderDetails ;
              WHERE ProductID IN (&cListValues)) ;
      ORDER BY OrderID ;
      INTO CURSOR ProductsOrdered
ENDIF

WITH ThisForm.grdOrders
   .RecordSource = "ProductsOrdered"
   .AutoFit()
ENDWITH
```

This approach has one significant limit. In VFP 8 and earlier, the IN operator can handle no more than 24 items. In VFP 9, while the limit has gone up and you can control it (with SYS(3055)), it's still theoretically possible to hit the limit.

A better solution is to build a cursor of the values and then use that cursor in the query. The code in the Use cursor button's Click method is shown in Listing 24. This approach is limited only by memory and disk space.

Listing 24. Cursors and tables are VFP's bread and butter. Putting the user's choices into a cursor offers a scalable solution.

```
LOCAL cListValues, nIndex

* Build a list of items
CREATE CURSOR ProductsChosen (ProductID I)

WITH ThisForm.lstProducts as ListBox
   FOR nIndex = 1 TO .ListCount
      IF .Selected[ nIndex ]
         INSERT INTO ProductsChosen VALUES ( VAL(.List[ nIndex, 2] ))
      ENDIF
   ENDFOR
ENDWITH

* Run the query
IF RECCOUNT("ProductsChosen") > 0
   SELECT OrderID, OrderDate, CustomerID, RequiredDate, ShippedDate ;
      FROM Orders ;
      WHERE OrderID IN ;
         (SELECT OrderID ;
            FROM OrderDetails ;
              JOIN ProductsChosen ;
                ON OrderDetails.ProductID = ProductsChosen.ProductID );
      ORDER BY OrderID ;
      INTO CURSOR ProductsOrdered
ENDIF

WITH ThisForm.grdOrders
   .RecordSource = "ProductsOrdered"
   .AutoFit()
ENDWITH
```

In both examples, the subquery lets you list each order only once in the results even if several of the chosen products appear in a particular order.

VFP actually offers a third way to filter based on user entries. You can store the items in an array and use ASCAN() to search within the array. However, this approach can get sticky because queries do string comparisons using the SET ANSI setting, but ASCAN() uses the SET EXACT setting.

Managing field type and size

When you use a query to create a cursor or table intended for further processing, you sometimes need to include additional fields not in the original tables. At other times, you want a field in the result to be of a different type or size than the field it's based on. The method for doing these things varies by VFP version and field type.

The first question is how VFP determines the type and size of fields in a query result. Not surprisingly, fields listed in the field list without any manipulation are turned into identically typed and sized fields in the result.

Where it gets interesting is when the field list contains expressions involving fields. The rules vary depending on the type of the fields used.

For character and memo fields, most expressions result in character fields. (The exceptions are those that actively convert characters to something else.) If the expression results in a fixed size, such as LEFT(MyField,3), the new field has that size. If the expression results in a variable length result and involves a single character field, the new field has the same size as the original field. However, if the original field is a memo, the new field size is based on the expression result from the first record in the table. When character expressions involve multiple fields, basically the same rules apply. If the expression result has a fixed length, the new field has that length. Otherwise, if the original fields are character, the field size is the total size of the fields involved; if there are any memo fields involved, the field size is determined by the first record in the table.

For numeric fields, the rules for expressions are more complicated, but they basically boil down to the new field being large enough to accommodate the largest possible value the expression can generate. In my testing, I found that occasionally, the new field is actually a little larger than that.

When you add currency, double and integer fields into the mix, it gets a little more complex. For example, a field created by adding two integer fields results in a 12-digit numeric field and the sum of two double fields is stored in a numeric field with 20 total positions and 2 decimals. In fact, in general, when the engine wants to allow maximum space for a numeric result, it uses an N(20,2) field. Some combinations, though, do use the other numeric types. For example, any combination of a currency field with another numeric type results in a currency field.

There’s one more wildcard. When you use an expression that can return different results for different records, such as calls to IIF() or ICASE(), VFP evaluates the expression for the first record and uses the resulting type and size.

Adding empty fields

To add empty fields to a query result, you need to specify a field of the appropriate type and size. Through VFP 8, you have several tools at your disposal for doing this. You can use the SPACE() function to create an empty string of a specified length. PADL(), PADR(), PADC() and TRANSFORM() also can create strings of various lengths. You can just list an empty string of the right length in the query. The Caller field in Listing 25 demonstrates using SPACE(), while the SpokeWith field shows a literal value.

Listing 25. Prior to VFP 9, creating empty fields with a query varied by field type. (EmptyFields8.PRG)

