Joins and subqueries: Using SQL commands for the hard stuff

Tamar E. Granor
Tomorrow's Solutions, LLC
8201 Cedar Road
Elkins Park, PA 19027
Voice: 215-635-1958
Email: tamar@tomorrowssolutionsllc.com

Writing simple SQL queries isn't hard to learn. But as soon as you need something a little more complex, you get tangled up in inner joins vs. outer joins and where to use a subquery and so forth.

This session bridges the gap between simple and complex queries with a deep dive into the JOIN clause and the use of subqueries. We'll also talk about how to optimize queries and the tools VFP provides to help you.

While this session focuses on VFP's SQL commands, much of the information is applicable to other SQL dialects. This session assumes some familiarity with the SQL-SELECT command.

Visual FoxPro's SQL sublanguage provides an easy way to perform some tasks. The ability to specify what you want rather than how to find it means that you don't have to worry about work areas or index orders or record pointers. In particular, the SQL SELECT command lets you gather a group of records based on their content.

Writing queries to pull data out of a single table is fairly simple. But when you get beyond that, it's easy to get confused about what you need to write. Two key elements of more complex queries are joins and subqueries. This paper digs into each of them to see how they work and what capabilities they provide.

The examples here use two databases. Many examples are based on the Northwind database that comes with VFP (8 and 9). In addition, some examples use a Contacts database that's included in the downloads for this session. The Contacts database treats the different ways of contacting an individual as parallel and is designed to make it easy to handle new types of contact information. (For example, in the years since I first created this example, many people have acquired Facebook, LinkedIn and Twitter IDs.)

The database, called Contacts, includes the following tables:

- **Person** stores the name and birth date for each person in the database. In a production application, it would probably include additional personal information.
- **ItemType** is a look-up table for contact items. Each record represents one type of contact item, such as phone, fax, email, address, and so forth.
- **Location** is a look-up table for the "places" a contact item may be associated with. Each record represents one place (or type of place), such as personal, business, or school.
- **ContactItem** contains information about a single contact item (address, phone number, email, URL), using a link to the ItemType look-up table to indicate the type of item. The actual contact information is stored in a memo field to avoid formatting and size concerns.
- **PersonToItem** is a many-to-many join table linking people to contact items. Each record in PersonToItem also has a pointer to Location to indicate where the item applies. The lPreferred field identifies the preferred item among several of the same type and location, while dEffective and dEnds indicate when that contact item takes effect and when it's no longer valid for that person.

Each table has a surrogate primary key field called iID; these fields are auto-incrementing integers. Foreign keys are indicated by FK at the end of the field name. For example, iTypeFK in ContactItem is a foreign key to ItemType.

Figure 1 shows the structure of the Contacts database, including the persistent relations.

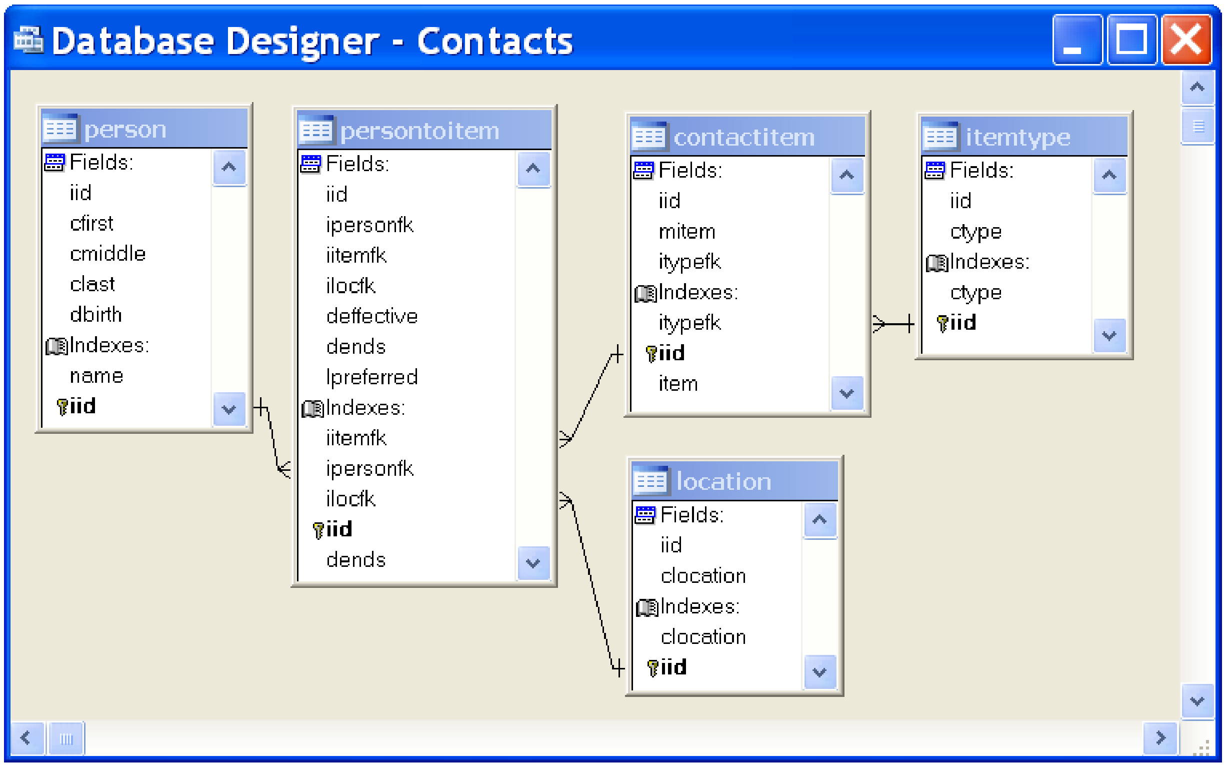


Figure 1. The Contacts database offers a normalized and extensible way of representing people and their contact information.

Most of the examples in this paper are included in the downloads, as well. In most cases, the name of the PRG files matches the name of the cursor created in the example without the "csr" prefix.

Inner join, Outer join, Self join

While some queries (such as those used to collect information for a picklist) involve only a single table, most queries use two or more tables. (Through VFP 8, you can include up to 30 tables. VFP 9 has no limit.) Combining more than two tables introduces a variety of issues.

The JOIN clause

When a query includes more than one table, you need to specify how records from the tables are to be matched up to produce the result set. This is called *joining* the tables. The JOIN sub-clause of SQL-SELECT's FROM clause is used for this purpose.

A join has two parts: the pair of tables to be joined, and the condition for matching records. The syntax looks like Listing 1.

Listing 1. The JOIN clause lists the tables to be joined and the expression that matches up records in the two tables.

```
<Table1> JOIN <Table2> ON <expression>
```

Most often, the expression following the ON keyword (called the *join condition*) matches the primary key (PK) of one table with a foreign key (FK) in another table. For example, using the sample Northwind database, the query in Listing 2 produces a list of products by supplier. To do so, it matches the SupplierID field of Suppliers (the PK) to the SupplierID of Products.

Listing 2. The simplest joins match the primary key of one table with a foreign key in another.

```
SELECT CompanyName, ProductName ;
  FROM Products ;
    JOIN Suppliers ;
      ON Suppliers.SupplierID = Products.SupplierID ;
  ORDER BY CompanyName, ProductName ;
  INTO CURSOR csrProductsBySupplier
```

However, the expression can be more complex than a simple comparison. For example, if the criteria for joining two tables involve multiple fields, you can use AND and OR to combine multiple comparisons. (The example databases, Northwind and Contacts, don't include any tables where you need multiple fields to join.)

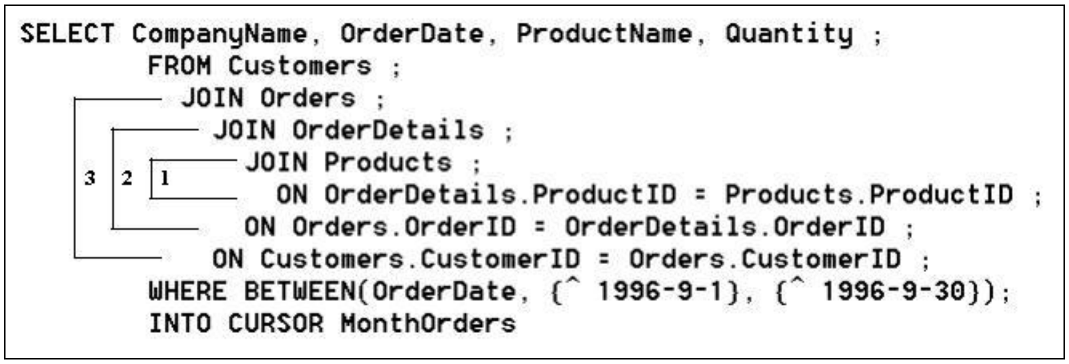
When more than two tables are involved in a query, VFP provides two ways to specify joins. The first, *nested*, style lists all the tables first, then lists all the join conditions. The last two tables listed are joined first, using the innermost join condition. Then, the next-to-last table listed is joined with the result of the first join, using the second join condition, and so forth.

