

# Building Queries in Visual FoxPro

*Session GRA127*

*Tamar E. Granor  
Tomorrow's Solutions, LLC  
8201 Cedar Road  
Elkins Park, PA 19027  
Voice: 215-635-1958*

*Email: [tamar@tomorrowssolutionsllc.com](mailto:tamar@tomorrowssolutionsllc.com)  
Web: [www.tomorrowssolutionsllc.com](http://www.tomorrowssolutionsllc.com)*

## **Overview**

Visual FoxPro's SQL-SELECT command provides an easy way to pull data out of one or more tables, but it can be difficult to get exactly the data you want. This session will dig into some queries that are hard to get right the first time, such as those involving "multiple unrelated siblings" and filtered outer joins.

In addition, VFP 8 features major improvements in the Query Designer and View Designer that make it useful for a far wider range of queries than in previous versions. This session will look at those changes, and discuss the remaining limitations.

This session assumes some familiarity with the SQL-SELECT command. It's focused on VFP's native SQL-SELECT command, not client-server queries.

## Using the Query Designer/View Designer

The Query Designer (QD) has been in VFP since FoxPro 2.0; the View Designer (VD) since VFP 3. However, through VFP 7, these tools were useful only for the most basic of queries and views. Fortunately, VFP 8 changes that. There are a number of changes that make the two designers both more capable and easier to use.

The biggest usability change is the addition of two-way editing. The window that appears when you choose the SQL button is now read-write. Changes you make there are reflected in the designer itself. So, when you know how to specify what you want, you can just type it right into the SQL window, and the designer will parse your code and set up the main tool. In addition, in the View Designer, the SQL window now includes all the DBSetProp() calls needed to set up your view.

There are quite a few other changes. Many of them are discussed below, as the relevant topics are covered.

## Getting the right 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. Joining 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. (JOIN was added in VFP 5. Prior to that, joins were placed in the WHERE clause.)

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

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

The expression following the ON clause 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. (Another type of complex condition is discussed in [Combining outer joins with filters](#) below.)

VFP provides two ways to specify joins when more than two tables are involved in a query. The first, or *nested*, style lists all the tables first, then lists all the join conditions. In nested style, 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.

For example, using VFP's TasTrade sample data, the FROM clause of a query involving the Customer, Orders, and Order\_Line\_Items tables might look like this:

```
FROM Customer ;
  JOIN Orders ;
    JOIN Order_Line_Items ;
      ON Orders.Order_Id = Order_Line_Items.Order_Id ;
    ON Customer.Customer_Id = Orders.Customer_Id
```

Orders and Order\_Line\_Items are joined first, using the condition Orders.Order\_Id = Order\_Line\_Items.Order\_Id. Then Customer is joined to the result using the condition Customer.Customer\_Id = Orders.Customer\_Id. Nested style works best when the tables in the query have a hierarchical relationship.

The alternative to nested syntax, the *sequential* style, 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. Here's the same join, using the sequential syntax:

```
FROM Customer ;
  JOIN Orders ;
    ON Customer.Customer_Id = Orders.Customer_Id ;
  JOIN Order_Line_Items ;
    ON Orders.Order_Id = Order_Line_Items.Order_Id
```

This time, Customer and Orders are joined first, using the condition Customer.Customer\_Id = Orders.Customer\_Id. Then Order\_Line\_Items is joined with the result, based on the condition Orders.Order\_Id = Order\_Line\_Items.Order\_Id.

The nested and sequential styles can be combined in a single query. There are some examples below.

## Multiple unrelated siblings

Writing a join condition is usually easy when the relationship among the tables is strictly hierarchical (parent-child-grandchild); either nested or sequential style can be used, as in the examples above. But when a table has multiple child tables and the child tables aren't related to each other, things start to get interesting.

Consider a query in which you want to find out the TasTrade employee who took an order and the shipper used to send the order. (Perhaps you want to determine whether an employee appears to be steering business to a particular shipper.) In this situation, the Orders table is the parent and the Employee and Shippers tables are the children. Your first impulse in writing this query might be:

```
SELECT Shippers.Company_Name, ;
       Orders.Order_Id, ;
       Employee.Last_Name ;
FROM Orders ;
  JOIN Shippers ;
  JOIN Employee ;
    ON Orders.Employee_Id = Employee.Employee_Id ;
  ON Shippers.Shipper_Id = Orders.Shipper_Id
```

Unfortunately, this query doesn't work. When you run it, one of two things happens. If the Orders tables is open, the query runs, but the results contain the same employee in each row rather than the one associated with the order. If the Orders table is closed, the error "SQL: Column 'EMPLOYEE\_ID' is not found." appears. The problem is that VFP attempts to join Shippers and Employee using the innermost condition:

```
Orders.Employee_Id=Employee.Employee_Id
```

When Orders isn't open, that triggers an error. When Orders is open, VFP's behavior is much more dangerous: it grabs the current value of Orders.Employee\_Id and plugs that into the join.

There are several ways to get the right results. You can use nested style by listing the tables in a different order:

```
SELECT Shippers.Company_Name, ;
       Orders.Order_Id, ;
       Employee.Last_Name ;
FROM Shippers ;
JOIN Orders ;
JOIN Employee ;
ON Orders.Employee_Id = Employee.Employee_Id ;
ON Shippers.Shipper_Id = Orders.Shipper_Id
```

In this case, Orders has been moved to the middle of the relationship. While this query gives the right results, to a reader, it may suggest a parent-child-grandchild relationship from Shippers to Orders to Employees. (Technically, it specifies that Shippers is to be joined with the result of joining Orders and Employee.) Similarly, the next version, also using the nested style, suggests a relationship between Shippers and Employee:

```
SELECT Shippers.Company_Name, ;
       Orders.Order_Id, ;
       Employee.Last_Name ;
FROM Shippers ;
JOIN Employee;
JOIN Orders;
ON Orders.Employee_Id = Employee.Employee_Id ;
ON Shippers.Shipper_Id = Orders.Shipper_Id
```

Fortunately, the sequential style makes it easier to deal with non-hierarchical relationships. You can get the right answer and have readable code:

```
SELECT Shippers.Company_Name, ;
       Orders.Order_Id, ;
       Employee.Last_Name ;
FROM Orders;
JOIN Shippers;
ON Shippers.Shipper_Id = Orders.Shipper_Id;
JOIN Employee ;
ON Orders.Employee_Id = Employee.Employee_Id
```

As the number of tables involved in a query increases, the value of having both the nested and sequential styles for joins becomes apparent, especially because you can mix the two

in a single query. This query extracts the customer name, shipper, order date, and each product in an order:

```
SELECT Customer.Company_Name, ;
       Shippers.Company_Name, ;
       Orders.Order_Date, ;
       Products.English_Name ;
FROM Customer ;
   JOIN Orders ;
      JOIN Order_Line_Items ;
         JOIN Products ;
            ON Order_Line_Items.Product_Id = ;
               Products.Product_Id ;
            ON Orders.Order_Id = Order_Line_Items.Order_Id ;
        ON Customer.Customer_Id = Orders.Customer_Id ;
    JOIN Shippers ;
       ON Orders.Shipper_Id = Shippers.Shipper_Id
```

Nested joins are used to move from Customer down to Products, where a hierarchical relationship exists. Then the final, sequential, join tells VFP to join that result with Shippers.

Finally, it's worth mentioning that any comparison operator can be used in the join condition. Because joins so often involve primary keys, most joins in most queries use equality, but there are situations where another operator, such as < or >, is called for.

## Using the Query Designer and View Designer for complex joins

Prior to VFP 8, the Query Designer and View Designer used only the nested style, and weren't very smart about the order in which to join the tables. This sometimes resulted in queries that didn't work right; there were some queries that simply couldn't be written in the QD/VD.

For example, the QD/VD in VFP 7 and earlier is unable to produce a query that can extract the customer name, employee name and shipper for each order in TasTrade, regardless of the order in which tables are added and even if you help it out along the way by specifying appropriate join conditions. Here's the FROM clause it generates if you add the tables Orders, Shippers, Employee and Customer, in that order:

```
FROM tastrade!shippers INNER JOIN tastrade!orders;
   INNER JOIN tastrade!employee;
      INNER JOIN tastrade!customer ;
         ON Customer.customer_id = Orders.customer_id ;
         ON Employee.employee_id = Orders.employee_id ;
         ON Shippers.shipper_id = Orders.shipper_id
```

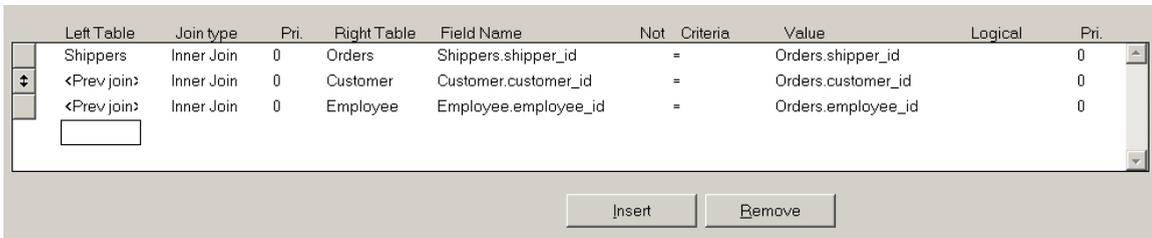
This version, like the three-table nested version above, results in either bad data or an error.