```
SELECT CustomerID, CompanyName, ;
```

```
       SPACE(30) AS Caller, ;
       .F. AS Reached, ;
       { : } AS CallTime, ;
       00 AS Rings, ;
       "                              " AS SpokeWith, ;
       $0 AS Pledge ;
   FROM Customers ;
   INTO CURSOR Calls READWRITE
```

To create numeric values of a specified length, use a series of 0's, as with the Rings field in Listing 25. For currency, use $0, as in the Pledge field.

For logical values, specify either .T. or .F.

To specify an empty date, use {}. For an empty datetime, you need to do something that indicates that the time should be included as well. At a minimum, you need one space and a colon inside the brackets; the CallTime field demonstrates.

There's no empty value you can specify for the other field types, such as memo and double. Instead, you need to create a one-record cursor containing fields of the appropriate type and join that in the query. This is one of the cases where you don't have to specify a join condition; if omitting it makes you uncomfortable (or makes writing other join conditions tricky), you can join ON .T., as in Listing 26.

Listing 26. To add an empty memo field to a query, create a cursor and join it to the other tables. (EmptyMemo8.PRG)

```
CREATE CURSOR Junk (Notes M)
INSERT INTO Junk VALUES ("")

SELECT CustomerID, CompanyName, ;
       SPACE(30) AS Caller, ;
       .F. AS Reached, ;
       { : } AS CallTime, ;
       00 AS Rings, ;
       "                              " AS SpokeWith, ;
       $0 AS Pledge, ;
       Notes ;
   FROM Customers ;
     JOIN Junk ;
       ON .T. ;
   INTO CURSOR Calls READWRITE
```

VFP 9 makes the whole thing much easier with the new CAST() function. The syntax is:

```
CAST(uValue AS cType [ (nWidth [, nDecimals] ) ] [ NULL | NOT NULL ] )
```

In other words, you provide a value (which can be an expression) and then specify the type for the value, including the field size and decimals as needed. You can also specify whether the field accepts nulls. Listing 27 shows the same example as Listing 26, but uses CAST() rather than the other techniques. As the example acknowledges, there's not much reason to use CAST() to create empty logical fields.

Listing 27. VFP 9's CAST() function makes creation of empty fields much easier. (EmptyMemo9.PRG)

```
SELECT CustomerID, CompanyName, ;
       CAST("" AS C(30)) AS Caller, ;
       .F. AS Reached, ;
       CAST({} AS T) AS CallTime, ;
       CAST(0 AS N(2)) AS Rings, ;
       CAST(" " AS C(30)) AS SpokeWith, ;
       CAST(0 AS Y) AS Pledge, ;
       CAST("" AS M) AS Notes ;
   FROM Customers ;
   INTO CURSOR Calls READWRITE
```

Changing field type or size

For fields originating in other tables, prior to VFP 9, you need tricks to overrule the engine's choice of type and size. For character fields, you can use the PADx() functions to force a particular size. When the string you pass to one of those functions is longer than the specified size, it's truncated. For numeric fields, you can force larger result fields, but not smaller. To specify the size of a numeric field, add a template of 0's to the result that indicate the size you want. For example, if MyNumber is defined as N(5,2), you can get a result field of N(7,3) using code like that in Listing 28. (Interestingly, in VFP 9, the query in Listing 28 results in a field of N(8,3).)

Listing 28. To force numeric fields to have a particular size, add a template of 0's. (ForceNumSize.PRG)