The query in Listing 3 gathers information about the products ordered in a single month, using the nested style to join the tables.

Listing 3. The nested style works best when joining a series of tables with a hierarchical relationship.

```
SELECT CompanyName, OrderDate, ProductName, Quantity ;
   FROM Customers ;
     JOIN Orders ;
       JOIN OrderDetails ;
         JOIN Products ;
           ON OrderDetails.ProductID = Products.ProductID ;
         ON Orders.OrderID = OrderDetails.OrderID ;
       ON Customers.CustomerID = Orders.CustomerID ;
   WHERE BETWEEN(OrderDate, {^ 1996-9-1}, {^ 1996-9-30});
   INTO CURSOR csrMonthOrders
```

Figure 2 shows how the joins and join conditions are matched. First, OrderDetails and Products are joined, using the first ON clause: OrderDetails.ProductID = Products.ProductID. Then, that intermediate result is joined with Orders using the condition Orders.OrderID = OrderDetails.OrderID. Finally, the second intermediate result is joined with Customers, based on the condition Customers.CustomerID = Orders.CustomerID. (Be aware that the actual order in which the joins are performed can be different from the logical order indicated by the query itself. When it won't affect the query results, VFP's SQL engine performs joins in what it deems the most efficient order.)



```
SELECT CompanyName, OrderDate, ProductName, Quantity ;
       FROM Customers ;
         JOIN Orders ;
           JOIN OrderDetails ;
             JOIN Products ;
               ON OrderDetails.ProductID = Products.ProductID ;
             ON Orders.OrderID = OrderDetails.OrderID ;
           ON Customers.CustomerID = Orders.CustomerID ;
       WHERE BETWEEN(OrderDate, {^ 1996-9-1}, {^ 1996-9-30});
       INTO CURSOR MonthOrders
```

Figure 2. Nested joins work from the inside out. Here, the first join performed is between OrderDetails and Products and it uses the condition OrderDetails.ProductID = Products.ProductID.

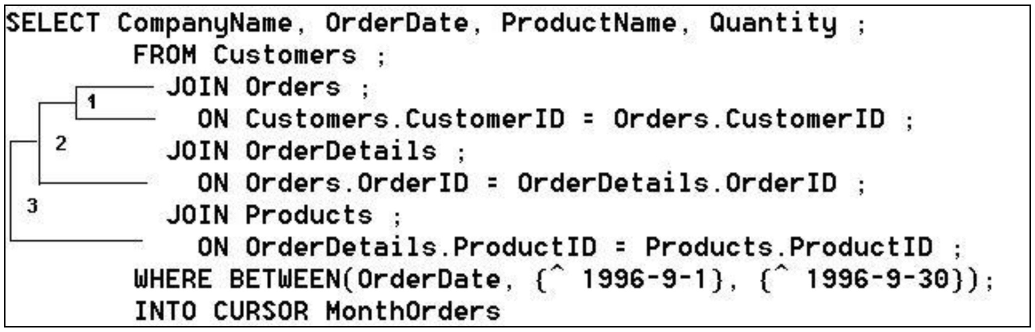
While the nested join style works best for hierarchical data, the alternative, *sequential*, style can be used in any situation. It lists the joins one at a time. Each time JOIN is specified to add a table, it's followed by ON, which tells how to join that table to the results so far. While nested joins are processed from the inside out, sequential joins are processed from the top down. So the first two tables listed are joined based on the first ON condition, then that intermediate result is joined with the next table based on the second ON condition, and so forth. (As with the nested style, the actual order of the joins may vary for performance reasons.)

Listing 4 shows the same query as above, gathering information about orders for a single month, but using the sequential style for joins.

Listing 4. The sequential style can be used to join any set of tables, including those with hierarchical relationships.

```
SELECT CompanyName, OrderDate, ProductName, Quantity ;
   FROM Customers ;
     JOIN Orders ;
       ON Customers.CustomerID = Orders.CustomerID ;
     JOIN OrderDetails ;
       ON Orders.OrderID = OrderDetails.OrderID ;
     JOIN Products ;
       ON OrderDetails.ProductID = Products.ProductID ;
   WHERE BETWEEN(OrderDate, {^ 1996-9-1}, {^ 1996-9-30});
   INTO CURSOR csrMonthOrders
```

Figure 3 shows the logical order of joins for this query. Customer and Orders are joined first, using the condition Customers.CustomerID = Orders.CustomerID. Then the result is joined with OrderDetails, based on the condition Orders.OrderID = OrderDetails.OrderID. Finally, that result is joined with Products using the last ON condition: OrderDetails.ProductID = Products.ProductID.



```
SELECT CompanyName, OrderDate, ProductName, Quantity ;
       FROM Customers ;
         JOIN Orders ;
           ON Customers.CustomerID = Orders.CustomerID ;
         JOIN OrderDetails ;
           ON Orders.OrderID = OrderDetails.OrderID ;
         JOIN Products ;
           ON OrderDetails.ProductID = Products.ProductID ;
       WHERE BETWEEN(OrderDate, {^ 1996-9-1}, {^ 1996-9-30});
       INTO CURSOR MonthOrders
```

Figure 3. Sequential joins are executed from the top down. Here, the first join is between Customers and Orders based on the condition Customers.CustomerID = Orders.CustomerID.

You're not restricted to using either the sequential or the nested style in a query; you can mix the two. For example, the query in Listing 5 uses both to collect information about the products ordered in a month, including shipper information.

Listing 5. A query can use both sequential and nested joins. For each join, choose the style that makes it easiest to read and maintain the query.

```
SELECT Customers.CompanyName AS Customer, ;
       OrderDate, ProductName, Quantity, ;
       Shippers.CompanyName AS Shipper ;
   FROM Customers ;
     JOIN Orders ;
       JOIN OrderDetails ;
         JOIN Products ;
           ON OrderDetails.ProductID = Products.ProductID ;
         ON Orders.OrderID = OrderDetails.OrderID ;
       ON Customers.CustomerID = Orders.CustomerID ;
     JOIN Shippers ;
       ON Orders.ShipVia = Shippers.ShipperID ;
   WHERE OrderDate BETWEEN {^ 1996-9-1} AND {^ 1996-9-30};
   INTO CURSOR csrMonthOrders
```

The first three joins, which traverse the hierarchy from Customer down to Products, are nested. The final join, which connects the shipper to the rest of the information, is sequential.

While the nested and sequential approaches produce the same results, I find the sequential version easier to read and tend to use it almost exclusively.

Outer joins

All of the multi-table queries in the preceding section include records from one table only if a match is found in the other tables. For example, only customers who placed orders in September, 1996, appear in the results of Listing 5. Similarly, only those products that were ordered in that month are included in the results. Such a join, which filters out records without matches, is called an *inner join*. Unless you specify otherwise, all joins are inner; you can explicitly specify that you want an inner join in VFP by including the keyword INNER before JOIN.

There are situations where you want to include the unmatched records, as well. A join that includes all the records from one or both of its tables, regardless of matches in the other table, is called an *outer join*. There are three types of outer joins: *left joins*, *right joins* and *full joins*. In a left join, all the records from the table listed before the JOIN keyword are included, along with whichever records they match from the table after the JOIN keyword; use LEFT JOIN or LEFT OUTER JOIN to specify a left join. A right join, indicated by RIGHT JOIN or RIGHT OUTER JOIN, is the opposite; the results include all the records from the table listed after the JOIN keyword, along with whichever records they match from the table listed before the JOIN keyword. A full join includes all the records from both tables, making matches where possible; for a full join, use FULL JOIN or FULL OUTER JOIN.

When you perform an outer join, there may be no values available for some of the fields in the field list for some of the records in the result. In that situation, those fields are set to null.

Let's start with a simple example. Suppose we want a list of all customers, along with the dates of any orders the customer placed. Listing 6 collects that information.

Listing 6. Outer joins include all the records from at least one table; where no matching record exists, fields are set to null.

```
SELECT CompanyName, OrderDate;
   FROM Customers ;
     LEFT JOIN Orders ;
       ON Customers.CustomerID = Orders.CustomerID ;
   INTO CURSOR csrOrdersAllCustomers
```

To see the effect of the outer join, look at the results for FISSA Fabrica Inter. Salchichas S.A. and Paris spécialités. In each case, there's a single record and the OrderDate column contains .null.; if you check the Orders table, you'll see that neither of these companies has placed any orders.