VFP 8's Query Designer lets you choose between the nested style and the sequential style, defaulting to the sequential. You can make the choice for each join in a query, and you can combine the two in a remarkable variety of ways. This enables the Designer to handle a wider range of queries without requiring any tweaking.

In VFP 8, the QD/VD has no problem dealing with the four-table join, involving Orders, Shippers, Employee and Customer. It produces this FROM clause, which is correct:

```
FROM ;
    tastrade!shippers ;
    INNER JOIN tastrade!orders ;
    ON Shippers.shipper_id = Orders.shipper_id ;
    INNER JOIN tastrade!customer ;
    ON Customer.customer_id = Orders.customer_id ;
    INNER JOIN tastrade!employee ;
    ON Employee.employee_id = Orders.employee_id
```

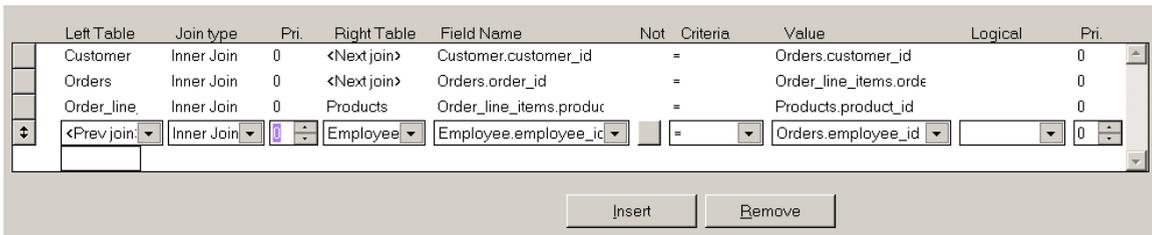
Besides the switch to sequential style, the VFP 8 QD/VD gives you much more ability to tweak the join. The Join page of the Designer indicates which tables are involved in each join, and lets you add parentheses to the joins. Figure 1 shows the Join page for the query above.



**Figure 1 Specifying joins – The VFP 8 Query Designer/View Designer provides increased control over joins.**

Several changes in the QD/VD increase your control over the results. The first is that the actual tables involved are listed for each join. Previously, the Join page showed only the type of join and the join expression. The left-hand table dropdown allows you to specify the previous join, while the right-hand dropdown lets you choose the next join. Choosing "<Next join>" specifies the nested style, while "<Prev join>" uses sequential.

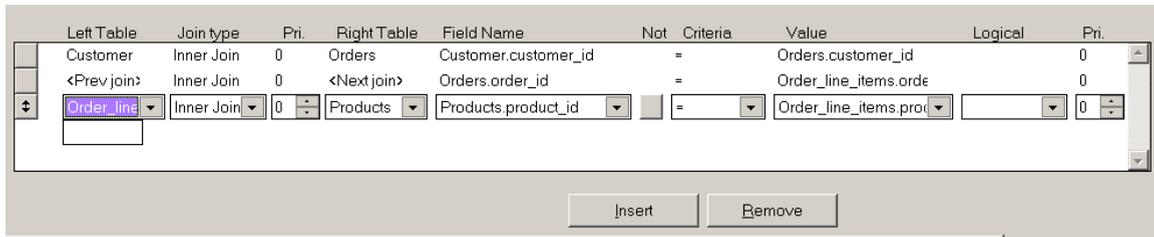
In Figure 1, you can see that the join between Shippers and Orders is to be performed first. Then, that result is joined to Customer and finally, that result is joined to Employee. Figure 2 shows the Join page for the final query in the previous section, the one that provides the customer name, shipper, order date, and each product for all orders. In this case, the first two joins specify "<Next join>" for the right-hand table, resulting in nested style. The final join, to Employee, uses "<Prev join>" for the left-hand table, making that one use the sequential style.



**Figure 2 Mixed joins – You can determine whether the nested style or the sequential style for joins is used, by specifying either "<Next join>" or "<Prev join>."**

It's worth noting that you can even have "<Prev join>" and "<Next join>" in the same row. For example, Figure 3 shows the Join page for one version of a query that collects the customer name, order id and product name for each product ordered. Here's the resulting code:

```
SELECT Customer.company_name, Orders.order_id, Products.product_name;
FROM ;
    tastrade!customer ;
    INNER JOIN tastrade!orders ;
    ON Customer.customer_id = Orders.customer_id ;
    INNER JOIN tastrade!order_line_items ;
    INNER JOIN tastrade!products ;
    ON Products.product_id = Order_line_items.product_id ;
    ON Orders.order_id = Order_line_items.order_id
```



**Figure 3 Monkey in the middle – In this query, the join between Orders and Order\_Line\_Items is deferred until after the joins between Customer and Orders, and between Order\_Line\_Items and Products.**

There's one thing to note about this example. When all the joins in a query are inner joins, the order in which they're performed doesn't affect the results. (That's not true for outer joins – see [Adding outer joins to the mix](#) below.) Thus, while the query specifies a particular order for the joins, in fact, VFP may choose to perform them in a different order for optimization reasons. (See [Improving Query Performance](#) below.)

The left-hand set of priority spinners (labeled "Pri.") let you put sequential style inside nested style. Nested joins with higher priority values are performed later. (The priority setting is ignored for sequential joins.) Figure 4 shows the Join page for a slightly different version of the query from Figure 2. In this case, the priority setting of 1 for the join involving Customer causes it to be performed last. Here's the code (somewhat reformatted from the way it appears in the SQL window):

```
SELECT Customer.company_name, ;
    Employee.last_name, ;
    Orders.order_date, ;
    Products.english_name ;
FROM Customer ;
    JOIN Orders ;
    JOIN Order_line_items ;
    JOIN Products ;
    ON Order_line_items.product_id = Products.product_id ;
    ON Orders.order_id = Order_line_items.order_id ;
JOIN Employee ;
    ON Employee.employee_id = Orders.employee_id ;
```

**ON Customer.customer\_id = Orders.customer\_id**

	Left Table	Join type	Pri.	Right Table	Field Name	Not	Criteria	Value	Logical	Pri.
+	Customer	Inner Join	1	<Next join>	Customer.customer_id		=	Orders.customer_id		0
	Orders	Inner Join	0	<Next join>	Orders.order_id		=	Order_line_items.orde		0
	Order_line	Inner Join	0	Products	Order_line_items.produc		=	Products.product_id		0
	<Prev join>	Inner Join	0	Employee	Employee.employee_id		=	Orders.employee_id		0

**Figure 4 Changing join order – The priority spinners let you put sequential joins inside nested joins.**

As with the original choice between nested and sequential style, as long as the join conditions match up to the right tables, the order specified doesn't matter. Choose the version that's easiest to maintain.

The right-hand set of priority spinners are discussed in [Specifying filtered joins in the Query Designer/View Designer](#) below.

Finally, the Join type dialog in VFP 8 includes a new type of join – the "cross join." This choice allows you to use the older comma-separated list of tables (presumably with join conditions in the WHERE clause), as well as the ANSI JOIN syntax. You're unlikely to use cross joins when writing new queries, but they're important in allowing the QD/VD to parse existing queries.

When a cross join is specified, only the Left Table column is used. Each table listed as cross join is added at the beginning of the FROM clause. If the query uses "\*" to indicate that all fields from all tables should be included in the result, the order of tables in the FROM clause determines the order of fields in the result.

Figure 5 shows the Join page for this query, which joins Customer, Orders and Employee using the WHERE clause:

```

SELECT Customer.company_name, Orders.order_id, Employee.last_name;
FROM ;
    tastrade!customer, ;
    tastrade!orders, ;
    tastrade!employee;
WHERE Customer.customer_id = Orders.Customer_ID;
AND Orders.employee_id = Employee.Employee_ID

```

	Left Table	Join type	Pri.	Right Table	Field Name	Not	Criteria	Value	Logical	Pri.
	Customer	Cross Join	0				=			0
	Orders	Cross Join	0				=			0
+	Employee	Cross Join	0				=			

**Figure 5 Specifying a cross join – Cross joins allow you to separate tables with commas and specify join conditions elsewhere.**

## The Membership data set

While the TasTrade data provided with VFP presents some interesting challenges, such as the multiple unrelated siblings problem, other complex queries are best demonstrated by a different database. Many of the examples below use the membership database described here.

This database (called Membership) represents part of the data for an organization like a YMCA that offers activities for adults, children and families. Typically, organizations like this provide membership and registration discounts for multiple family members, so data needs to be maintained at both the individual and family levels. The key table in this database is Person – each record represents one person:

Field Name	Type	Width	Description
IPERSONID	Integer	4	Primary key
CFIRSTNAME	Character	12	First Name
CLASTNAME	Character	20	Last Name
DBIRTHDATE	Date	8	Birthdate
IEMERGFK	Integer	4	Person key for Emergency Contact
IFAMILYFK	Integer	4	Family key

Person contains two foreign keys. iEmergFK is a pointer back into the Person table for this person's emergency contact. iFamilyFK links the person to his or her family.