```
CREATE CURSOR Test (MyNumber N(5,2))
INSERT INTO Test VALUES (12.34)
INSERT INTO Test VALUES (7.68)
INSERT INTO Test VALUES (37)
INSERT INTO Test VALUES (12.77)

SELECT MyNumber + 00.000 AS MyNewNumber ;
   FROM Test ;
   INTO CURSOR TestResult
```

As for creating empty fields, in VFP 9, CAST() lets you tell VFP exactly what you want; you can even change types, as long as the data is compatible. With CAST(), you can make numeric fields smaller, though any value that doesn't fit into the new field turns into asterisks. Listing 29 demonstrates.

Listing 29. With CAST(), you can even shrink numeric fields. (CastNumSize.PRG)

```
CREATE CURSOR Test (MyNumber N(7,2))
INSERT INTO Test VALUES (12.34)
INSERT INTO Test VALUES (7.68)
INSERT INTO Test VALUES (37)
INSERT INTO Test VALUES (12.77)
INSERT INTO Test VALUES (1234.77)

SELECT MyNumber, ;
       CAST(MyNumber AS N(7,3)) AS MyNewNumber, ;
       CAST(MyNumber AS N(5)) AS MyIntNum, ;
       CAST(MyNumber AS B) AS MyDbl, ;
       CAST(MyNumber AS N(5,1)) AS MyOneDec, ;
       CAST(MyNumber AS N(3,2)) AS MySmallNum ;
   FROM Test ;
   INTO CURSOR TestResult
```

Using user-defined functions (UDFs) in queries

While you can do a great deal with expressions in the field list of a query, occasionally you need to perform an operation that simply can't be written as an expression. In those cases, it is possible to call your own function (UDF) to do the calculation. However, to get the desired results, you need to understand the way VFP handles UDFs in a query.

The first thing VFP does when executing a query is figure out the structure of the result. It examines each item in the field list to determine the type and size of the resulting field. When the item is a field, it's simple—the new field has the same type and size. When the item is an expression, VFP sometimes evaluates the expression using the first available record to see the type and size returned. (See the previous section for more details.)

When the item includes a call to a UDF, VFP executes the UDF to see what type and size it returns. Often, that's not a problem, except for the additional execution time involved. But for some functions, an extra call changes the results.

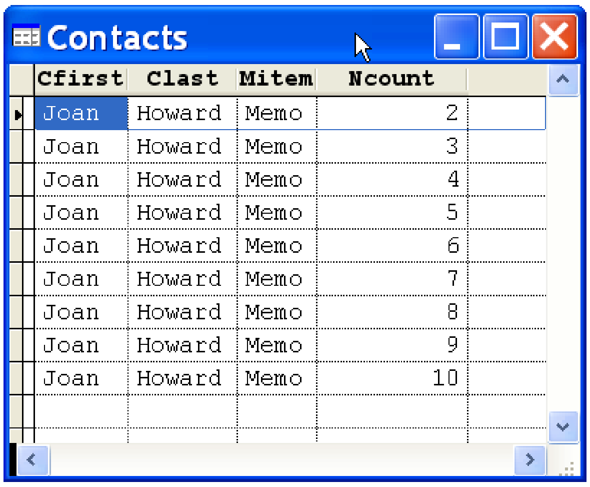
Consider the query in Listing 30—the goal here is to get all the contact items for one person and number them in order. There's no expression you can use in a query to assign numbers in sequence, so the query calls a function that accepts a parameter, increments it, and returns the new value. **Figure 10** shows the results; the first record has 2 for the counter field.

Listing 30. Using functions that count or produce output in a query can give surprising results. (UDFinQuery.PRG)

```
nItemCount = 0
SELECT cFirst, cLast, mItem, GetCount(@nItemCount) AS nCount ;
   FROM Person ;
     JOIN PersonToItem ;
       ON Person.iID = PersonToItem.iPersonFK ;
     JOIN ContactItem ;
```