When only two tables are involved, left joins and right joins are interchangeable; you just need to change the order of the tables. So, the query in Listing 7 returns the same results as the one in Listing 6.

Listing 7. You can switch left and right joins by changing the order of the tables.

```
SELECT CompanyName, OrderDate;
   FROM Orders ;
     RIGHT JOIN Customers ;
       ON Customers.CustomerID = Orders.CustomerID ;
   INTO CURSOR csrOrdersAllCustomers
```

You can combine outer joins with filters, but there are some caveats. If you're filtering on the table on the "all" side (that is, the table from which you want to include all records), there's no problem. The filter removes those records that don't meet the conditions, but the outer join ensures that all other records from that table appear in the result. For example, the query in Listing 8 includes all customers in France, along with the dates of their orders; Paris spécialités is included with a null order date.

Listing 8. It's easy to filter the "all" side of an outer join.

```
SELECT CompanyName, OrderDate;
   FROM Customers ;
     LEFT JOIN Orders ;
       ON Customers.CustomerID = Orders.CustomerID ;
   WHERE Customers.Country = 'France' ;
   INTO CURSOR csrOrdersAllCustomersFrance
```

Filtering on the "some" side (the table from which only matching records are included) is a little trickier. To get the desired results, you have to move the filter condition into the join condition.

To limit the list to include orders only from a specified month, you might be inclined to write the query in Listing 9.

Listing 9. When using outer joins, filter conditions may have unexpected results.

```
SELECT CompanyName, OrderDate;
   FROM Customers ;
     LEFT JOIN Orders ;
       ON Customers.CustomerID = Orders.CustomerID ;
   WHERE OrderDate BETWEEN {^ 1996-9-1} AND {^ 1996-9-30} ;
   INTO CURSOR csrOrdersAllCustomers
```

However, the resulting cursor contains only customers who've placed an order in the specified month, with one record for each order in the month. That's because the outer join is performed before the filter condition for Orders is applied. So Customer and Orders are joined, making sure to include every customer. Then the filter condition removes the records for any orders outside the specified month. At this point, all records are removed for any customer who didn't place any orders in that month (including those customers who never placed any orders and thus have .null. for the order date) and those customers don't appear in the results.

To get the desired results, you need the query in Listing 10, with the filter condition included in the ON clause.

Listing 10. In some situations involving outer joins, filter conditions need to become part of the join condition.

```
SELECT CompanyName, OrderDate;
   FROM Customers ;
     LEFT JOIN Orders ;
       ON Customers.CustomerID = Orders.CustomerID ;
          AND OrderDate BETWEEN {^ 1996-9-1} AND {^ 1996-9-30} ;
   INTO CURSOR csrOrdersAllCustomers
```

When you run this query, the filter condition is applied as the two tables are joined. So records from the Customer table are matched with only those Orders records from the specified month. After that's done, records are added for each missing customer. As a result, the cursor includes all the records extracted by the previous version, but it also has a single record for each customer who didn't place an order in that month. In those records, the OrderDate field is set to null.

When a query includes an outer join, the order of the joins becomes much more significant. In order to get accurate results, you may need to perform the joins in a particular order. In some situations, you may need to change some inner joins into outer joins to carry unmatched results along.

Listing 11 gathers all business contact information for each person in the Contacts database. All people are listed, even if they have no business contact information.

Listing 11. The order in which you list joins can be more significant with outer joins. In some cases, you must put the joins in a specific order to get the desired results.

```
SELECT cFirst, cLast, mItem ;
   FROM PersonToItem ;
     JOIN ContactItem ;
       ON PersonToItem.iItemFK = ContactItem.iID ;
     JOIN Location ;
       ON PersonToItem.iLocFK = Location.iID ;
      AND UPPER(cLocation)="BUSINESS" ;
     RIGHT JOIN Person ;
       ON Person.iID = PersonToItem.iPersonFK ;
   INTO CURSOR csrPersonBusinessContacts
```

To make this query work, the join with Person needs to come last. Doing it earlier results in either eliminating those people with no business contact information or including personal contact information in the results, depending on which other joins are set as outer joins. Note that the join with Location uses a filter condition, even though it's not an outer join; as in Listing 10, putting this condition in the WHERE clause gives inaccurate results.

The next example uses the Northwind database to extract a list of all companies and any demographic information stored for them. (In fact, the CustomerDemographics table that comes with VFP is empty, which makes this an even better demonstration of the power of outer joins.) Clearly, to include all companies, you need an outer join between Customers and CustomerCustomerDemographics. But that's not sufficient—if you do that join first, you also need an outer join between CustomerCustomerDemographics and CustomerDemographics, as in Listing 12, to keep those records added to the intermediate result by the outer join from falling out during the second join, when no record matches.

Listing 12. To keep records pulled into an outer join from being removed, you may need to use outer joins for subsequent joins in the same query.

```
SELECT CompanyName, CustomerDesc ;
   FROM Customers ;
     LEFT JOIN CustomerCustomerDemo AS CCD ;
       ON Customers.CustomerID = CCD.CustomerID ;
     LEFT JOIN CustomerDemographics CustDemo;
       ON CCD.CustomerTypeID = CustDemo.CustomerTypeID ;
   INTO CURSOR csrCompaniesDemographics
```

Local aliases

The example in Listing 12 also demonstrates the use of *local aliases*. When you open a table with the USE command, you can specify a name, or alias, by which that table is referenced. SELECT gives you a similar capability. You can assign an alias to any table in the query; the alias you assign applies only within that query. Once you assign a local alias, you must use the alias rather than the table name to refer to that table throughout the query.

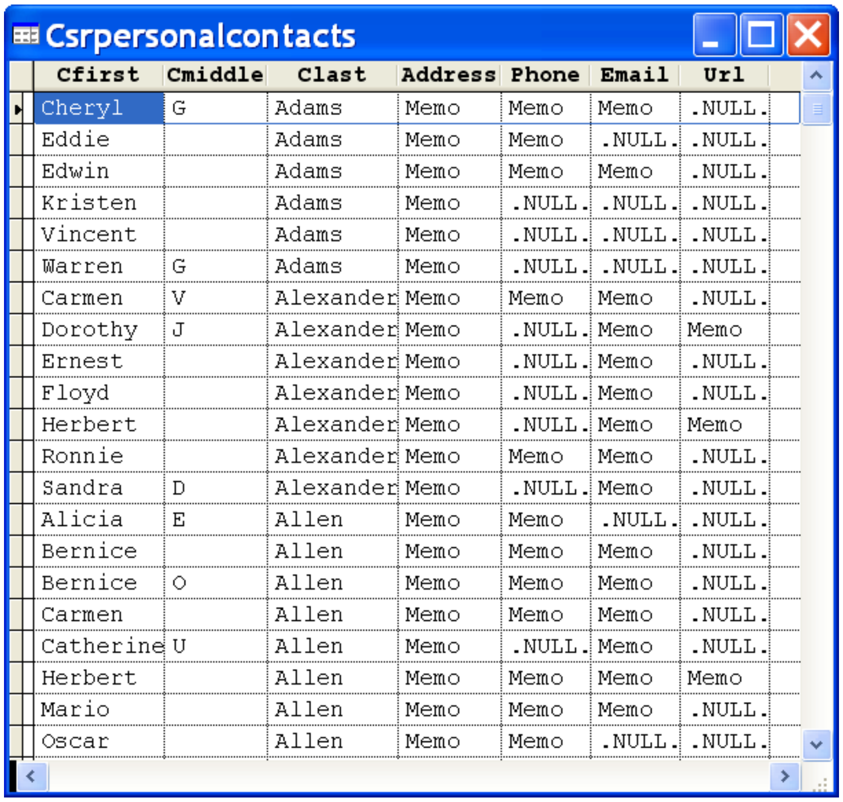
To assign a local alias, follow the table name with the alias. If you choose, you can use the AS keyword before the local alias, as in Listing 12. Many people find queries more readable with the AS keyword.

Local aliases are useful in several situations. The first, shown above, is for dealing with long, unwieldy table names; a local alias simply cuts down on the size of the query and the typing needed to create it. The second, more important, use for local aliases is when a single table appears more than once in a query.

