

Chapter 5

Working with Subqueries

Performing a query within another SQL command increases the abilities of SQL considerably. This chapter looks at the rules for forming such queries and the places you use them.

A *subquery* is a query that appears within another SQL command. Three of VFP's SQL DML commands (SELECT, DELETE, and UPDATE) support subqueries, though the rules and reasons for using them vary. VFP 9 increased the capabilities of subqueries and the ways they can be used.

Some subqueries stand alone; you can run the subquery independent of the command that contains it. Other subqueries rely on fields from the containing command—these subqueries are said to be *correlated*. See “Correlation” later in this chapter.

A subquery is a complete query, but cannot contain the TO or INTO clause. Subqueries are enclosed in parentheses in the containing query. Subqueries can appear in the WHERE clause of SELECT, UPDATE, and DELETE. Beginning in VFP 9, subqueries can also be used in the field list of SELECT, in the SET clause of UPDATE, and in the FROM clause of SELECT, UPDATE, and DELETE.

In VFP 8 and earlier, a subquery cannot contain another subquery. Beginning in VFP 9, subqueries can be nested.

VFP 8 and earlier imposed other restrictions on subqueries as well. In those versions, a correlated subquery cannot contain a GROUP BY clause. VFP 9 permits grouping and correlation in the same subquery. VFP 8 and earlier also prohibit the use of TOP N in subqueries; VFP 9 lifts that restriction, as well.

Filtering with subqueries

The most common use for subqueries is filtering data in the WHERE clause of a SQL command. Four special operators (shown in **Table 1**), as well as the conventional operators like = and >, are used to connect the containing command and the subquery. The IN operator is probably the most frequently used; it says to include in the result set of the containing command all those records with a matching value in the subquery results. IN is often combined with NOT to operate on all records that are not in the subquery results.

Table 1. These operators let you compare data to results of subqueries in the WHERE clause of a SQL command.

Operator	Meaning
IN	Operates on any records in the containing command where the specified expression appears in the subquery results.
EXISTS	Operates on any records in the containing command where the subquery result has at least one row.
ALL	Used in conjunction with one of the comparison operators (=, <>, <, <=, >, >=). Operates on any records in the containing command where the specified expression has the specified relationship to every record in the subquery result.
ANY, SOME	Used in conjunction with one of the comparison operators (=, <>, <, <=, >, >=). Operates on any records in the containing command where the specified expression has the specified relationship to at least one record in the subquery result.

One well-known use for a subquery is to find all the records in one list not in another list. For example, the query in **Listing 1** retrieves a list of all customers who didn't place an order in 1996. The subquery builds a list of customers who did order in that year.

Listing 1. The subquery here retrieves a list of all customers who placed orders in the specified year. The main query uses that list to find the reverse—those who didn't order in that year.

```
SELECT CompanyName ;
FROM Customers ;
WHERE CustomerID NOT IN ;
  (SELECT CustomerID ;
   FROM Orders ;
   WHERE YEAR(OrderDate) = 1996) ;
INTO CURSOR NoOrders
```



The query in Listing 1 is included in the Developer Downloads for this chapter, available from www.hentzenwerke.com, as NoOrders.PRG.

With UPDATE and DELETE, IN lets you act on all the records selected by a subquery. For example, the code in **Listing 2** increases prices by 10% for products that come from suppliers in Japan, something you might do if new export fees were introduced.

Listing 2. You can use a subquery in UPDATE to determine which records to change.

```
UPDATE Products ;
SET UnitPrice = 1.1 * UnitPrice ;
WHERE SupplierID IN ( ;
  SELECT SupplierID ;
  FROM Suppliers ;
  WHERE UPPER(Country) = "JAPAN")
```



The code in Listing 2 is included in the Developer Downloads for this chapter, available from www.hentzenwerke.com, as *RaisePrices.PRG*.

You can combine queries with the other commands to get the desired results. The code in **Listing 3** finds contact items not currently in use and deletes them. The query creates a cursor of items for which all uses have expired. The subquery in the SELECT gets a list of items currently in use with no expiration date (in other words, those to be used for the foreseeable future). The query eliminates those contact items from consideration, and then finds the last expiration date for the remaining items. The HAVING clause keeps only those items for which the last expiration date has passed. The DELETE command then uses a simple subquery to find the expired items and remove them from the ContactItem table.