```
        ON PersonToItem.iItemFK = ContactItem.iID ;
    WHERE Person.iID = 4000 ;
    INTO CURSOR Contacts

FUNCTION GetCount(nCounter)
nCounter = nCounter + 1
RETURN nCounter
```


Contacts

Cfirst	Clast	Mitem	Ncount
Joan	Howard	Memo	2
Joan	Howard	Memo	3
Joan	Howard	Memo	4
Joan	Howard	Memo	5
Joan	Howard	Memo	6
Joan	Howard	Memo	7
Joan	Howard	Memo	8
Joan	Howard	Memo	9
Joan	Howard	Memo	10

Figure 10. Queries that call UDFs don't always return the results you expect.

UDFs that produce output are also a problem in this regard; the extra call produces extra output. It appears that each use of the function in a query calls it one extra time. However, in a UNION with the function called in both queries, the two calls occur at different times; the initial checks for the second query in the UNION aren't done until after executing the first query.

The bottom line is that you have to expect extra calls to a function that's included in a query.

Including record numbers in results

SQL deals only with sets of records and has no notion of a record number. But there are times when the Xbase record number can be extremely useful. Including RECNO() in a query's field list can be tricky, though.

The problem is the way VFP handles work areas and aliases in queries. When a query executes, VFP opens each table, whether or not it's already open. If the table is not open, it uses its normal alias and VFP leaves it open after executing the query. However, if the table is already open, VFP opens it again with a different alias and closes it afterwards. So, you can't be sure what alias the query uses to address a particular table.

This means that you can't pass the optional alias parameter to RECNO(), which in turn means you can't use RECNO() in a multi-table query. (In fact, in a multi-table query, it's risky to pass the optional alias parameter to any VFP function that accepts it.) Prior to VFP 9, that meant using an extra query to add a record number field before joining a table with other tables; Listing 31 demonstrates. In VFP 9, you can solve the problem using a derived table, as in Listing 32.

Listing 31. In VFP 8 and earlier, it takes two queries to add record numbers to a multi-table query. (NumberedOrders.PRG)

```
SELECT RECNO() AS nRec, * ;
   FROM Orders ;
   INTO CURSOR OrdersWithRecNo

SELECT CompanyName, nRec, OrderDate, OrderID ;
   FROM Customers ;
     JOIN OrdersWithRecNo ;
       ON Customers.CustomerID = OrdersWithRecNo.CustomerID ;
   WHERE Customers.CompanyName = "M" ;
   INTO CURSOR NumberedOrders
```

Listing 32. In VFP 9, you can use a derived table to add record numbers in one step. (NumberedOrdersDerived.PRG)

```
SELECT CompanyName, nRec, OrderDate, OrderID ;
```

```
   FROM Customers ;
     JOIN (SELECT RECNO() AS nRec, * ;
            FROM Orders) OrdersWithRecNo ;
       ON Customers.CustomerID = OrdersWithRecNo.CustomerID ;
   WHERE Customers.CompanyName = "M" ;
   INTO CURSOR NumberedOrders
```

A similar technique can be used to add a counter after querying. In VFP 8 and earlier, run the query you want and follow it with a query like the first one in Listing 31. In VFP 9, the real query can be a derived table in a query that adds the record numbers, as in Listing 33.

Listing 33. You can add the record number after querying in order to get a sequential number field. (ProductList.PRG)