One situation where you may need to use the same table more than once is in denormalizing normalized data. For example, the long query in Listing 13 assembles the preferred personal contact information for each person in the Contacts database, putting the name, address, phone number, email address and website address for each person into a single record. The JOIN clause of this query has four sections. The first collects telephone information; it uses the PersonToItem, ContactItem, ItemType and Location tables, giving each a local alias that includes the word "phone." The other three sections are structurally identical; they gather address, email and web information, respectively, using appropriate local aliases. The result of this query is one record per person, showing that person's name, and preferred personal phone number, address, email address and URL. Figure 4 shows part of the results.

Listing 13. When denormalizing data, you may need to use the same table more than once in a query. In such cases, you need local aliases.

```
SELECT cFirst, cMiddle, cLast, ;
       Address.mItem AS Address, Phone.mItem AS Phone, ;
       Email.mItem AS Email, Web.mItem AS URL ;
   FROM Person ;
     LEFT JOIN PersonToItem PhoneLink ;
       JOIN ContactItem Phone ;
         JOIN ItemType PhoneType ;
           ON Phone.iTypeFK = PhoneType.iID ;
              AND PhoneType.cType="Voice" ;
         ON PhoneLink.iItemFK = Phone.iID ;
            AND PhoneLink.lPreferred ;
       JOIN Location PhoneLoc ;
         ON PhoneLink.iLocFK = PhoneLoc.iID ;
            AND PhoneLoc.cLocation = "Personal" ;
       ON Person.iID = PhoneLink.iPersonFK ;
     LEFT JOIN PersonToItem AddrLink ;
       JOIN ContactItem Address ;
         JOIN ItemType AddrType ;
           ON Address.iTypeFK = AddrType.iID ;
              AND AddrType.cType = "Address" ;
         ON AddrLink.iItemFK = Address.iID ;
            AND AddrLink.lPreferred ;
       JOIN Location AddrLoc ;
         ON AddrLink.iLocFK = AddrLoc.iID ;
            AND AddrLoc.cLocation = "Personal" ;
       ON Person.iID = AddrLink.iPersonFK ;
     LEFT JOIN PersonToItem EmailItem;
       JOIN ContactItem Email ;
         JOIN ItemType EmailType ;
           ON Email.iTypeFK = EmailType.iID ;
              AND EmailType.cType="Email" ;
         ON EmailItem.iItemFK = Email.iID ;
            AND EmailItem.lPreferred ;
       JOIN Location EmailLoc;
         ON EmailLoc.iID = EmailItem.iLocFK ;
            AND EmailLoc.cLocation="Personal" ;
       ON Person.iID = EmailItem.iPersonFK ;
     LEFT JOIN PersonToItem WebItem;
       JOIN ContactItem Web ;
         JOIN ItemType WebType ;
           ON Web.iTypeFK = WebType.iID ;
              AND WebType.cType="URL" ;
         ON WebItem.iItemFK = Web.iID ;
            AND WebItem.lPreferred ;
       JOIN Location WebLoc;
         ON WebLoc.iID = WebItem.iLocFK ;
            AND WebLoc.cLocation="Personal" ;
       ON Person.iID = WebItem.iPersonFK ;
   ORDER BY cLast, cFirst, cMiddle ;
   INTO CURSOR csrPersonalContacts
```



Csrpersonalcontacts

Cfirst	Cmiddle	Clast	Address	Phone	Email	Url
Cheryl	G	Adams	Memo	Memo	Memo	.NULL.
Eddie		Adams	Memo	Memo	.NULL.	.NULL.
Edwin		Adams	Memo	Memo	Memo	.NULL.
Kristen		Adams	Memo	.NULL.	.NULL.	.NULL.
Vincent		Adams	Memo	.NULL.	.NULL.	.NULL.
Warren	G	Adams	Memo	.NULL.	.NULL.	.NULL.
Carmen	V	Alexander	Memo	Memo	Memo	.NULL.
Dorothy	J	Alexander	Memo	.NULL.	Memo	Memo
Ernest		Alexander	Memo	.NULL.	Memo	.NULL.
Floyd		Alexander	Memo	.NULL.	Memo	.NULL.
Herbert		Alexander	Memo	.NULL.	Memo	Memo
Ronnie		Alexander	Memo	Memo	Memo	.NULL.
Sandra	D	Alexander	Memo	.NULL.	Memo	.NULL.
Alicia	E	Allen	Memo	Memo	.NULL.	.NULL.
Bernice		Allen	Memo	Memo	Memo	.NULL.
Bernice	O	Allen	Memo	Memo	Memo	.NULL.
Carmen		Allen	Memo	Memo	Memo	.NULL.
Catherine	U	Allen	Memo	.NULL.	Memo	.NULL.
Herbert		Allen	Memo	Memo	Memo	Memo
Mario		Allen	Memo	Memo	Memo	.NULL.
Oscar		Allen	Memo	Memo	.NULL.	.NULL.

Figure 4. Local aliases are helpful in denormalizing normalized data.

This query also demonstrates one of the significant changes in VFP 9; the query has 16 joins. VFP 8 and earlier handle a maximum of 9 joins in a single query.

Self-joins

In some situations, not only do you need to use a particular table more than once in a query, you actually need to join one instance of the table to the other. Such a join is called a *self-join*.

In Listing 14, the Northwind Products table is joined with itself to find cases where the same supplier offers two products in the same category. (Note that the query doesn't gracefully handle the case of more than two products in the same category from the same supplier; each pair is listed.)

Listing 14. Self-joins let you do comparisons with data in the same table.

```
SELECT First.ProductName AS FirstProduct, ;
       Second.ProductName AS SecondProduct, ;
       CompanyName, CategoryName ;
   FROM Products First ;
     JOIN Products Second ;
       ON First.SupplierID = Second.SupplierID ;
       AND First.CategoryID = Second.CategoryID ;
```

```
      AND First.ProductID < Second.ProductID ;
    JOIN Suppliers ;
      ON First.SupplierID = Suppliers.SupplierID ;
    JOIN Categories ;
      ON First.CategoryID = Categories.CategoryID ;
  ORDER BY CompanyName, CategoryName ;
  INTO CURSOR csrSameSupplierAndCategory
```

The join condition between the two instances of the Products table (using local aliases First and Second) is a little unusual. The first two parts check for matching values of SupplierID and CategoryID, which makes sense to find products in the same category from the same supplier. The third part of the join condition (First.ProductID < Second.ProductID) ensures that a product doesn't get matched with itself and that each pair is listed only once.

One of the most common places to use a self-join is when a single table holds hierarchical information, such as components for a bill of materials. In Northwind, the Employees table contains a ReportsTo field, indicating the employee's supervisor. That field is a foreign key to the Employees table. To get a list of employees and their immediate supervisors, you need a self-join, as in Listing 15.

Listing 15. Self-joins are useful when a single table contains hierarchical data.

```
SELECT Emp.FirstName AS EmpFirst, Emp.LastName AS EmpLast, ;
       Sup.FirstName AS SupFirst, Sup.LastName AS SupLast ;
   FROM Employees Emp ;
     LEFT JOIN Employees Sup ;
       ON Emp.ReportsTo = Sup.EmployeeID ;
   INTO CURSOR csrSupervisors
```

Cartesian joins

There's one additional kind of join, a *Cartesian* or *cross* join. In this type of join, you don't specify a join condition, so every record from the first table is matched with every record from the second table. Most of the time, you want to avoid Cartesian joins. But there are a few situations in which they provide a way to do something you couldn’t otherwise. There's an example later in this document.

Using joins in other SQL commands

In VFP 9, the SQL UPDATE and DELETE commands support joins. For UPDATE, a join clause can specify both which records to update and what the new values are. For DELETE, a join clause helps indicate which records to mark for deletion. (Note that the use of joins in UPDATE and DELETE is a VFP extension to the SQL standard.)

Specifying updates with joins

VFP 9's UPDATE command accepts a FROM clause that lets you determine which records to update and specify the new field values using fields from other tables. The syntax for this form of UPDATE, known as a *correlated update*, is shown in **Listing 16**.

Listing 16. VFP 9 lets you use a FROM clause in UPDATE to draw values from other tables and determine which records to update.

```
UPDATE Table | Alias
       SET FieldName1 = uExpr1
          [, FieldName2 = uExpr2 [, ... ] ]
       FROM [FORCE] TableName1
         [ JOIN TableName2 … ON JoinExpression ]
       [ WHERE lFilterCondition ]
```

You can list the table being updated in the FROM clause and use ON to specify the conditions that join it to another table, or you can omit the table the FROM clause and use the WHERE clause to specify the join conditions. The rules for joins are the same in UPDATE as in queries, including the ability to specify outer joins, and the choice of nested or sequential style for multiple joins.

If the update table is listed in the FROM clause, you use its local alias following the UPDATE keyword to indicate which table is to be updated.