The Family table contains one record for each family and tracks address. For simplicity, it handles only US addresses.

Field Name	Type	Width	Description
IFAMILYID	Integer	4	Primary key
MADDRESS	Memo	4	Street address
CCITY	Character	15	City
CSTATE	Character	2	State
CZIP	Character	9	Zip

One of the more challenging aspects of tracking people these days is managing phone numbers. The Membership database uses two tables for the task. Each record in Phone contains information about a single phone number. HasPhone links phones to people, in a many-to-many relationship. Phone has this structure:

Field Name	Type	Width	Description
IPHONEID	Integer	4	Primary Key
CAREACODE	Character	3	Area code
CNUMBER	Character	7	Phone number
CEXTENSION	Character	5	Extension

The link table, HasPhone, has this structure:

Field Name	Type	Width	Description
IHASPHONEID	Integer	4	Primary key
IPERSONFK	Integer	4	Foreign key to Person
IPHONEFK	Integer	4	Foreign key to Phone
CTYPE	Character	1	Type of phone

The last field, cType, contains a one-character code indicating the type of phone: "H" for home, "W" for work, "F" for work fax, "M" for mobile and "X" for home fax. While a production application might link to a look-up table for those values, that approach adds complexity that makes the examples below hard to understand.

The Membership database is included in the conference materials.

## **Table: Join thyself**

Using the same table for a person and his or her emergency contact raises an interesting question. How do you extract that information? The answer is what's known as a *self-join*, in which the same table appears more than once in a query.

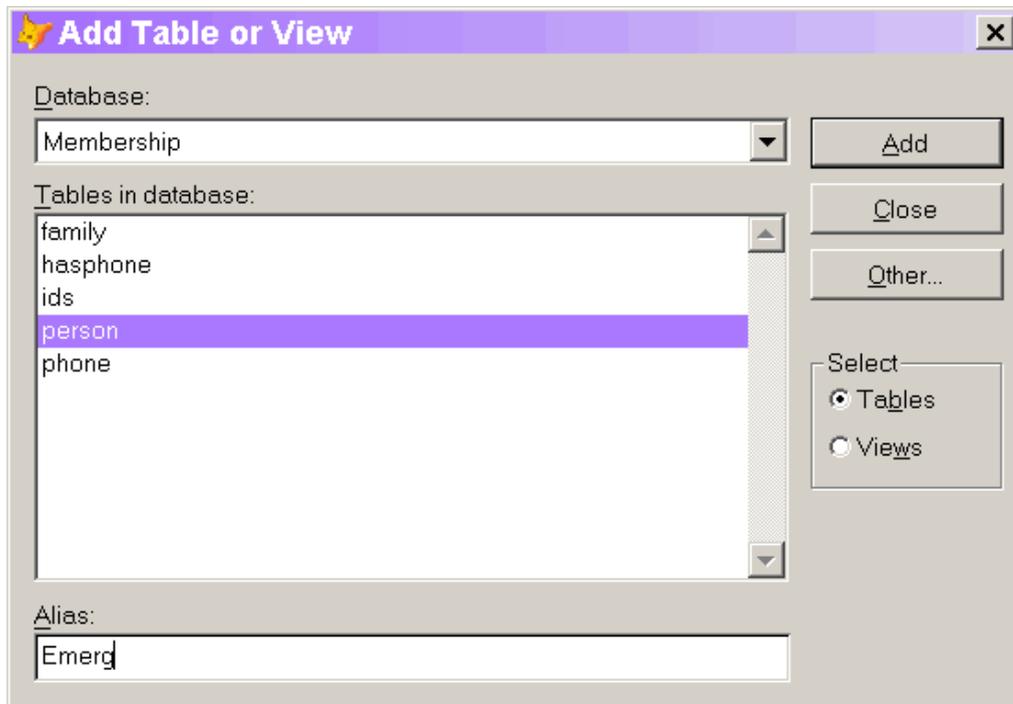
To create a self-join, you open the table more than once and give it a different alias each time. The term *local alias* is used for an alias that's created and used within a query. Here's a FROM clause that assigns the local alias Cust to the Customer table:

```
FROM Customer Cust ;  
  JOIN Orders ;  
    ON Cust.Customer_Id = Orders.Customer_Id
```

As the example suggests, when a table has a local alias, you must use that alias each time you refer to the table; you cannot use the actual table name.

In VFP 7 and earlier, the QD/VD doesn't provide a way to specify the local alias for a table. If you add the same table to a query more than once, the second instance is assigned an alias of the table name plus "\_a". The third instance gets the table name plus "\_b", and so on.

In VFP 8, the Add Table or View dialog lets you specify a local alias for the table. Figure 6 shows this dialog populated with the Membership database's tables, and specifying a local alias of Emerg for the Person table.



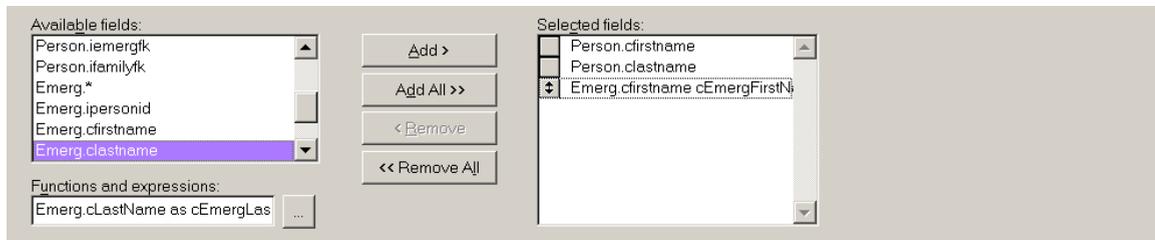
**Figure 6 Specifying a local alias – In VFP 8, the Add Table or View dialog lets you specify a local alias for any table.**

As with any other join, a self-join needs an appropriate condition that indicates which records to match up. Here's a self-join that produces a list of people showing their names and the names of their emergency contacts.

```
SELECT Person.cFirstName, Person.cLastName, ;
       Emerg.cFirstName AS cEmergFirstName, ;
       Emerg.cLastName AS cEmergLastName;
FROM Person ;
     JOIN Person Emerg ;
     ON Person.iEmergFK = Emerg.iPersonId ;
INTO CURSOR Contacts
```

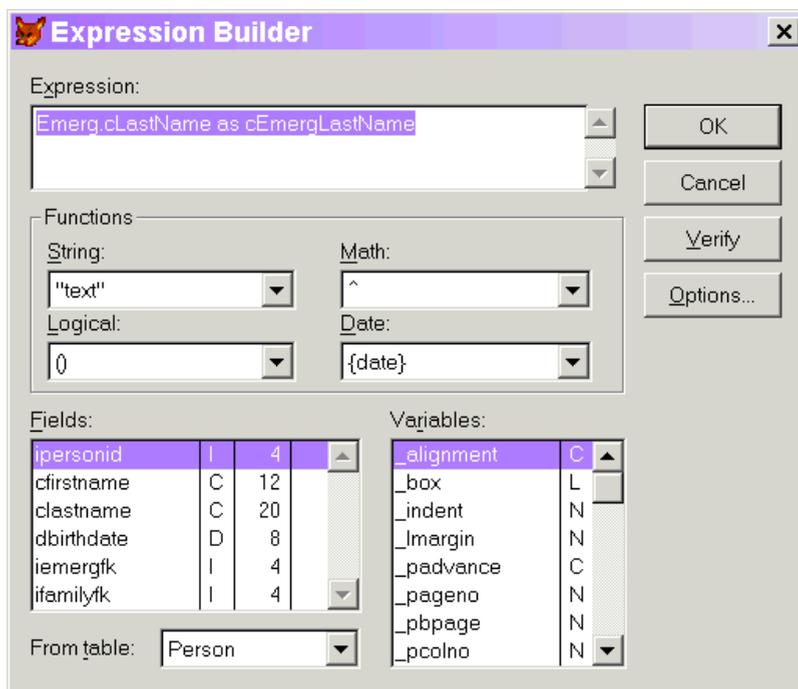
The query also shows how to assign a field name to a field in the result. The AS keyword indicates that the preceding expression should be given the field name following. In the example, where several fields in the query have the same name, this technique prevents VFP from attaching a suffix ("\_a", "\_b") to distinguish them. (That is, without AS, the fields in the result would be cFirstName\_a, cLastName\_a, cFirstName\_b, cLastName\_b.) AS is also useful when you have computed fields in a query.

Assigning field names is another area where VFP 8 improves the Query Designer and View Designer. In previous versions, there was no way to specify the name for a result field. VFP 8 provides two ways to specify the field name. Figure 7 demonstrates adding the field name in the Functions and expressions textbox on the Fields page.