Listing 3. Both the SELECT and the DELETE command here use subqueries. The two commands combine to delete all contact items for which all uses have expired.

```
SELECT iItemFK, MAX(dEnds) as dLastDate ;
  FROM PersonToItem ;
  WHERE iItemFK NOT in ( ;
      SELECT iItemFK ;
      FROM PersonToItem ;
      WHERE EMPTY(dEnds)) ;
  GROUP BY iItemFK ;
  HAVING dLastDate < DATE() ;
  INTO CURSOR Expired

DELETE FROM ContactItem;
  WHERE ContactItem.iID IN ( ;
      SELECT iItemFK FROM Expired)
```

VFP 9 offers several other ways to handle this example. See “Derived tables—Subqueries in FROM” later in this chapter and Chapter 6, “Complex Data Manipulation.”



The code in Listing 3 is included in the Developer Downloads for this chapter, available from www.hentzenwerke.com, as *DeleteExpired.PRG*.

Subquery operators other than IN don’t usually occur with free-standing subqueries; instead, they tend to be used for correlated subqueries.

Correlation

Sometimes, to get the desired results, a subquery needs to refer to a field of a table from the containing command. Such a subquery is said to be *correlated*. At least from a logical point of view, a correlated subquery executes once for each row in the containing command. Correlated subqueries are more likely than stand-alone subqueries to use operators other than IN.

For example, the query in **Listing 4** offers another solution to a problem posed in Chapter 4, “Retrieving Data:” how to get additional information from a child record when performing aggregation. The subquery here is designed to return a single value. It’s correlated with the

Orders table in the main query based on the CustomerID field. (Note the use of the local alias, MinOrd, for the Orders table in the subquery.) Each time the subquery runs, it looks at orders for a single customer; for each customer, the subquery extracts the date of that customer's first order in September, 1996. The main query then keeps only records with the same order data for that customer.

Listing 4. *A correlated subquery here avoids the issue of extracting additional information from a child record when grouping.*

```
SELECT Customers.CustomerID, CompanyName, ;
       OrderDate, ShippedDate ;
FROM Customers ;
JOIN Orders ;
ON Customers.CustomerID = Orders.CustomerID ;
WHERE OrderDate = ;
      (SELECT MIN(OrderDate) FROM Orders MinOrd ;
       WHERE MinOrd.CustomerID = Orders.CustomerID ;
       AND OrderDate BETWEEN {^ 1996-9-1} AND {^ 1996-9-30}) ;
INTO CURSOR FirstOrder
```

Listing 5 combines a correlated subquery with the EXISTS operator to get a list of all the customers who have placed suspicious orders. Here, the subquery (which is the same query as in Listing 58 in Chapter 4, "Retrieving Data") is correlated with the Customer table based on both the CustomerID field and the address. The subquery finds a list of suspicious orders for each customer; if there are any records returned, the main query extracts the customer id and company name. The final result is a list of companies who placed orders deemed suspicious.

Listing 5. *Use the EXISTS operator to find records for which the subquery produces results. With EXISTS, you'll almost always use a correlated subquery.*

```
SELECT CustomerID, CompanyName ;
FROM Customers ;
WHERE EXISTS ( ;
  SELECT Orders.OrderID ;
  FROM Orders ;
  JOIN OrderDetails ;
  ON Orders.OrderID=OrderDetails.OrderID ;
  WHERE Orders.CustomerID=Customers.CustomerID ;
  AND Orders.ShipAddress <> Customers.Address ;
  GROUP BY Orders.OrderID ;
  HAVING SUM(Quantity*UnitPrice)> 4000) ;
INTO CURSOR SuspiciousCustomers
```

Listing 5 demonstrates one of the changes in VFP 9. In VFP 8 and earlier, a correlated subquery could not use the GROUP BY clause. VFP 9 lifts that restriction.



The queries in Listing 4 and Listing 5 are included in the Developer Downloads for this chapter, available from www.hentzenwerke.com, as FirstOrderCorrelated.PRG and SuspiciousCustomers.PRG, respectively.

The unusual operators

As indicated in Table 1, you can also use the ALL or ANY/SOME operators in the WHERE clause. ALL compares the specified expression to every record returned by the subquery (which must include only a single field in its field list). Only records where the expression meets the specified condition (based on the comparison operator used) for every subquery record appear in the results. For example, the query in **Listing 6** creates a list of customers who have never shipped to the address listed in the Customers table. Note that the subquery is correlated with the Customers table based on the customer ID.