```
SELECT RECNO() AS nPosition, * ;
   FROM ( ;
      SELECT * ;
         FROM Products ;
         ORDER BY ProductName) ProductsAlpha ;
   INTO CURSOR ProductList
```

Cleaning up data

With its emphasis on results rather than procedures, SQL can be very handy when you need to look for and deal with problems in data. This section explores some of the common situations.

The Northwind database doesn't contain the kind of problems we’re looking for. To demonstrate them, I copied some of the data from two Northwind tables. My copy of Customers is called Cust and contains only the first thirty alphabetically; it also contains duplicate records with slightly different names and customer IDs for several of the customers. Ord contains the first 200 records from the Northwind Orders table. Because Cust contains only a subset of the Northwind customers, Ord has a number of orphaned records. Both tables are included in the materials for this session.

Dealing with duplicates

One very common question is how to find all duplicated records in a table. The first issue is deciding what constitutes a duplicate record. This is a business question, not a programming question and the answer depends on the situation. In a customer table, it might be matching addresses, while in a table of people, it might require an exact match of first and last names, plus social security number.

Once you know what constitutes a duplicate, identifying duplicate values can be easy. Listing 34 shows a query that finds the duplicated information in Cust, based on a rule that an exact address match constitutes a duplicate. However, this approach doesn’t tell you what specific records are duplicated, just which identifying values they contain.

Listing 34. You can identify duplicate values by grouping on the fields that determine a duplicate and keeping only those where the count is more than one. (IdentifyDupValues.PRG)

```
SELECT Address, City, Region, Country, COUNT(*) AS nCount ;
   FROM Cust ;
   GROUP BY Address, City, Region, Country ;
   HAVING nCount > 1 ;
   INTO CURSOR Dups
```

To find the actual records, you need to use the results of the previous query and grab all the records that match. The query in **Listing 35** does the trick.

Listing 35. To find the actual duplicate records, use the query from Listing 34 as a derived table. (FindDups.PRG)

```
SELECT * ;
FROM Cust ;
  JOIN ( ;
   SELECT Address, City, Region, Country, COUNT(*) AS nCount ;
      FROM Cust ;
      GROUP BY Address, City, Region, Country ;
      HAVING nCount > 1) Dups ;
   ON ((Cust.Address = Dups.Address) OR ;
      (Cust.Address IS NULL AND Dups.Address IS NULL)) ;
   AND ((Cust.City = Dups.City) OR ;
       (Cust.City IS NULL AND Dups.City IS NULL)) ;
   AND ((Cust.Region = Dups.Region) OR ;
       (Cust.Region IS NULL AND Dups.Region IS NULL)) ;
   AND ((Cust.Country = Dups.Country) OR ;
       (Cust.Country IS NULL AND Dups.Country IS NULL)) ;
   INTO CURSOR HasDups
```

The need to check each of the comparison fields for null make the join clause hard to read. (This is an issue only if the original table allows nulls in the fields of interest.) An alternative is to get rid of the nulls as you go, replacing them with the empty string. (This approach works only in situations where null and empty values can be considered the same.) Listing 36 shows a version that uses NVL() to turn nulls into empty strings.

Listing 36. Using NVL() lets you get rid of all the IS NULL tests. (FindDupsNVL.PRG)

```
SELECT * ;
FROM Cust ;
  JOIN ( ;
   SELECT NVL(Address, "") AS Address, NVL(City, "") AS City, ;
          NVL(Region, "") AS Region, NVL(Country, "") AS Country, ;
          COUNT(*) AS nCount ;
      FROM Cust ;
      GROUP BY Address, City, Region, Country ;
      HAVING nCount > 1) Dups ;
   ON NVL(Cust.Address, "") = Dups.Address ;
   AND NVL(Cust.City, "") = Dups.City ;
   AND NVL(Cust.Region, "") = Dups.Region ;
   AND NVL(Cust.Country, "") = Dups.Country ;
   INTO CURSOR HasDups
```

Removing duplicate records is trickier than finding them. Normally, the right approach is to show a user the duplicates and let him decide what to discard and what to keep.

In addition, identifying duplicates isn't always as easy as finding exact matches. More often, duplicates vary in subtle ways, such as one record having "Street" spelled out while another uses the abbreviation "St." While it's not a SQL problem (so not discussed here), figuring out how to match data in different records can be a major issue in the de-duplication process.

Handling orphaned records

Orphaned records are child records with no parent; in a properly functioning database application, there should be no orphans. But in the real world, records do get orphaned.

Finding orphans is the same as finding unmatched records generally. You use a subquery to find all the values in the parent table, and then keep only those without a match. Listing 37 shows a query that identifies the orphaned records in Ord.

Listing 37. Finding orphaned records calls for a subquery. (FindOrphans.PRG)