**Figure 7 Specify a result field name – One way to get result field names into the query is by adding the result field name in the Functions and expressions textbox.**

In addition, when you click the ellipsis button next to the Functions and expressions textbox, the Expression Builder opens. All the fields from all the tables in the query are available. Once you choose the field you want to use, you can type the result field name in before clicking OK to accept the expression. Figure 8 shows this approach.



**Figure 8 Specifying a result field name – You can add a result field name to any expression in the Expression Builder.**

A self-join isn't limited to just multiple instances of the same table. Here's a more complex query that extracts the phone numbers of the emergency contact as well. Note that, in this case, the result may contain multiple records for a given person, if there are multiple phones on file for the person's emergency contact.

```

SELECT Person.cFirstName, Person.cLastName, ;
       Emerg.cFirstName AS cEmergFirstName, ;
       Emerg.cLastName AS cEmergLastName, ;
       Phone.cAreaCode, Phone.cNumber, ;
       Phone.cExtension, HasPhone.cType;
FROM Person ;
JOIN Person Emerg ;
ON Person.iEmergFK = Emerg.iPersonId ;

```

```

JOIN HasPhone ;
    ON Emerg.iPersonId = HasPhone.iPersonFK ;
JOIN Phone ;
    ON HasPhone.iPhoneFK = Phone.iPhoneId ;
ORDER BY 2, 1 ;
INTO CURSOR Contacts

```

## Adding outer joins to the mix

The queries above all use *inner joins*, in which only those records that match in the joined tables appear in the result. Any record in one table that has no counterpart in the table to which it's being joined is omitted. FoxPro has included inner joins since SQL SELECT was first added to the language and all of the queries above can be written with join conditions in the WHERE clause rather than using the newer JOIN syntax.

However, many useful queries include the unmatched records in the result instead of omitting them. These queries are called *outer joins*. There are three types - they differ in which tables contribute unmatched records to the result. A *left outer join* includes unmatched records from the first (left) table listed in the join. Correspondingly, a *right outer join* includes unmatched records from the second (right) table listed. A *full outer join* includes unmatched records from both tables. No matter which type of outer join is used, any fields in the result that come from the table with no matching records are filled with null values (.NULL.).

The words "inner" and "outer" are usually omitted when talking about joins, so you'll hear people refer to joins (that is, inner joins), left joins, right joins and full joins. Similarly, the INNER and OUTER keywords of SELECT are optional, but the Query Designer always includes them.

Each join in the FROM clause is considered separately for the purposes of distinguishing inner and outer joins and determining whether an outer join is left, right or full. So a single query may contain a left join, two inner joins and a full join, for instance.

As with inner joins, a two-table outer join is pretty simple to assemble. Going back to the TasTrade data, here's a query that shows each company and the date of its most recent order. Every customer is included, even if they've never placed an order.

```

SELECT Customer.Company_Name, MAX(Orders.Order_Date) AS dMostRecent;
FROM Customer ;
    LEFT JOIN Orders ;
        ON Customer.Customer_Id = Orders.Customer_ID ;
GROUP BY 1 ;
INTO CURSOR MostRecentOrder

```

This is a left join, so all Customer records are included, but some Orders records could be omitted. (In this example, omitted Orders records would be a sign of trouble since every Orders record should be associated with a Customer record.) The Order\_Date field is drawn from Orders; it's filled with .NULL. for any company with no orders on file.

When more than two tables are involved, it gets a little more complex. Sometimes, you have to use an outer join in a situation that doesn't appear to need it, so that records already added via an outer join stay in the data set.

Returning to the Membership database, suppose we want to get a list of people and their phone numbers. If every person were required to provide a phone number, this would be easy. There are several ways to write the query. Here's one that works:

```
SELECT cFirstName, cLastName, cAreaCode, cNumber, cExtension ;
FROM Person ;
JOIN HasPhone ;
ON Person.iPersonId = HasPhone.iPersonFK ;
JOIN Phone ;
ON HasPhone.iPhoneFK = Phone.iPhoneId ;
INTO CURSOR PhoneList
```

However, anybody with no phone records falls out of the result set. To include everyone, we need to use an outer join:

```
SELECT cFirstName, cLastName, cAreaCode, cNumber, cExtension ;
FROM Person ;
LEFT JOIN HasPhone ;
ON Person.iPersonId = HasPhone.iPersonFK ;
LEFT JOIN Phone ;
ON HasPhone.iPhoneFK = Phone.iPhoneId ;
INTO CURSOR PhoneList
```

In this case, we do an outer join from Person to HasPhone, which makes sure that all the people are included. Then we do another outer join between that result and Phone. The second outer join may seem extraneous, since every HasPhone record has a matching Phone record. But the results of the first join includes some records with null data for the HasPhone part, so the outer join is needed to keep those records from falling out.

For this example, there are some alternatives that let you dispense with the second outer join. Any version that does the join between HasPhone and Phone before the join to Person will work. Here's a query using the nested style that needs only one outer join:

```
SELECT cFirstName, cLastName, cAreaCode, cNumber, cExtension ;
FROM Person ;
LEFT JOIN HasPhone ;
JOIN Phone ;
ON HasPhone.iPhoneFK = Phone.iPhoneId ;
ON Person.iPersonId = HasPhone.iPersonFK ;
INTO CURSOR PhoneList
```

With sequential style, putting the Phone-HasPhone join first would also work.

The moral of the story is that you need to consider the order in which you're asking VFP to join the tables (logically – the actual execution order may be different; see [Testing Optimization](#) below). Once records are added via an outer join, be careful not to push them back out with a later inner join.

Outer joins are another area where the QD/VD is easier to use in VFP 8 than in earlier versions. The addition of the tables themselves to the joins page makes it clear which table is which in an outer join.

Outer joins may also be an area where setting the priority of a join in the QD/VD is useful. Since the order of joins determines which joins must be outer joins and which can

be inner joins, the ability to refine the join order may allow you to avoid outer joins in some situations, or simply to create a more readable query.

## Combining outer joins with filters

Outer joins can be used like any other joins and mixed with other query clauses. However, there's a subtle trap when filtering data in a query that involves outer joins. It's possible for the filter to interfere with the join. Suppose we want to get a list of all people and their home phone numbers. Since not everyone has a phone number in the system, an outer join is needed. Here's a first attempt:

```
SELECT cFirstName, cLastName, cAreaCode, cNumber, cExtension ;
FROM Person ;
LEFT JOIN HasPhone ;
ON Person.iPersonId = HasPhone.iPersonFK ;
LEFT JOIN Phone ;
ON HasPhone.iPhoneFK = Phone.iPhoneId ;
WHERE HasPhone.cType = "H" ;
INTO CURSOR HomePhones
```

However, this query doesn't work because at least conceptually, joins are performed before filters. So, the query does an outer join that includes at least one record for each person, then it filters out those records where the phone type isn't "H". So people with no phones get filtered right back out (.NULL. = "H" is not true), as do people who have phones other than home phones.

The problem occurs when the filter directly impacts the outer join, that is, when it changes the set of records that are added to the result by the outer join. If the filter is based on a field or fields from the "all" table, the one for which all records are included in the result, there's no issue. But when the filter affects the "some" table, as above, there can be trouble. The solution, in general, is to move the filter into the join clause. Here's a query that lists everyone, providing the home phone if it's available.

```
SELECT cFirstName, cLastName, cAreaCode, cNumber, cExtension ;
FROM Person ;
LEFT JOIN HasPhone ;
ON Person.iPersonId = HasPhone.iPersonFK ;
AND HasPhone.cType = "H" ;
LEFT JOIN Phone ;
ON HasPhone.iPhoneFK = Phone.iPhoneId ;
INTO CURSOR HomePhones
```

Here, the join itself filters out all phone records not for home phones, then the outer join adds a dummy record for each person without a home phone.

In some cases, the filtering expression can be quite complex and requires restructuring the query to make sure that the right set of records is created. Putting the filter at the right point in the join sequence may involve trying lots of variations.

As noted above, not every filter condition needs to move into the JOIN clause. If the filter doesn't impact the outer join, it can remain in the WHERE clause. Filters that specifically should eliminate outer joined records can also go into the WHERE clause. For example,

suppose we want a list of children including their home phones, if available. This query does the trick.

```
SELECT cFirstName, cLastName, cAreaCode, cNumber, cExtension ;
FROM Person ;
LEFT JOIN HasPhone ;
ON Person.iPersonId = HasPhone.iPersonFK ;
AND HasPhone.cType = "H" ;
LEFT JOIN Phone ;
ON HasPhone.iPhoneFK = Phone.iPhoneId ;
WHERE dBirthdate > GOMONTH( DATE(), -18*12 ) ;
INTO CURSOR HomePhones
```

The most important thing to remember about using filters with outer joins is that the interactions are complex and you should test each query with a diverse data set before feeling confident that your queries work.

## Specifying filtered joins in the Query Designer/View Designer

Filtering in joins is another area where the tools are better in VFP 8. The Join page of the QD/VD includes a way to add parentheses in the join clause so you can write complex conditions involved AND and OR.