The commands in **Listing 17** and **Listing 18** demonstrate the two approaches to joining the tables. In each case, the unit price in the Products table is updated based on data in a cursor.

Listing 17. The new FROM clause in UPDATE lets you both draw replacement data from additional tables and specify which records are updated. Here, the table to be updated is not included in the FROM clause, so the join condition appears in the WHERE clause.

```
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 = NewPrices.yPrice ;
   FROM NewPrices ;
   WHERE Products.ProductID = NewPrices.iProductID
```

Listing 18. In this UPDATE command, the table to be updated is listed in the FROM clause and ON is used to join the tables.

```
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 = NewPrices.yPrice ;
   FROM Products ;
      JOIN NewPrices ;
         ON Products.ProductID = NewPrices.iProductID
```

Keep in mind that the table following the UPDATE keyword is really an alias. So the code in **Listing 19** has the same effect as that in **Listing 17** and **Listing 18**.

Listing 19. Use the local alias of the table following the UPDATE command.

```
UPDATE SomeProducts ;
   SET UnitPrice = NewPrices.yPrice ;
   FROM Products AS SomeProducts;
      JOIN NewPrices ;
         ON SomeProducts.ProductID = NewPrices.iProductID
```

Correlated deletion

Along with correlated updates, VFP 9 introduced *correlated deletes*, which let you specify which records are to be deleted by referring to other tables. When more than one table is involved in DELETE, you have to specify the deletion table, the table from which records are being deleted. As with UPDATE, you choose whether to include the deletion table in the FROM clause as well and use a JOIN condition or omit it and join it in the WHERE clause. The syntax for correlated deletes is shown in Listing 20.

Listing 20. Adding the FROM clause to DELETE lets you remove records based on data in other tables.

```
DELETE [ DeletionAlias ]
     FROM Table1 [ JOIN Table2 … ON lJoinCondition ]
     [ WHERE lCondition ]
```

When the deletion table is included in the FROM clause, use its local alias following the DELETE keyword.

Listing 21 shows a simple correlated DELETE command; it deletes all products that come from suppliers in Australia. (You might do this if you have problems getting Australian products delivered.) Note that this command will not execute against the Northwind database because of referential integrity rules. (In the downloads for this session, the programs for this example and the next create cursors on which to operate.)

Listing 21. The ability to include joins in DELETE lets you decide which records to delete based on data in other tables.

```
DELETE Products ;
   FROM Products ;
     JOIN Suppliers ;
       ON Products.SupplierID = Suppliers.SupplierID ;
     WHERE UPPER(Suppliers.Country) = "AUSTRALIA"
```

Listing 21 includes the Products table in the FROM clause. Listing 22 gives the same results, but Products isn't included in the FROM clause, so the join is performed in the WHERE clause.

Listing 22. You can list the deletion table only following the DELETE keyword and use the WHERE clause to join tables.

```
DELETE Products ;
   FROM Suppliers ;
       WHERE Products.SupplierID = Suppliers.SupplierID ;
         AND UPPER(Suppliers.Country) = "AUSTRALIA"
```

While joins give you a lot of power, there are other relationships between tables that you may want to use in SQL commands. The next section looks at subqueries and the many ways they let you specify what data to extract or operate in.

Working with subqueries

A *subquery* is a query that appears within another SQL command. SELECT, DELETE, and UPDATE all 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*.

A subquery is a complete query, but cannot specify a destination using TO or INTO. Subqueries are enclosed in parentheses in the containing query. Subqueries can appear in the WHERE clause of SELECT, UPDATE and DELETE. 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. I'll refer to the command that includes a subquery as the *containing* command.

In VFP 8 and earlier, a subquery cannot contain another subquery. In VFP 9, subqueries can be nested. VFP 8 and earlier also 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 basic 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 where there's 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 with 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. The subquery must include only a single field.
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. The subquery must include only a single field.
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. The subquery must include only a single field.

One well-known use for a subquery is to let you find all the records in one list that aren't in another list. For example, the query in Listing 23 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 23. The subquery here gets 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 csrNoOrders
```

With UPDATE and DELETE, IN lets you act on all the records selected by a subquery. For example, the code in Listing 24 increases prices by 10% for products that come from suppliers in Japan, something you might do if new export fees were introduced.

Listing 24. 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")
```

You can combine queries with the other commands to get the desired results. The code in Listing 25 finds contact items that are 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 25. 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, so we'll return to this example later.

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, suppose you want to find the earliest order from each customer in a particular time period, and return specific information about that order. The query in Listing 26 shows one way to do that. 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 date for that customer.

Listing 26. The correlated subquery here lets you find the record that matches a particular record chosen by aggregation (the MIN() function here).

```
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 csrFirstOrderInMonth
```

Listing 27 combines a correlated subquery with the EXISTS operator to get a list of all the customers who have placed "suspicious" orders. A suspicious order is defined as one where the shipping address doesn't match the address on file for the Customer and the total amount of the order is more than $4,000. Here, the subquery 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 have placed orders deemed suspicious.

Listing 27. 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 csrSuspiciousCustomers
```

The query in Listing 27 works only in VFP 9. In VFP 8 and earlier, a correlated subquery could not use the GROUP BY clause.

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 28 creates a list of customers who never have 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 28. 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 csrDontShipHere
```

Of course, this query can be performed using NOT IN rather than <> ALL.

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.”

To date, I’ve never used either of these operators in an application nor seen them in any code I’ve been called on to modify.

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.) 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. In addition, VFP closes the cursors that are created as derived tables, once the containing command is complete.

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.

One place a derived table is useful is when extracting additional child data after grouping. In many cases, the grouping can be performed in a derived table. Listing 29 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; this problem was first addressed in Listing 26.

Listing 29. 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 csrFirstOrderInMonth
```

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 30 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 30. 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 csrMoreThanAYear
```

A derived table offers another approach to the problem in Listing 25, 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 31. The DELETE command has a subquery. That subquery simply extracts the iItemFK 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. The innermost subquery creates a list of contact items that do not have an expiration date. The derived table Expired uses that result to build a list of contact items where all uses have expired and the most recent expiration date has already passed (dLastDate < DATE()), that is, the same as the cursor Expired in Listing 25. The main subquery pulls just the contact item's key from the derived table, and the DELETE deletes all contact items in that list.

Listing 31. 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 25 is that the query followed by DELETE version leaves the cursor Expired open; the derived table version does not.

Listing 32 demonstrates not only a derived table, but inserting from a UNIONed query and a legitimate use for a Cartesian join (where every record in one table is matched with every record in the second table). This query populates a data warehouse that tracks product sales and total sales by employee. The data is added annually. (In a production situation, the warehouse, which is created here using CREATE CURSOR, would already exist as a table.)

Listing 32. In this example, sales data by product and employee is assembled and added to a data warehouse.

```
nYear = 1997

CREATE CURSOR Warehouse ;
  (ProductID C(6), EmployeeID C(6), ;
   Year N(4), ;
   TotalSales Y, UnitsSold N(8))

INSERT INTO Warehouse ;
SELECT CrossProd.ProductID, ;
       CrossProd.EmployeeID, ;
       m.nYear as Year, ;
       NVL(UnitsSold, 0), NVL(TotalSales, $0);
   FROM (SELECT Employees.EmployeeID, Products.ProductID ;
      FROM Employees, Products) AS CrossProd ;
     LEFT JOIN ( ;
      SELECT ProductID, EmployeeID, ;
             SUM(Quantity) AS UnitsSold, ;
             SUM(Quantity * UnitPrice) AS TotalSales ;
         FROM Orders ;
           JOIN OrderDetails ;
             ON Orders.OrderID = OrderDetails.OrderID ;
         WHERE YEAR(OrderDate) = m.nYear ;
         GROUP BY ProductID, EmployeeID ) AS AnnualSales ;
      ON CrossProd.EmployeeID = AnnualSales.EmployeeID  ;
      AND CrossProd.ProductID = AnnualSales.ProductID ;
```

```
UNION ;
SELECT 0 AS ProductID, Employees.EmployeeID, ;
       m.nYear AS Year, ;
       CAST(NVL(SUM(Quantity),0) as N(12)) AS UnitsSold, ;
       NVL(SUM(Quantity * UnitPrice), $0) AS TotalSales ;
   FROM Orders ;
     JOIN OrderDetails ;
       ON Orders.OrderID = OrderDetails.OrderID ;
      AND YEAR(OrderDate) = m.nYear ;
     RIGHT JOIN Employees ;
       ON Orders.EmployeeID = Employees.EmployeeID ;
   GROUP BY Employees.EmployeeID ;
ORDER BY 2, 1
```

The second query in the UNION is straightforward; it computes the total sales for each employee for the specified year. It includes an outer join to ensure that every employee is listed and uses CAST() and NVL() to get the computed results into the right format.