```
SELECT * ;
   FROM Ord ;
   WHERE CustomerID NOT IN ;
      (SELECT CustomerID FROM Cust) ;
   INTO CURSOR Orphans
```

Removing the orphaned records is no harder than finding them; Listing 38 demonstrates. (In some situations, however, rather than removing orphaned records, the right solution is to re-assign them to a new parent.)

Listing 38. Removing orphaned records uses the same subquery as finding them. (DeleteOrphans.PRG)

```
DELETE ;
   FROM Ord ;
   WHERE CustomerID NOT IN ;
      (SELECT CustomerID FROM Cust)
```

Looking for missing values

When you're working with surrogate keys, it doesn't matter whether there are gaps in the sequence of key values. But there are situations where you need to keep track of a set of sequential numbers. For example, sometimes you have pre-printed forms with id numbers, and every form must be tracked.

In those situations, you may need a way to get a list of missing values, those in the sequence that aren't listed in the relevant table. To do so, you need to know the possible range of values.

The trick is to create a cursor containing all the values of interest. Then, you can apply the technique used in the preceding section to find all the records in the cursor that have no match in the table of interest.

To demonstrate, we'll use the Items table described earlier in this document. The iID field is missing some values. (In this case, we probably don't care, since iID is an auto-generated surrogate key.) Listing 39 produces a list of missing values; to use this code, call it, passing the low and high values you want to check.

Listing 39. Getting a list of missing values is much like looking for orphans. (MissingValues.PRG)

```
LPARAMETERS nMin, nMax

LOCAL nVal

CREATE CURSOR AllVals (iVal I)
FOR nVal = nMin TO nMax
   INSERT INTO AllVals VALUES (nVal)
ENDFOR

SELECT iVal ;
   FROM AllVals ;
   WHERE iVal NOT IN (;
      SELECT iID FROM Items) ;
   INTO CURSOR MissingValues
```

Managing SQL commands and settings

While SQL commands are often easier to write than their Xbase counterparts, they do bring their own set of management issues, things like tables left open, effects of SET commands and so forth. This section addresses some of those issues.

Handling tables left open

VFP's SQL commands open any tables they need to work with and leave them open. In some situations, you may want to keep tight control over what's open and what's not. It's not too hard to clean up after the SQL commands. (The program ShowWorkAreas.PRG in the session materials demonstrates that SQL commands leave tables open and that SELECT re-opens of open tables.)

The SQL commands use essentially the same strategy as many developers to find work areas. They perform the equivalent of USE table IN 0, which opens the table in the lowest available work area. To close tables opened by a SQL command, make a list of open tables before the command and another afterward. Then compare the lists, closing all those that weren't open beforehand. There's one exception, of course; if the command was SELECT, you don't want to close the cursor or table created by the command.

Because the technique requires code both before and after the command, I created a class that can be used to wrap SQL commands. The version in Listing 40 is a barebones approach with no error-handling and addressing only the question of closing tables opened by the command, but it might form the basis for a generic SQL wrapper class. To use it, instantiate the object and call the RunSQL method, passing the SQL command as a parameter.

Listing 40. To clean up after SQL commands, you need to get a list of open tables before the command and compare it to a list of open tables after the command. (ManageSQL.PRG)

```
DEFINE CLASS ManageSQL AS Custom

cCommand = ""
lCloseResult = .F.
```

```
DIMENSION aBefore[1], aAfter[1]
cResultAlias = ""

PROCEDURE RunSQL(cCommand, lCloseResult)

IF EMPTY(m.cCommand)
   IF EMPTY(This.cCommand)
      RETURN .F.
   ENDIF
ELSE
   This.cCommand = m.cCommand
ENDIF