Consider the home phones example above. Suppose, instead, you wanted a list of everyone, including any fax number, whether it's a home fax or a work fax. Here's the FROM clause that chooses the right records:

```
FROM Person ;
LEFT JOIN HasPhone ;
ON Person.iPersonId = HasPhone.iPersonFK;
AND ( HasPhone.cType = "F";
OR HasPhone.cType = "X" ) ;
LEFT JOIN Phone ;
ON Phone.iPhoneId = HasPhone.iPhoneFK
```

Figure 9 shows the Join page in the QD for this join. By setting the priority of the AND higher than the priority of the OR, parentheses are added around the entire OR clause.

Left Table	Join type	Pri.	Right Table	Field Name	Not	Criteria	Value	Logical	Pri.
Person	Left Outer	0	Hasphone	Person.ipersonid	=		Hasphone.ipersonfk	AND	2
		0		Hasphone.cType	=		"F"	OR	1
		0		Hasphone.cType	=		"X"		0
<Prev join>	Left Outer	0	Phone	Phone.iphoneid	=		Hasphone.iphonefk		0

**Figure 9 Adding parentheses to joins – The right-hand priority column lets you specify the order in which logical conditions are to be evaluated. Conditions with higher priority are evaluated later.**

## Counting with outer joins

Returning to the TasTrade data, suppose we want to count the number of orders placed by each customer and include 0 for any customer who hasn't ordered. Here's a first attempt:

```
SELECT Company_Name, COUNT(*) AS OrderCount;
```

```

FROM Customer ;
  LEFT JOIN Orders ;
    ON Customer.Customer_Id = Orders.Customer_Id ;
GROUP BY Customer.Customer_Id

```

A look at the results shows no records with OrderCount = 0. However, a look at the original data shows that Uncle's Food Factory doesn't have any orders. Why does it contain 1 for OrderCount?

To see why, you have to understand the way the aggregate functions (COUNT(), SUM(), AVG(), MIN() and MAX()) and the GROUP BY clause work. First, the query performs joins and filtering and produces an intermediate result. The GROUP BY clause is applied to that intermediate result with the functions performed against those records. COUNT(\*) indicates that we want the number of records in the group. Since every customer is included in the intermediate result, there's at least one record per customer, even though it may not be the result of a match with Orders.

The solution is to specify a field from Orders in the COUNT() function, for example, COUNT(Order\_Id). The Order\_Id field for the unmatched records contains .NULL. – null values are ignored by the aggregate functions. So, to count the number of orders per customer, we use:

```

SELECT Company_Name, COUNT(Order_Id) AS OrderCount;
FROM Customer ;
  LEFT JOIN Orders ;
    ON Customer.Customer_Id = Orders.Customer_Id ;
GROUP BY Customer.Company_Name

```

The other aggregate functions don't run into this problem because they always require a field name.

## Using the Designers for complex filters

The WHERE (filter) clause of a query can be any expression, as long as the result is logical. It's not unusual for the WHERE clause to include a complex combination of ANDs and ORs. Prior to VFP 8, the QD/VD couldn't handle any conditions except those that used the default precedence rules for combining AND and OR. (By default, AND is evaluated before OR.)

Like the Join page, the Filter page of the QD/VD in VFP 8 includes a way to add parentheses to conditions so they're evaluated in the right order. Each row can have a priority set, and as with joins, higher priority operations occur later.

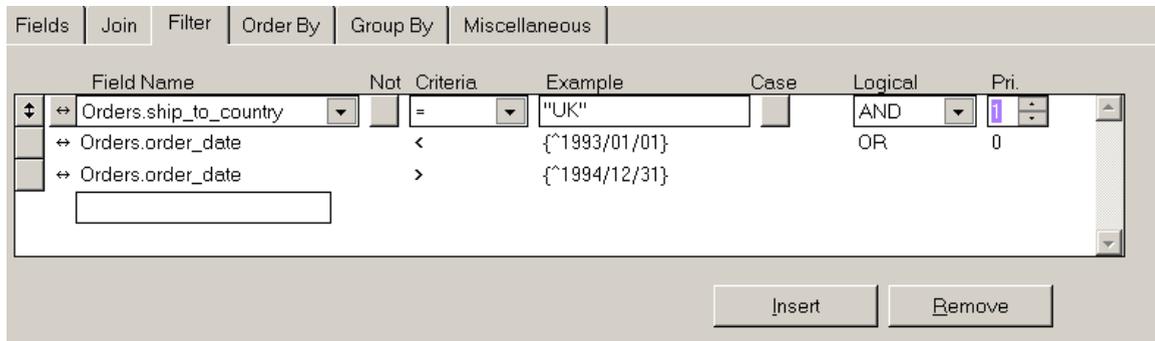
For example, using TasTrade, suppose you want to get a list of all orders shipped to the UK before 1993 and after 1994. Here's the WHERE clause for that query:

```

WHERE Orders.ship_to_country = "UK";
  AND ( Orders.order_date < {^1993/01/01};
  OR Orders.order_date > {^1994/12/31} )

```

Figure 10 shows the Filter page of the Query Designer for this query.



**Figure 10 Adding parentheses to filters – The priority spinners on the Filter page let you indicate the order in which logical operators are evaluated.**

The Having dialog, accessible through the Having button on the Group By page, also includes a Priority column, to add parentheses in the Having clause.

## Special Queries

Once you have joins and filtering working together properly, there are still lots of queries that require additional special handling. This section looks at subqueries, queries involving UNIONS, and a variant of the multiple unrelated siblings problem.

### Handling Multi-Step Processes

Some problems require the results of one query to be fed into another query. While it's often possible to use a series of queries in this case, sometimes it makes more sense to consolidate the process into a single command. In other cases, embedding one query in another is the only way to get the desired result with SQL commands.

A query within a query (or within another SQL command) is called a *subquery*. It computes intermediate results, which are then used in the main command. The results of the subquery are used in a comparison in the WHERE clause of the main query. Subqueries must be enclosed in parentheses and may not be nested.

A number of special operators perform comparisons between expressions and subquery results, though the usual comparison operators can be used in some situations. The special operators are IN, EXISTS, ANY, SOME and ALL. IN or NOT IN are used far more than the others.

IN compares a specified field or expression to the results of a subquery and selects records for which the subquery results contain a match. NOT IN selects those records with no match in the subquery. In either case, the field list in the subquery should contain a single field or expression.

For example, to get a list of the people with no home phone number on record, you can use:

```
SELECT cFirstName, cLastName ;
FROM Person ;
WHERE iPersonId NOT IN ;
      (SELECT iPersonFK FROM HasPhone ;
       WHERE cType = "H") ;
```

```
INTO CURSOR NoHomePhone
```

The subquery creates a list of ids for people with home phones. Then, the NOT IN comparison excludes anyone with a match in that list.

IN is also useful when you need to compare against computed results. However, the subquery can be pretty odd-looking in such cases. This query creates a list of the oldest member of each family.

```
SELECT cFirstName, cLastName, iFamilyFK, dBirthdate ;
FROM Person ;
WHERE STR(iFamilyFK) + DTOS(dBirthdate) IN ;
      ( SELECT STR(iFamilyFK) + DTOS(MIN(dBirthdate)) ;
        FROM Person ;
        WHERE NOT EMPTY(dBirthdate) ;
        GROUP BY iFamilyFK ) ;
INTO CURSOR OldestInFamily
```

To provide the single field to match, we have to combine the family id and birth date into an expression. (It's also worth noting that, as written, this query omits families where no birth dates are available. To include those families would make the code even uglier.) In cases like this, it often makes more sense to perform two queries in sequence:

```
SELECT iFamilyFK, MIN(dBirthdate) dBirthMin;
FROM Person ;
WHERE NOT EMPTY(dBirthdate) ;
GROUP BY 1 ;
INTO CURSOR Oldest

SELECT cFirstName, cLastName, Person.iFamilyFK, dBirthdate ;
FROM Person ;
JOIN Oldest ;
ON Person.iFamilyFK = Oldest.iFamilyFK ;
AND Person.dBirthdate = Oldest.dBirthMin ;
INTO CURSOR OldestInFamily
```

Here, the first query creates a cursor with the family id and birth date of the oldest family member, then the second does a join to pull out the person's name.

The other subquery comparison operators are generally used less. The big reason is that subqueries using them are usually *correlated*. A correlated subquery includes one or more fields from the main query and thus must be executed for each record examined in the main query. The examples above (like most subqueries in involving IN and NOT IN) are not correlated - the subquery stands alone, so it can be executed once, then used to check each record in the main query.

EXISTS tests whether the subquery found any records at all. The main query selects those records for which the subquery produces any results. EXISTS queries, which are usually correlated, can often be rewritten to use IN and an uncorrelated subquery.

The ALL operator is used together with a comparison operator to test whether an expression compares as specified to ALL the records in the subquery result. Similarly, ANY and SOME, which are identical, check whether the expression compares as specified to any record in the subquery result.