The first query is more complicated. We want sales of each product by each employee, but there's no relationship between products and employees, except through sales. The query creates a derived table that uses a Cartesian join to ensure that every combination of product and employee is included. That table is then outer joined to the actual sale results to produce an intermediate result that includes one record for each product-employee combination, with sales totals where appropriate.

Like SELECT and DELETE, the FROM clause of UPDATE supports derived tables, so you can assemble the data you need for an update without using multiple commands.

For example, consider a small data warehouse (different from the one in Listing 32) designed to contain only the previous month's sales data by product. It contains one record for each product, listing the product id and two additional fields: total sales and units sold of that product for a month. At the end of each month, you need to update the table with the prior month's data. Listing 33 uses a derived table to compute the monthly total. UPDATE joins the derived table to the data warehouse to update the warehouse. This code assumes that the data warehouse already exists. (The version of this example in the downloads for this session creates the warehouse first.)

Listing 33. You can use a derived table in UPDATE to compute the replacement data on the fly.

```
nMonth=3
nYear=1997

UPDATE SalesByProduct ;
   SET SalesByProduct.TotalSales = NVL(MonthlySales.TotalSales, $0), ;
      SalesByProduct.UnitsSold = NVL(MonthlySales.UnitsSold, 0) ;
   FROM SalesByProduct ;
     LEFT JOIN (SELECT OrderDetails.ProductID, ;
          SUM(Quantity*OrderDetails.UnitPrice) AS TotalSales, ;
          SUM(Quantity) AS UnitsSold ;
      FROM OrderDetails ;
```

```
        JOIN Orders ;
          ON OrderDetails.OrderID = Orders.OrderID ;
          AND (MONTH(OrderDate) = nMonth AND YEAR(OrderDate) = nYear) ;
      GROUP BY 1);
   AS MonthlySales ;
   ON SalesByProduct.ProductID = MonthlySales.ProductID
```

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 one 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 34 shows one way 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, other solutions may be easier to maintain and run 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 34. 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 csrCustomerMonthly
```

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 35 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 35. 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)
```

Improving Query Performance

With all the tools that joins and subqueries give you, you can accomplish an incredible amount using SQL commands. But that's only helpful if those commands operate efficiently.

There are a number of things you can do to optimize the performance of your SQL commands. (Many of them will improve the speed of your Xbase code as well.) In addition, VFP provides a way to see what part of a query needs improvement.

Having the right tags

VFP's Rushmore technology optimizes commands, including SQL commands, using indexes. When a command filters on a condition and there's an index for that condition, Rushmore uses the index to find the matching records rather than searching sequentially through the table. In almost every case, reading the index is faster than reading the actual records.

For Rushmore to use an index, the index expression must exactly match the expression in the command. For example, using the Northwind Customers table, the query in Listing 36 is not fully optimized because the index based on the City field uses the key expression UPPER(City). But the version of the query in Listing 37 is optimized.

Listing 36. When the expression you're matching in a WHERE clause isn't identical to the key expression for an index, that filter condition can't be optimized.

```
SELECT CustomerID, CompanyName ;
   FROM Customers ;
   WHERE City="London" ;
   INTO CURSOR csrLondoners
```

Listing 37. Use the exact index key to get Rushmore optimization.

```
SELECT CustomerID, CompanyName ;
   FROM Customers ;
   WHERE UPPER(City)="LONDON" ;
   INTO CURSOR csrLondoners
```

Because the Customer table is quite small, the actual performance difference between the two versions is also small, but for large tables, matching the index can have a significant impact on performance.

Even if an index is based on several fields, you need to use the exact expression in the command. For example, using the Contacts database's Person table, the query in Listing 38 is not optimized because the Name index is based on the expression UPPER(cLast + cFirst). The query in Listing 39 is optimized.

Listing 38. This query can't be fully optimized because the filter condition doesn't match the index.

```
SELECT cFirst - (" " + cLast) AS FullName ;
   FROM Person ;
   WHERE UPPER(cLast) = "N" ;
   INTO CURSOR csrNamesWithN
```

Listing 39. Even if you only want to match part of a combined index, use the entire index expression for optimization.

```
SELECT cFirst - (" " + cLast) AS FullName ;
   FROM Person ;
   WHERE UPPER(cLast + cFirst) = "N" ;
   INTO CURSOR csrNamesWithN
```

In my tests, the second query runs in about one-third the time of the first. (The downloads for this session includes code to time the queries in Listing 36 through Listing 39.)

Rushmore cannot use any indexes that are filtered, that is, created with a FOR clause in the INDEX command. It also cannot use any indexes that include the NOT operator; instead, index on the expression without NOT and apply NOT in the filter condition.

Expressions involving BETWEEN() and INLIST() are optimized if the first parameter can be optimized; the same is true for the corresponding SQL operators. However, expressions involving ISBLANK() and EMPTY() cannot be optimized, regardless of the parameter.

As with VFP's Xbase commands, in a filter expression the optimizable expression (the one that exactly matches an index) must be on the left-hand side of the comparison. However, for join conditions in the FROM clause, VFP looks at both sides of the expression and chooses which index to use.

When the WHERE clause contains multiple conditions combined with AND and OR, each condition is considered separately. Those that can be optimized are; the remaining conditions are checked against the records that match the optimizable conditions. VFP 9 includes improvements in optimization in queries involving complex conditions using OR.

In Listing 40, only those records where the City field contains “London” are read into memory. Then, each is checked to see that the Country field contains “UK” and those from other countries are eliminated. The number of records to check this way is quite small and should have only a minimal impact on performance.

Listing 40. VFP optimizes what it can, and then checks the remaining conditions sequentially.

```
SELECT CompanyName ;
   FROM Customers ;
   WHERE Country = "UK" ;
     AND UPPER(City) = "LONDON" ;
   INTO CURSOR csrLondonEngland
```

VFP’s optimization trick

In one situation, VFP’s goal of giving you results as fast as possible can be a problem. When a query uses a single table, has only fields from that table in the field list (and no expressions or literals), puts the results into a cursor, and can be fully optimized, VFP filters the source table rather than creating a new file on disk. In some cases, such as reporting, this isn’t a problem. But if you want to use that cursor in a subsequent query or display it in a grid, this trick causes problems.

You can check whether VFP has taken this route by examining DBF() for the resulting cursor. The code in Listing 41 demonstrates, using the queries from Listing 36 and Listing 37. If you see a temporary file for both queries, SET DELETED OFF and try again. (See the next section, “Deletion and Optimization” for an explanation.)

Listing 41. Some very simple queries based on a single table filter the original table rather than creating a new disk presence.

```
OPEN DATABASE HOME(2) + "Northwind\Northwind"

SELECT CustomerID, CompanyName ;
   FROM Customers ;
   WHERE UPPER(City)="LONDON" ;
   INTO CURSOR csrLondoners
?'With fully optimized query, DBF("Londoners") = ', DBF("Londoners")

SELECT CustomerID, CompanyName ;
   FROM Customers ;
   WHERE City="London" ;
   INTO CURSOR csrLondoners
?'With unoptimized query, DBF("Londoners") = ', DBF("Londoners")
```

In FoxPro 2.x and early versions of VFP, the only ways to work around VFP's good intentions were to force the query to be not fully optimized or to add an expression to the field list. Fortunately, the VFP team realized that there are legitimate reasons for preventing this trick. In VFP 5, the NOFILTER keyword was added; use it after INTO CURSOR to tell VFP to create a new file, no matter what, as in Listing 42. In addition, the READWRITE keyword added in VFP 6 has the same effect; in order to ensure that the cursor can be modified, VFP has to create a new file.

Listing 42. Add the NOFILTER clause to force VFP to create a "real" cursor.

```
SELECT CustomerID, CompanyName ;
   FROM Customers ;
   WHERE UPPER(City)="LONDON" ;
   INTO CURSOR Londoners NOFILTER
```

Deletion and optimization

VFP uses a two-step deletion mechanism. When you delete a record, whether you use the Xbase DELETE command, the SQL DELETE command or the deletion column in a grid or Browse, the record is marked as deleted. It's not physically removed from the table until you pack the table using the PACK command.

Because of this structure, VFP includes the SET DELETED command that determines whether other commands see records marked for deletion. SET DELETED OFF to include deleted records in processing; SET DELETED ON to omit them.