IF PCOUNT()>1
   This.lCloseResult = m.lCloseResult
ENDIF

This.ListOpenTables("This.aBefore")
&cCommand
This.cResultAlias = ALIAS()
This.ListOpenTables("This.aAfter")

This.CleanUp()

RETURN
ENDPROC

PROCEDURE ListOpenTables(cArrayName)

LOCAL nUsedCount

nUsedCount = AUSED(&cArrayName)

RETURN nUsedCount
ENDPROC

PROCEDURE CleanUp
* Compare the lists and close the tables opened

LOCAL nAfterCount, nTable, cTableAlias

nAfterCount = ALEN(This.aAfter, 1)

FOR nTable = 1 TO nAfterCount
   cTableAlias = This.aAfter[ nTable, 1]
   IF ASCAN(This.aBefore, cTableAlias , -1, -1, 1) = 0
      * Not found--opened by command
      IF This.lCloseResult OR (This.cResultAlias <> cTableAlias )
         USE IN (cTableAlias)
      ENDIF
   ENDIF
ENDFOR

RETURN
ENDPROC

ENDDEFINE
```

The key methods here are ListOpenTables and CleanUp. ListOpenTables calls AUSED() to store a list of work areas in use and their aliases. CleanUp goes through the list of open tables afterwards. For each, it searches in the before list; if the table is not found, it's deleted unless it's the result table and the lCloseResult property is False.

Do keep in mind that opening tables is one of the slowest things VFP can do. If you're going to use the same table repeatedly, it's better to leave it open until you're done. In addition, if SQL code is running in a private data session, the tables opened will be closed when the data session is destroyed.

SQL and the SET commands

One of the biggest challenges of working in VFP is the large collection of SET commands that can change the behavior of code. SQL commands are affected by a number of the SET commands. Some, like SET DATE, have the same effect on SQL commands as they do on Xbase commands. But a few are specific to SQL commands.

VFP has an unusual way of comparing strings. When SET EXACT is OFF, strings are compared only to the end of the right-hand string. That means that, in Xbase code, the first comparison here is True, while the second is False:

```
"Smith" = "S"
"S" = "Smith"
```

This is convenient for doing look-ups and so forth, though it's also one of the traps VFP developers fall into regularly.

SQL commands are not affected by SET EXACT. However, the SET ANSI command has a similar impact on them. SET ANSI OFF allows partial string matching in SQL commands.

SET ANSI is a little different than SET EXACT. It compares to the end of the shorter string, no matter which side of the operator it's on. So the two commands in Listing 41 produce identical results.

Listing 41. With SET ANSI OFF, strings are compared to the end of the shorter value, no matter which side of the operator it's on. (CompareANSIOff.PRG)

```
SELECT CompanyName ;
   FROM Customers ;
   WHERE Country = "A" ;
   INTO CURSOR FromACountries1

SELECT CompanyName ;
   FROM Customers ;
   WHERE "A" = Country ;
   INTO CURSOR FromACountries2
```

SET ANSI ON for exact string matching. Be aware, though, that SQL ignores trailing blanks, even with ANSI ON; this applies even to the "exactly equals" operator (==). Also, keep in mind that the "not equals" operator is affected by SET ANSI, as well, whether you use "<>" or "#".

Two other SET commands have a direct effect on SQL behavior. SET ENGINEBEHAVIOR determines whether certain behavior changes introduced in VFP 8 and VFP 9 are in effect; see Help for the complete list.

SET SQLBUFFERING specifies the default setting for the WITH (Buffering = lExpr) clause introduced in VFP 9.

Summing Up

VFP's SQL commands offer quick access to data and the opportunity to look at problems in terms of the desired results, rather than the way to find the data. Changes in recent versions, especially VFP 9, make SQL more powerful than ever. Learn to use it effectively to make yourself more productive.

These notes are based on Chapter 8 of *Taming Visual FoxPro's SQL*, by Tamar E. Granor, published by Hentzenwerke Publishing.