Surprisingly, the QD/VD can handle some subqueries. To specify a subquery, choose the appropriate operator from the Criteria dropdown, then type (or paste) the subquery into the Example textbox. In VFP 7 and earlier, only the IN subquery operator is included in the dropdown. (In fact, while you can create queries involving IN subqueries in those versions, you can't open them with the Designer once you've saved them.) In VFP 8, EXISTS has been added to the dropdown. In addition, the two-way editing of VFP 8 makes it easier to add the subquery, since you can type it into the SQL window rather than being restricted to the Example textbox. Be aware, though, the Designer can only handle fairly simple subqueries. Queries involving complex subqueries still have to be written by hand.

## Consolidating query results

In some situations, it takes multiple queries to collect all the results, but a single result set is called for. For example, using TasTrade, we might want a list of all the companies with which there's a business relationship (customers, suppliers and shippers), say for a holiday card list.

The UNION clause of SELECT combines the results of two (or more) separate queries into a single result set. To use UNION, the field lists of the queries must contain the same number of items and corresponding fields must be the compatible. ("Compatible" is explained below.)

To create a list of all companies in TasTrade, use a query like this:

```
SELECT Company_Name ;
      FROM Customer ;
UNION ;
SELECT Company_Name ;
      FROM Supplier ;
UNION ;
SELECT Company_Name ;
      FROM Shippers ;
      INTO CURSOR AllCompanies
```

This query won't take us very far for a holiday mailing list, though, since it doesn't include addresses. The problem is that the Shippers table only has the company name and no address information. Since each query in the UNION must have a matching field list, we have to supply dummy data for fields that are missing. In this query, the empty string is substituted for the address-related fields from Shippers:

```
SELECT Company_Name, Address, City, Region, Postal_Code, Country ;
      FROM Customer ;
UNION ;
SELECT Company_Name, Address, City, Region, Postal_Code, Country ;
      FROM Supplier ;
UNION ;
SELECT Company_Name, "", "", "", "", "" ;
      FROM Shippers ;
      INTO CURSOR AllCompanies
```

In VFP 7 and earlier versions, the first query in the UNION determines the type and size of the fields in the result. The field lists for all the other queries must be *union-*

*compatible* with the first. The general rule is that actual fields listed in subsequent queries must exactly match the type and size of the corresponding fields in the first query. However, if the subsequent query uses an expression as a field, it's sufficient for it to be the same type and no larger than the corresponding field in the first query. The example works because the listed fields are defined the same way in each of the three original tables, and the third query uses an expression evaluating to the empty string, which has length 0, for the fields not in the Shippers table.

Often, though, fields in the first query must be adjusted to make sure all the data will fit. Use functions like SPACE() and PADR() to ensure that character fields are large enough for all the possible entries. For numeric fields, use a template of 0's plus decimal point, if appropriate. For example, to leave space for a field with 4 digits to the left and 2 digits to the right of the decimal point, use the value 0000.00 rather than just 0. For date fields, substitute the empty date, {}.

You can even supply an empty memo field by creating a cursor with a single memo field, adding one record, then adding that cursor to the list of tables in the query. Here, an empty memo field is added to each record in the list of companies. Since Dummy contains only one record, there's no need for a join condition.

```
CREATE CURSOR Dummy (mMemo M)
INSERT INTO Dummy VALUES ("")

SELECT Company_Name, Address, City, Region, Postal_Code, Country, ;
       mMemo ;
FROM Customer, Dummy ;
UNION ;
SELECT Company_Name, Address, City, Region, Postal_Code, Country, ;
       mMemo ;
FROM Supplier, Dummy ;
UNION ;
SELECT Company_Name, "", "", "", "", "", mMemo ;
FROM Shippers, Dummy ;
INTO CURSOR AllCompanies

USE IN Dummy
```

Adding a field in every query in the UNION like this is an unusual case, but the technique is quite handy when one query in a UNION involves a memo and others don't. (Also, beginning in VFP 7, you can use this technique together with the READWRITE keyword to add an editable memo field to query results.)

In VFP 8, the rules for field matching in queries are looser. Some types are automatically converted to others in order to make them union-compatible. For example, if you specify a character field in one query and a memo field in the other in a UNION, the result uses a memo field in that position. In addition, if fields of the same type have different sizes, the result uses the largest size, rather than the size in the first query.

Table 1 shows the types that can be combined and the type of the resulting field.

**Table 1 Automatic Type Conversions – VFP 8 automatically converts and resizes some fields to make it easier to create UNIONS.**

<b>Table 1 Field Type</b>	<b>Table 2 Field Type</b>	<b>Result Field Type</b>	<b>Size considerations</b>
Character	Character	Character	The larger of the two original fields.
Character	Memo	Memo	
Character	Character Binary	Character Binary	The larger of the two original fields.
Character Binary	Memo	Memo	
Character Binary	Character Binary	Character Binary	The larger of the two original fields.
Numeric	Numeric	Numeric	The larger of the two original fields.
Numeric	Integer	Numeric	The larger of 11 and the size of the original numeric field, with as many decimal places as in the original numeric field.
Numeric	Double	Numeric	20 digits with the larger number of decimal places from the original fields.
Integer	Double	Double	The number of decimal places from the original double field.
Integer	Currency	Currency	
Double	Double	Double	The larger number of decimal places from the original fields.
Date	DateTime	DateTime	

Keep in mind that all clauses except ORDER BY and INTO apply to individual queries in the UNION. ORDER BY and INTO, which can be considered post-processing clauses, affect the query as a whole.

Another thing that affects the query as a whole are the optional ALL and DISTINCT keywords. By default, VFP performs all the queries and dumps all the results into one set, then it culls out any exact duplicates. Adding the DISTINCT keyword after UNION does the same thing. The ALL keyword tells VFP not to bother.

In VFP 8, you can't include memo fields in a UNION DISTINCT; you have to use UNION ALL if you have any memo fields. It turns out that VFP 7 and earlier never did compare the memo fields of records that otherwise matched, so when a query with UNION DISTINCT included memos, it was luck of the draw which memo data was

included. If the new behavior presents a problem for existing applications, you can use the new SYS(3099) function to restore the VFP 7 behavior for those queries affected.

Queries involving UNIONS are among those you still can't create with the Query Designer and View Designer interface. Considering the structure of the tools and the structure of such queries, this isn't really surprising. However, you can create and edit such queries using the SQL window. While the query won't be shown in the Designer pane, you can save it by making careful choices from the dialogs that appear. Also, you can create the individual queries within a UNIONed query using the tools, and then use cut-and-paste from the SQL window to build a UNIONed query.

## Multiple unrelated siblings redux – multiple detail bands

The problem of a single parent with multiple child tables comes up with a different spin when you consider reporting. A typical requirement is to have a report that shows the parent, then has information about one kind of child record, then information about the next kind of child, and so forth. In other words, the goal is to have a report that appears to have multiple details bands.

The Visual FoxPro Report Designer doesn't natively support such reports. But a creative query, together with some Report Designer cleverness, can trick VFP into doing the job.

Consider a "customer profile" report for TasTrade. The report shows customer information, then a list of employees the customer has dealt with, then a list of all the products the customer has ever ordered. The report also includes the total value of the orders placed through the employee and the total of the orders for each item.

The first step is to gather the data into a single cursor. The key to doing so is UNION. Use one query to collect the employee data, a second to collect product data, and UNION to produce a single result set. An extra field is added to identify the source of each record.

Here's the query that assembles the information. The cType field contains "E" for records originating in the Employee table and "P" for records originating in the Products table. The query organizes the report in customer order. Within each customer, employee information comes first, then product information.