Listing 6. Use ALL to compare an expression to every record in a subquery.

```
SELECT CustomerID, CompanyName, Address ;
FROM Customers ;
WHERE Address <> ALL ;
    (SELECT ShipAddress ;
     FROM Orders ;
     WHERE Orders.CustomerID = Customers.CustomerID) ;
INTO CURSOR DontShipHere
```

Of course, this query can be performed using NOT IN rather than \diamond ALL.



The query in Listing 6 is included in the Developer Downloads for this chapter, available from www.hentzenwerke.com, as DontShipHere.PRG

The ANY or SOME operator performs the same type of comparison as ALL, but includes a record in the result if it has the specified relationship to any record in the subquery's results. For an example, see the next section, "Derived Tables—Subqueries in FROM."

Derived Tables—subqueries in FROM

Beginning in VFP 9, you can use subqueries in the FROM clause of SELECT, DELETE, and UPDATE. (The FROM clause itself is new to UPDATE in VFP 9, as well. Be aware that FROM in UPDATE is an extension to the SQL-92 standard.) The subquery result, which can then be joined with other tables, is called a *derived table*. While you can usually solve the same problems by using a series of commands, derived tables often let you perform a process with a single command; the single command can be faster than the series it replaces.

Like any other subquery, a subquery in the FROM clause is enclosed in parentheses. You must assign a local alias to the derived table and use the local alias to refer to fields of the subquery result in the containing command. Derived tables cannot be correlated; VFP must be able to run the subquery before any joins are performed in the containing command.

A derived table is useful for extracting additional child data after grouping. In many cases, the grouping can be performed in a derived table. **Listing 7** shows another solution to the problem of finding each customer's first order in a specified month and including the shipping date for that order in the results. Several solutions to this problem are included in Chapter 4, "Retrieving Data"; Listing 4 demonstrates another approach.

Listing 7. A subquery in the FROM clause creates a cursor on the fly that you can join to other tables in the command.

```
SELECT Customers.CustomerID, CompanyName, ;
       OrderDate, ShippedDate ;
FROM Customers ;
JOIN Orders ;
ON Customers.CustomerID = Orders.CustomerID ;
JOIN (SELECT CustomerID, MIN(OrderDate) MinDate ;
      FROM Orders AS MinOrd ;
      WHERE OrderDate BETWEEN {^ 1996-9-1} AND {^ 1996-9-30} ;
      GROUP BY 1) AS FirstCustOrder ;
ON FirstCustOrder.CustomerID = Orders.CustomerID ;
AND FirstCustOrder.MinDate = Orders.OrderDate ;
INTO CURSOR FirstOrder
```



The code in Listing 7 is included in the Developer Downloads for this chapter, available from www.hentzenwerke.com, as *FirstOrderDerived.PRG*.

You can combine the various uses of subqueries, so a derived table can include a subquery and a subquery can use a derived table. **Listing 8** demonstrates the latter ability, along with the use of the ANY keyword. It finds all products for which a single order contains more units than were ordered in any entire year for that product. That is, it's looking for unusually large orders of any product.

Listing 8. Even a subquery can use a derived table.

```
SELECT OrderID, ProductID, Quantity ;
FROM OrderDetails MainDetails ;
WHERE Quantity > ANY ;
      (SELECT SUM(Quantity) ;
      FROM (SELECT ProductID, YEAR(OrderDate) as nYear, Quantity ;
      FROM OrderDetails ;
      JOIN Orders ;
      ON OrderDetails.OrderID = Orders.OrderID ) ProdYear ;
      WHERE ProdYear.ProductID = MainDetails.ProductID ;
      GROUP BY nYear) ;
INTO CURSOR MoreThanAYear
```

A derived table offers another approach to the problem in Listing 3, deleting expired contact information. Rather than performing a query followed by DELETE, you can combine the two into a single DELETE command, as in **Listing 9**. The DELETE command has a subquery. That subquery simply extracts the itemFK field from a derived table. The subquery that computes the derived table also uses a subquery. So this example demonstrates both a subquery containing a derived table and a derived table containing a subquery.

Listing 9. A derived table often lets you do in one command what would otherwise take two or more.

```
DELETE FROM ContactItem;
WHERE ContactItem.iID IN (
  SELECT iItemFK FROM (
    SELECT iItemFK, MAX(dEnds) as dLastDate ;
    FROM PersonToItem ;
    WHERE iItemFK NOT in (
      SELECT iItemFK ;
      FROM PersonToItem ;
      WHERE EMPTY(dEnds) ) ;
    GROUP BY iItemFK ;
    HAVING dLastDate < DATE() ) Expired )
```

Another difference between this version and the version in Listing 3 is that the version with a query followed by DELETE leaves the cursor Expired open; the derived table version does not.