Having SET DELETED ON is equivalent to filtering every command on the expression NOT DELETED(). For many years, therefore, the standard advice for those operating with DELETED ON was to create an index for each table based on the DELETED() function, so that the implied filter could be optimized.

However, it turns out that, while an index on DELETED() allows commands to be fully optimized, such commands can be slower than they would be without that index. (The article that explained this paradox is by Chris Probst and appeared in FoxPro Advisor in May, 1999; it was updated and reprinted in March 2005.)

The issue is that there are only two possible values for this index and, most often, the vast majority of records have the same value (False). When Rushmore attempts to optimize the implied expression NOT DELETED(), it must read the whole portion of the index that corresponds to False. Especially in a network situation, loading that portion of the index into memory can take much longer than simply checking each record that meets all the other criteria to see whether it's deleted. (Keep in mind that filters that can't be optimized are checked after all the optimizable filters are applied, so in many cases, the number of records to check "manually" for deletion is quite small.) As a result, for a large table and a small result set, you're usually better off without an index based on the DELETED() function; the meaning of "large" and "small" depends on the amount of memory available.

VFP 9 changes the rules a little. The new binary index type is designed to create extremely small indexes for expressions with only two possible values. For example, on a test table with 1,000,000 records, a regular index based on DELETED() resulted in an index file of 3,205,632 bytes. Using a binary index, instead, the file size was 135,168 bytes. Clearly, reading the binary index is less likely to slow a command down.

For small tables where an index on DELETED() was already useful in optimization, a binary index speeds things up even more. Binary indexes also raise the threshold between tables where an index helps and those where an index hurts.

Binary indexes are only for optimization and can't be used for searching

Tuning memory

The Visual FoxPro engine knows how to use memory extremely efficiently, caching data to avoid time-consuming disk access. However, there's one situation where this ability can result in slow code; that's the case where VFP thinks it has memory, but is actually using page files on disk.

When you start VFP, it figures out how much memory to use; normally it takes about half of the physical memory of the computer. However, with other applications (like Windows itself) running, there's a good chance that amount of physical memory isn't actually available, and that behind the scenes Windows is swapping out to disk. But VFP doesn't know it, so it makes decisions that assume all the memory it's using is physical memory.

Fortunately, you can control the amount of memory VFP thinks it has. The function SYS(3050) lets you set both the foreground and background memory available to VFP. The syntax is shown in Listing 43.

Listing 43. You can control how much memory VFP uses. Sometimes, giving it less memory than the default speeds up execution.

```
cMemoryInBytes = SYS(3050, nType [, nMemoryInBytes ] )
```

Pass 1 for nType to set the amount of memory available when VFP is in control (in the foreground) and 2 to set the memory available to VFP when another application is in the foreground. VFP rounds the value you pass for nMemoryInBytes down to the nearest 256K and allocates that much memory for itself. The function returns that value as a character string. Note that the value you pass is in bytes, not KB or GB.

VFP expert Mac Rubel did extensive testing of the effects of SYS(3050) with VFP 6 and VFP 7 and discovered that most often, you want the foreground setting to be less than the default. You need to test in your production environment to find the right setting, but a good place to start is around one-third of physical memory.

Make code pages match

In VFP 9, if the current code page (as indicated by CPCURRENT()) is different from the code page of a table (indicated by CPDBF()), operations involving character expressions from that table cannot use existing indexes for optimization (though VFP may still build temporary indexes). While this may seem arbitrary, it's one of a number of changes in VFP 8 and VFP 9 designed to prevent inaccurate query results.

An index is sorted according to the rules for the table's code page. Even if VFP translated from the table's code page to the current code page, the sort order might be different. This means comparisons to data using the current code page may be incorrect. This is what VFP's new rules prevent.

Make collation sequences match

Like code pages, collation sequences aid in working with languages other than English. A collation sequence indicates the sorting order of the characters.

Every index is created with an associated collation sequence. For optimization, VFP can only use indexes created with the collation sequence in effect when the command executes. In other words, for VFP to take advantage of a tag, IDXCOLLATE() for that tag must match SET(“COLLATE”). This is true not only in VFP 9, but in earlier versions as well.

WHERE vs. HAVING

SQL-SELECT has two clauses that filter data: WHERE and HAVING. A good grasp of the English language might lead us to believe that these are synonyms, but SQL is not English, and confusing these two indiscriminately is a mistake. It's not obvious where a particular condition should go at first glance. But getting it wrong can lead to a significant slowdown.

Here's why. The conditions in WHERE filter the original data. Existing index tags are used to speed things up as much as possible. This produces an intermediate set of results. HAVING operates on the intermediate results, with no tags in sight. So, by definition, HAVING is slower than WHERE, if a query is otherwise constructed to be optimized.

The query in Listing 39, which finds people whose last name begins with "N" in the Contacts database, and uses a filter in WHERE, is optimizable. However, if we move the filter to the HAVING clause, as in Listing 44, the query can't be fully optimized.

Listing 44. Filters that appear in the HAVING clause operate on intermediate results and can't be optimized.

```
SELECT cFirst - (" " + cLast) AS FullName ;
   FROM Person ;
   HAVING UPPER(cLast + cFirst) = "N" ;
   INTO CURSOR csrNamesWithN
```

When should you use HAVING? When you group data with GROUP BY and want to filter on aggregate data formed as a result of the grouping rather than on the raw data. For example, if you group customers by country, counting the number in each, and you're interested only in countries with three or more customers, put the condition COUNT(*) >= 3 in the HAVING clause, as in

Listing 45. Use the HAVING clause to filter data after aggregating it.

```
SELECT Country, COUNT(*) ;
   FROM Customers ;
   GROUP BY Country ;
   HAVING COUNT(*) > 3 ;
   INTO CURSOR csrCountriesThreeOrMore
```

There's a simple rule of thumb: Don't use HAVING unless you also have a GROUP BY. That doesn't cover all the cases, but it eliminates many mistakes. To make the rule complete, remember that a condition in HAVING should contain one of the aggregate functions (COUNT, SUM, AVG, MAX or MIN) or a field that was named with AS and uses an aggregate function.

Testing Optimization

Visual FoxPro makes it incredibly easy to figure out how it's optimizing SQL commands. Two functions give you access to what’s called a "SQL ShowPlan," basically a listing of the optimization steps taken.

The first function, SYS(3054), has been in the language since VFP 5 and has been enhanced several times since. The second function, SYS(3092), was added in VFP 9 to make working with SYS(3054) easier.

SYS(3054) controls the display of the SQL ShowPlan. When you turn it on and execute SELECT, UPDATE or DELETE, VFP displays the optimization choices it makes. The syntax for SYS(3054) is shown in Listing 46.

Listing 46. The function SYS(3054) turns SQL ShowPlan on and off. Use it to see how your SQL commands are being optimized.

```
cSetting = SYS(3054 [, nSetting [, cOutputVar ] ])
```

The key parameter is nSetting, which lets you specify what to display. The acceptable values are shown in Table 2. Pass 1, 2, 11 or 12 to turn on display of the optimization plan, then execute the commands you're checking. When you're done, call the function again, passing 0 for nSetting.

Table 2. SYS(3054) lets you turn SQL ShowPlan on and off. You also determine whether the command itself is included in the output.

Value	Meaning
0	Turn off SQL ShowPlan
1	Turn on SQL ShowPlan for filters only
2	Turn on SQL ShowPlan for filters only and include the command in the output
11	Turn on SQL ShowPlan for filters and joins
12	Turn on SQL ShowPlan for filters and joins and include the command in the output

Pass 1 or 2 for nSetting to see optimization of filter conditions only. For example, the code in Listing 47 produces the output in Listing 48. Based on the value of 1 for nSetting, the output looks only at the optimization of filter conditions and doesn't consider how tables were joined.

Listing 47. Use SYS(3054) to see how the VFP engine optimizes a query or other SQL command.

```
SYS(3054,1)
SELECT OrderID, OrderDate, ;
       Customers.CompanyName AS Customer, ;
       Employees.LastName, Employees.FirstName, ;
       Shippers.CompanyName AS Shipper ;
   FROM Orders ;
     JOIN Customers ;
       ON Orders.CustomerID = Customers.CustomerID ;
     JOIN Employees ;
       ON Orders.EmployeeID = Employees.EmployeeID ;
     JOIN Shippers;
       ON Orders.ShipVia = Shippers.ShipperID ;
   WHERE BETWEEN(OrderDate, {^ 1997-2-1}, {^ 1997-2-28}) ;
   ORDER BY OrderDate DESC, LastName ;
   INTO CURSOR csrOrderInfo
SYS(3054, 0)
```

Listing 48. When you pass 1 to SYS(3054), only the optimization of filter conditions is reported.

```
Using index tag Orderdate to rushmore optimize table orders
Rushmore optimization level for table orders: full
Rushmore optimization level for table customers: none
Rushmore optimization level for table employees: none
Rushmore optimization level for table shippers: none
```

At first glance, the information in Listing 48 might seem alarming, as it shows no optimization for three of the four tables in the query. However, a look at the query shows that only the Orders table is filtered, so no optimization for the others is acceptable.