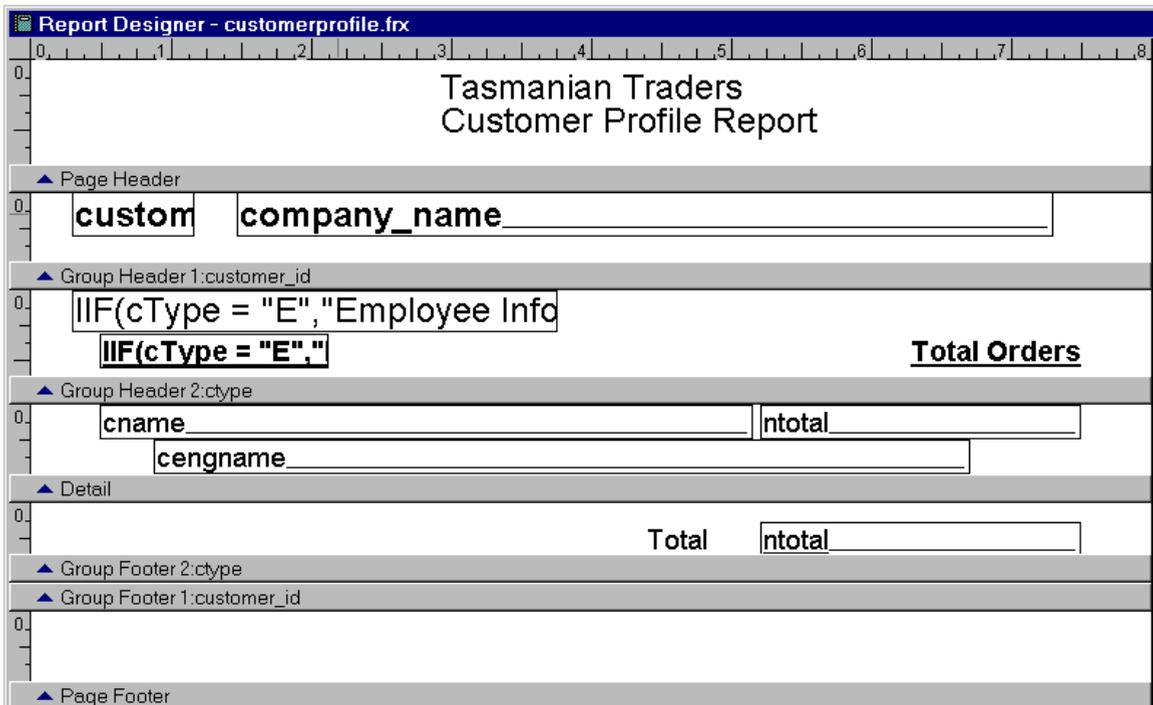
```
SELECT Customer.Customer_Id, Company_Name, ;
       PADR(TRIM>Last_Name) + ", " + First_Name, 40) AS cName, ;
       SPACE(50) AS cEngName, ;
       SUM(Quantity * Unit_Price) AS nTotal, ;
       "E" AS cType ;
FROM Customer ;
JOIN Orders ;
JOIN Order_Line_Items ;
ON Orders.Order_ID = Order_Line_Items.Order_ID ;
ON Customer.Customer_ID = Orders.Customer_ID ;
JOIN Employee ;
ON Orders.Employee_ID = Employee.Employee_ID ;
GROUP BY 1, 2, 3, 4, 6 ;
UNION ALL ;
SELECT Customer.Customer_ID, Customer.Company_Name, ;
       Products.Product_Name AS cName, ;
       Products.English_Name AS cEngName, ;
       SUM(Quantity * Order_Line_Items.Unit_Price) AS nTotal, ;
```

```

    "P" AS cType ;
FROM Customer ;
  JOIN Orders ;
    JOIN Order_Line_Items ;
      JOIN Products ;
        ON Order_Line_Items.Product_ID = Products.Product_ID ;
        ON Orders.Order_ID = Order_Line_Items.Order_ID ;
        ON Customer.Customer_ID = Orders.Customer_ID ;
  GROUP BY 1, 2, 3, 4, 6 ;
  ORDER BY 2, 6, 3 ;
INTO CURSOR CustomerProfile

```

To create a report for this cursor, add two groups. The outer group is based on Customer\_ID and is set to start every group on a new page. The inner group is based on cType. Figure 11 shows the report in the Report Designer.



**Figure 11 Reporting on multiple unrelated siblings – The second grouping distinguishes employee information from product information.**

In the report itself, IIF() and the Print When dialog are used to print the right information for each section of the report. For example, in the group header for cType, the heading for the two sections of the report uses IIF() to decide what to print:

```
IIF(cType = "E", "Employee Information", "Product Information")
```

The cEngName field, which contains the English name of products, is set to remove blank lines, so the line containing that field disappears entirely in the Employee section of the report.

Figure 12 shows one page of the sample report.

Tasmanian Traders Customer Profile Report		
<b>ALFKI    Alfreds Futterkiste</b>		
<b>Employee Information</b>		
<u>Employee</u>		<u>Total Orders</u>
Brid, Justin		4536.6000
	Total	<u>4536.6000</u>
<b>Product Information</b>		
<u>Product</u>		<u>Total Orders</u>
Aniseed Syrup		60.0000
Licorice Syrup		
Chartreuse verte		378.0000
Green Chartreuse (Liqueur)		
Chef Anton's Gumbo Mix		119.2000
Chef Anton's Gumbo Mix		
Grandma's Boysenberry Spread		400.0000
Grandma's Boysenberry Spread		
Lakkalikööri		270.0000
Cloudberry Liqueur		
Manjimup Dried Apples		445.2000
Manjimup Dried Apples		
Original Frankfurter grüne Soße		26.0000
Original Frankfurter Green Sauce		
Pâté chinois		336.0000
Shepard's Pie		
Raclette Courdavault		825.0000
Courdavault Raclette Cheese		
Rössle Sauerkraut		775.2000
Rössle Sauerkraut		
Spegesild		24.0000
Salt Herring		
Vegie-spread		878.0000
Vegetable Sandwich Spread		
	Total	<u>4536.6000</u>

**Figure 12 Customer Profile Report – Using a clever query and some reporting tricks, you can create reports that appear to have multiple detail bands.**

This technique is most useful when the child tables are fairly similar in structure, but with liberal use of IIF() and Print When, can be used in many situations.

## Using GROUP BY and HAVING in VFP 8

VFP's version of SELECT has always been fairly loose in enforcing some rules of SQL. One in particular has to do with queries involving the GROUP BY clauses.

In the SQL standard and in most SQL engines, the field list for such queries is restricted to fields listed in the GROUP BY and fields involving the aggregation functions (COUNT, SUM, AVG, MAX, and MIN). In VFP 7 and earlier, there's no such restriction, and you can list any fields at all in such a query. When a field isn't part of the

GROUP BY clause and isn't aggregated, the value from the last record in the group is put in that field.

VFP 8 enforces the standard here, so a grouped query is restricted to fields listed in the GROUP BY and those using aggregation functions. For example, this query is valid in VFP 7, but not in VFP 8:

```
SELECT Customer.Customer_ID, Customer.Company_Name, ;
       COUNT(Orders.Order_Date) AS OrderCount ;
FROM Customer ;
LEFT JOIN Orders ;
ON Customer.Customer_ID = Orders.Customer_ID ;
GROUP BY Customer.Customer_ID ;
INTO CURSOR OrderCountByCustomer
```

In VFP 8, this query fails with error 1807, "SQL: GROUP BY clause is missing or invalid." To make the query work, you need to either remove the Company\_Name field from the field list or add it to the GROUP BY clause. Here's an alternate version taking the second approach:

```
SELECT Customer.Customer_ID, Customer.Company_Name, ;
       COUNT(Orders.Order_Date) AS OrderCount ;
FROM Customer ;
LEFT JOIN Orders ;
ON Customer.Customer_ID = Orders.Customer_ID ;
GROUP BY Customer.Customer_ID, Customer.Company_Name ;
INTO CURSOR OrderCountByCustomer
```

By the same token, the HAVING clause is restricted in VFP 8 to filtering on aggregate results. That is, the expressions in the HAVING clause must use at least one field based on an aggregate function. Since filtering original data in the HAVING clause slows queries down considerably (see [It's all in where you put it](#) below), this isn't a serious restriction.

Because some people may need to support existing code that breaks these rules, the new SYS(3099) function lets you turn the rules off. To restore the VFP 7 and earlier behavior, call it like this:

```
SYS(3099, 70)
```

To return to VFP 8 behavior, use:

```
SYS(3099, 80)
```

Your best bet is to leave the default setting for all new code, and only change this function to avoid rewriting working code.

## Improving Query Performance

There are a number of things you can do to optimize the performance of your queries. (Many of them will improve the speed of your Xbase code as well.) In addition, VFP 5 and later make it easier to see what part of a query needs improvement.

## Having the right tags

FoxPro's Rushmore optimization system is based on using available index tags to avoid reading records from the disk. When tags are available, FoxPro reads the (usually smaller) index files instead of the data and creates a list of records that meet the specified conditions. Then, only those records are retrieved.

In some cases, tags exist to optimize some conditions, but not others, so FoxPro handles the optimizable conditions first, then performs a sequential search through the records that meet those conditions to test the others. (Actually, sometimes FoxPro creates its own temporary index tags for those conditions, but this is still slower than having the tag in the first place.)

Obviously, the more of your conditions FoxPro can optimize by using existing tags, the faster your queries will run.

What makes a condition optimizable? In filters, it's a tag whose key *exactly* matches the left-hand side of the condition. The word "exactly" is important here. If the left-hand side is sort of like the key, it's not good enough. For example, if you have an index on UPPER(LastName), a condition of LastName = "Smith" can't be optimized. You need to use UPPER(LastName) = "SMITH" instead.

Surprisingly, for filters, placing the optimizable expression on the left-hand side is also important. Given the same tag on UPPER(LastName), the condition "SMITH" = UPPER(LastName) in the WHERE clause is *not* optimized.

Joins also need an exact match to the tag – a tag on UPPER(LastName) won't optimize a join condition based on LastName. However, there may be tags for each side of the condition. The rule is that only one index tag is applied, but you can't predict which one it is. If only one table involved has an appropriate tag, that one is usually used, but if both tables have tags, VFP decides which one helps more. Sometimes, VFP rejects all available tags and creates its own.

## Too much of a good thing

While the right tags can make all the difference in a query, it also turns out that the wrong tags can be a bad thing. When VFP uses a tag to optimize, it has to load a portion of the index into memory. The amount of the index that gets loaded is related to the index expression and, most importantly, to how balanced the values of that expression are. If 90% of the records in the table meet the specified condition, 90% of the index must be loaded. With large tables, this can cause serious slowdowns, especially when working across a network.