The examples in Listing 8 and Listing 9 are included in the Developer Downloads for this chapter, available from www.hentzenwerke.com, as MoreThanAYear.PRG and DeleteExpiredDerived.PRG, respectively.

Chapter 6, “Complex Data Manipulation,” includes additional examples of derived tables.

Subqueries in the field list

Another VFP 9 change lets you put subqueries into the field list of a query. A subquery in the field list must have only one field in its own field list. For each record in the containing query, it must return no more than one record. If the subquery returns no records for a record in the containing query, that field in the result gets the null value.

As with other subqueries, these must be surrounded by parentheses. It’s a good idea to specify a name for the resulting field of the containing query. If you don’t, VFP generates one for you. You can specify it either by following the subquery with AS and the name, or by renaming the field in the subquery. Subqueries in the field list can be correlated and usually are.

A subquery in the field list offers an alternative solution to the problem of selecting additional fields from a parent table when grouping. Perform the grouping in the subquery and the containing query is significantly simplified.

Listing 10 shows another way to solve a problem originally posed in Chapter 4, “Retrieving Data.” The goal is to retrieve a bunch of customer information along with the total of the customer’s order for a specified month. In this example, with two aggregates to be computed and only two fields from the parent table, the earlier solutions may be easier to maintain and faster. In cases where you need only one aggregated field and there are lots of fields from the parent table, the subquery solution is faster.

Listing 10. The subquery in the field list here avoids problems related to grouping and aggregation.

```
SELECT Customers.CustomerID, Customers.CompanyName, ;
      (SELECT SUM(Quantity) ;
       FROM Orders ;
       JOIN OrderDetails ;
       ON Orders.OrderID = OrderDetails.OrderID ;
       WHERE Orders.CustomerID = Customers.CustomerID ;
       AND OrderDate BETWEEN {^ 1996-9-1} AND {^ 1996-9-30}) ;
      AS ItemCount, ;
      (SELECT SUM(Quantity*UnitPrice) ;
       FROM Orders Orders2 ;
       JOIN OrderDetails OD2 ;
       ON Orders2.OrderID = OD2.OrderID ;
       WHERE Orders2.CustomerID = Customers.CustomerID ;
       AND OrderDate BETWEEN {^ 1996-9-1} AND {^ 1996-9-30}) ;
      AS OrderTotal ;
FROM Customers ;
INTO CURSOR CustomerMonthly
```



The query in Listing 10 is included in the Developer Downloads for this chapter, available from www.hentzenwerke.com, as *MonthOrderTotalSubquery.PRG*.

Computing update values

VFP 9 introduced another new home for subqueries—the SET clause of UPDATE. Of all the uses for subqueries, this is the most restrictive. Only one field in the SET list can use a subquery and when it does, the WHERE clause of that UPDATE command cannot use a subquery. (This is a VFP limitation; the SQL standard allows any number of fields to use subqueries.)

For the one field, you can base the replacement value on the results of a subquery. Typically, such a subquery is correlated to match the computed result to the record being updated. If the subquery returns an empty result, the specified field gets the null value.

The example in **Listing 11** (*UpdatePricesSubquery.PRG*) updates the unit price for those items listed in the *NewPrices* cursor. *NVL()* prevents the unit price from being overwritten for those products not in the cursor.

Listing 11. Beginning in VFP 9, UPDATE can use a subquery in the SET clause to specify the new value.

```
CREATE CURSOR NewPrices (iProductID I, yPrice Y)
INSERT INTO NewPrices ;
      VALUES (1, $20)
INSERT INTO NewPrices ;
      VALUES (2, $22)
INSERT INTO NewPrices ;
      VALUES (3, $12.50)
```



```
UPDATE Products ;
SET UnitPrice = ;
  NVL((SELECT yPrice FROM NewPrices ;
  WHERE Products.ProductID = NewPrices.iProductID), UnitPrice)
```



The code in Listing 11 is included in the Developer Downloads for this chapter, available from www.hentzenwerke.com, as UpdatePricesSubquery.PRG.

While there are some uses for a subquery in SET, another VFP 9 feature gives UPDATE far more flexibility; see Chapter 6, “Complex Data Manipulation.”

Put subqueries to work

The examples in this chapter show some prototypical uses for subqueries. The next chapter shows a lot more ways that subqueries can contribute to data collection and manipulation.