"Full" and "none" aren't the only possible results. The optimization level for a table can also be "partial"; this occurs when there's more than one filter for a table, at least one of the filters can be optimized and at least one cannot.

In the query in Listing 40, Customers is filtered on both UPPER(City) and Country; there's an index for UPPER(City), but not for Country. Listing 49 shows the SYS(3054) output, which indicates partial optimization.

Listing 49. When a table is filtered on more than condition and only some of the conditions can be optimized, the results show "partial."

```
Using index tag City to rushmore optimize table customers
Rushmore optimization level for table customers: partial
```

One case where you'll often see "partial" is with SET DELETED ON. Unless you have an index on DELETED(), every table with an optimizable filter condition will show "partial." See "Deletion and Optimization" earlier in this document for the reasons why and why not to have such an index.

When you pass 2 to SYS(3054), you get the same information, but the query itself is included in the listing. Listing 50 shows the output for the query in Listing 40, after calling SYS(3054,2).

Listing 50. Pass 2 or 12 for nSetting to see the query as well as the optimization information.

```
SELECT CompanyName FROM Customers WHERE Country = "UK" AND UPPER(City) = "LONDON"
INTO CURSOR csrLondonEngland
Using index tag City to rushmore optimize table customers
Rushmore optimization level for table customers: partial
```

When you pass 11 or 12 for nSetting, the output tells you not only how filters and joins were optimized, but also the order in which joins were performed. The VFP engine often joins tables in a different order than the command specifies, in order to speed up the results.

Listing 51 shows the output for the same query as in Listing 47, but passing 12 for nSetting. Although the query itself in the output, it's no longer neatly formatted.

Listing 51. Pass 11 or 12 to SYS(3054) to find out about both filters and joins.

```
SELECT OrderID, OrderDate, Customers.CompanyName AS Customer, Employees.LastName,
Employees.FirstName, Shippers.CompanyName AS Shipper FROM Orders JOIN Customers ON
Orders.CustomerID = Customers.CustomerID JOIN Employees ON Orders.EmployeeID =
Employees.EmployeeID JOIN Shippers ON Orders.ShipVia = Shippers.ShipperID WHERE
BETWEEN(OrderDate, {^ 1997-2-1}, {^ 1997-2-28}) ORDER BY OrderDate DESC, LastName
INTO CURSOR OrderInfo
Using index tag Orderdate to rushmore optimize table orders
Rushmore optimization level for table orders: full
Rushmore optimization level for table customers: none
Rushmore optimization level for table employees: none
Rushmore optimization level for table shippers: none
Joining table employees and table orders using index tag Employeeid
Joining table shippers and intermediate result using temp index
Joining intermediate result and table customers using index tag Customerid
```

In this example, where all joins are inner joins, all the filtering is done first and then tables are joined. The logical order of joins, specified in the query, is Orders to Customers, then that result to Employees, and finally that result to Shippers. But the SYS(3054) output tells us that VFP first joined Employees and Orders, then joined that result to Shippers and finally joined that result with Customers.

For joins, SYS(3054) lists the table whose index was used for optimization second. So, in the example, the join between Employees and Orders was optimized using the EmployeeID index of Orders (which makes sense, as it's a much bigger table than Employees). For the join between the initial result and Shippers, the VFP engine decided that no existing index would be useful and created an index on the fly (listed as "temp index"). Again, that makes sense, because Shippers is a tiny table (with only three records), so none of its indexes would help speed things up and, of course, the intermediate result from the first join doesn't have any indexes. Nonetheless, for the final join, an existing index on the Customers table was seen as offering more help than creating an index on the intermediate result.

In addition to using an existing index or creating an index on the fly, the output can indicate that no optimization was possible because a Cartesian join occurred. When you see that result, it's almost always a sign that something is wrong. Listing 52 shows SYS(3054) output for the command in Listing 32, which is one those rare cases where a Cartesian join was intended. (The command puts sales data into a data warehouse.) Normally, if you see “(Cartesian product)” in SYS(3054) output, check the command for a missing join condition.

Listing 52. When SQL ShowPlan output indicates that a Cartesian join was performed, make sure you've specified appropriate join conditions.

```
Rushmore optimization level for table employees: none
Rushmore optimization level for table products: none
```

```
Joining table employees and table products (Cartesian product)
Rushmore optimization level for table orders: none
Rushmore optimization level for table orderdetails: none
Joining table orders and table orderdetails using index tag Orderid
Rushmore optimization level for intermediate result: none
Rushmore optimization level for intermediate result: none
Joining intermediate result and intermediate result using temp index
Rushmore optimization level for table orders: none
Rushmore optimization level for table orderdetails: none
Joining table orders and table orderdetails using index tag Orderid
Rushmore optimization level for intermediate result: none
Rushmore optimization level for table employees: none
Joining table employees and intermediate result using temp index
```

Listing 52 also shows what happens for a query that includes outer joins. Instead of doing all the filtering and then all the joins, the two are interspersed.

Managing showplan output

By default, SYS(3054) sends its output to the active window. In VFP 7 and later, you can capture the output to a variable instead. Pass the name of the variable as the third parameter. (Note that you must pass the name, not the variable itself; that's because there's no way to pass parameters by reference to VFP's built-in functions.) Listing 53 demonstrates; after running a query, the variable cOptInfo contains the optimization information.

Listing 53. When you pass the name of a variable as the third parameter to SYS(3054), the optimization information is stored in that variable.

```
SYS(3054, 12, "cOptInfo")
```

The variable can hold the results for only a single SQL command. That is, you pass a variable name to SYS(3054) and run a SQL command. If you then run another SQL command, the variable is cleared and only the results from the second command are saved. This behavior makes it very difficult to take advantage of SYS(3054) in an application setting.

In VFP 9, the Fox team introduced a better approach that allows you to track optimization throughout an application. SYS(3092) lets you send optimization information to a file; call it before you set SYS(3054). The syntax for SYS(3092) is shown in Listing 54.

Listing 54. SYS(3092) lets you indicate where to store the optimization information produced by SYS(3054).

```
cLogFile = SYS(3092 [, cFileName [, lAdditive ] ] )
```

Call the function with no additional parameters (just SYS(3092)) to find the name of the currently active log file. When you pass a file name as the second parameter, that file becomes the active log file and the function returns that value. The lAdditive parameter determines whether new data is added to an existing file or the file is deleted first. Once you turn logging on, all SQL ShowPlan results are stored to the specified file.

To turn off the log so you can see its contents, call SYS(3092) again, passing the empty string for the file name.

Listing 55 shows a complete example. SYS(3092) sets up a log file, and then SYS(3054) is called to turn on SQL ShowPlan. A query is executed, and then SQL ShowPlan and the log are turned off.

Listing 55. Combine SYS(3054) with SYS(3092) to let you store optimization results in a file.

```
SYS(3092, "Optim.Log")
SYS(3054, 12, "cDummy")
SELECT CustomerID, ;
       COUNT(DISTINCT OrderDate) AS DatesOrdered, ;
       COUNT(OrderDate) AS TotalOrders ;
   FROM Orders ;
   GROUP BY 1 ;
   INTO CURSOR csrHowManyOrders
SYS(3054, 0)
SYS(3092, "")
```

Turning on logging doesn't keep SYS(3054) from displaying its results in the active window. If you want the output to go only to the log file, pass a variable name to SYS(3054), as in the example.

Logging optimization to a file isn't useful only for tracking multiple SQL commands. It's also handy for figuring out what's going on at a client site. You might set up a hidden mechanism in your application to turn logging of SQL ShowPlan on and off. When a client reports a slowdown, have the client turn logging on, run the troublesome process, turn logging off, and send you the log.

Summary

VFP's SQL sublanguage provides an extremely powerful mechanism for working with data. Getting started with simple SQL commands is easy, but working with multiple tables and subqueries is trickier, though the secret is always to build your commands a little bit at a time until you get what need.

SQL commands can be very fast, if you know how to design them. For problem situations, VFP provides a way to see what's going on under the hood.

Much of the material in this paper is excerpted from my book, *Taming Visual FoxPro's SQL*, Hentzenwerke Publishing, 2005. (More information at http://www.hentzenwerke.com/catalog/tamingvfpsql.htm)