The solution is to consider the distribution of values when deciding whether to create a tag for a particular expression. If the values are widely distributed, a tag is likely to help. If most records have the same value, the tag may do more harm than good. (For more information about this problem, see the article "Rushmore: More is Less" in the May 1999 *FoxPro Advisor*.)

## What about deletion status?

FoxPro's two-stage deletion process (DELETE a record now and it's marked for removal at the next PACK) can have an effect on optimization. The SET DELETED command determines whether deleted records are included in query results or not. When DELETED is ON, records marked for deletion are filtered out. The key word is "filtered." Removal of deleted records works just like a filter condition and thus, can be optimized in the same way, with an appropriate index tag.

For many years, it was an article of faith in the FoxPro community that a tag on DELETED() was a good idea for each table in any application that operated with SET DELETED ON. However, a tag like this one:

```
INDEX ON DELETED () TAG IsDeleted
```

almost always is heavily weighted to the .F. side. That is, in most tables, most records are not deleted. So, for large tables or a network situation, having a tag on DELETED() turns out to slow queries down rather than speed them up.

## Memory matters – a lot

While testing query speed for this session, I ran into a problem. I didn't see the results I expected. Generally, adding a tag on a field used in a join or filter sped things up, but not by orders of magnitude, as I'd expected. I asked someone else to test and he got similar results.

I knew I'd tested these rules in the past and seen tremendous differences. What was going on in this case? The other tester and I both had enormous amounts of memory and a large page file which meant that big tables could be kept in memory and even huge ones could be moved in and out of memory without much difficulty. In this situation, using optimizable expressions didn't matter much and, in fact, as the number of tags grew so that the index file took up more room, things could slow down (even without the issue of network traffic).

How can you deal with this if you don't know how capable users' machines will be? The slowdowns from additional tags, when they occurred, were tiny while the potential gains from adding tags are huge. So the lesson is clearly to add the tags. Users on loaded machines won't notice the difference, but users on low-end machines will appreciate all optimization.

The other lesson is that, as Microsoft, and Fox Software before them, have been telling us for a long time, memory is perhaps *the* most important factor in how fast FoxPro processes data.

However, FoxPro can have too much memory, too. When VFP starts, it grabs a chunk of memory to work with, usually about half of the available memory. (You can check with SYS(3050,1).) Since other things are running at the same time (at least Windows, often much more), and those applications need memory as well, VFP can end up swapping data out to disk rather than using only actual, physical memory. VFP knows how to manage the memory it's been allocated extremely efficiently, but writing to disk is always slow.

Mac Rubel did extensive testing in this area several years ago. His results are documented in a series of *FoxPro Advisor* articles. (The articles appeared from November, 1997-October, 1998 and February-April, 1999.) In general, it's wise to use SYS(3050) to *lower* VFP's memory allocation so that it only works with physical memory. On a 64MB machine, I found that VFP 6's performance was maximized by setting the memory allocation around 24MB with a call like this SYS(3050, 1, 24000000). On a 256MB machine, VFP 8's performance seemed to be maximized by setting the memory allocation between 100MB and 120MB.

## It's all in where you put it

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

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.

Suppose you want to find all families in the Membership database whose zip code is between 19000 and 19999. This query, with the filter in WHERE, is optimizable:

```
SELECT * ;
  FROM Family ;
  WHERE BETWEEN(cZip,"19000","19999") ;
  INTO CURSOR FamiliesInZone
```

If we move the filter to the HAVING clause, the query can't be fully optimized:

```
SELECT * FROM Family ;
  HAVING BETWEEN(cZip,"19000","19999") ;
  INTO CURSOR FamiliesInZone
```

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:

```
SELECT Country, COUNT(*) ;
  FROM Customer ;
  GROUP BY Country ;
  HAVING CNT(*) >= 3 ;
  INTO CURSOR CountriesThreeOrMore
```

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. VFP 8 enforces this rule by default. Use SYS(3099) to turn the rule off temporarily.

## Testing Optimization

Until VFP 5, the only way to figure out whether a query was fully optimized was to test it, over and over, until you found the fastest arrangement. Getting it right was difficult since test data sets are often smaller than the actual data to be used and many factors can impact the speed of an operation. (In addition, memory issues– see [Memory matters a lot](#) - can make it very hard to tell which version will really be faster on a user's machine.)

Starting with VFP 5, it's much easier to see whether a query is fully optimized and, if not, what parts are and are not optimized. The SYS(3054) function controls a feature called SQL ShowPlan. There are five settings in VFP 7 and later:

- 1.SYS(3054,0) – turn off SQL ShowPlan.
- 2.SYS(3054,1) – turn on SQL ShowPlan for filters only.
- 3.SYS(3054,2) – turn on SQL ShowPlan for filters only and include the query in the result. (VFP 7 and later only)
- 4.SYS(3054,11) – turn on SQL ShowPlan for filters and joins.
- 5.SYS(3054,12) – turn on SQL ShowPlan for filters and joins and include the query in the result. (VFP 7 and later only)

In VFP 5, issuing SYS(3054) immediately produced a message in the active window, indicating the ShowPlan state. This message was removed in VFP6. Instead, the function returns, as a character string, the setting you pass it.

The output from the function (described below) also appears in the active window. You can send all the messages elsewhere using SET ALTERNATE and prevent it from appearing by defining and activating a window off-screen. VFP 7 introduced a new reporting option. If you pass the name of a variable as the third parameter, the entire ShowPlan report is stored in the variable rather than echoed to the active window.

Don't turn SYS(3054) on while you're running actual timing tests. The function slows things down and interferes with the results, so perform the two kinds of tests separately.

### Checking filters for optimization

For filters, SQL ShowPlan shows two kinds of information. First, it indicates which tags are being used to filter the table. Then, it provides an assessment of optimization for the table: none, partial or full.

Here's a simple query involving the Tastrade Customer table:

```
SELECT customer.company_name ;  
FROM customer ;  
WHERE company_name="H" ;  
INTO CURSOR test
```

If we run the query after issuing SYS(3054,1), ShowPlan provides the following information:

Rushmore optimization level for table customer: none

A look at the tags for Customer shows that, while there's a tag called Company\_Na, the key for it is UPPER(Company\_Name). A slight change to the query:

```
SELECT customer.company_name ;
FROM customer ;
WHERE UPPER(company_name)="H" ;
INTO CURSOR test
```

gives this ShowPlan output:

```
Using index tag Company_na to rushmore optimize table customer
Rushmore optimization level for table customer: full
```

Matching the tag takes the query from totally unoptimized to fully optimized.

Be aware that ShowPlan indicates an optimization level of none for a table that's not filtered in the query. For example, for this query:

```
SELECT customer.company_name, orders.Order_date ;
FROM customer ;
JOIN orders ;
ON customer.customer_id = orders.customer_id ;
WHERE UPPER(company_name)="H" ;
INTO CURSOR test
```

the filter-only version of ShowPlan gives this feedback:

```
Using index tag Company_na to rushmore optimize table customer
Rushmore optimization level for table customer: full
Rushmore optimization level for table orders: none
```

The optimization level for Orders is none because there are no filters on Orders to be optimized.

Also, be aware that full optimization isn't always your goal. Because of the issue of index selectivity (see [Too much of a good thing](#) above), some queries will give better performance when only partially optimized.

## Checking joins for optimization

When ShowPlan is enabled for joins as well as filters, the output also includes one line for each join, indicating how it was optimized, if at all. The output even includes lines for any missing join conditions, indicating that a Cartesian join was used. (A Cartesian join is one in which every record of one table is matched with every record of another table.)

If you issue SYS(3054,11) before executing the simple join above, the join portion of the ShowPlan output is:

```
Joining table customer and table orders using index tag Customer_I
```

It indicates which tables were joined and which tag, if any, was used to join them. At most one tag is used even if both tables have appropriate tags. VFP decides which tag to use, if multiple tags are available.

If no tag is available, the ShowPlan output says "using temp index". Sometimes, ShowPlan says it's using a temporary index even when one table has a tag that applies. This appears to happen when the tables are of very different sizes and only the smaller table has a tag. VFP decides that creating a temporary tag for the larger table is more efficient than using the existing tag of the smaller table.

For multi-table joins, the order in which join information appears indicates the order in which the joins are actually being performed. This order may be quite different from the logical order of the joins described above. For example, this query:

```
SELECT customer.company_name, ;
       orders.order_date, ;
       order_line_items.quantity, ;
       products.English_name ;
FROM customer ;
   JOIN orders ;
     JOIN order_line_items ;
       JOIN products ;
         ON order_line_items.product_id = ;
           products.product_id ;
         ON orders.order_id = order_line_items.order_id ;
       ON customer.customer_id = orders.customer_id
```

produces the following ShowPlan output:

```
Rushmore optimization level for table customer: none
Rushmore optimization level for table orders: none
Rushmore optimization level for table order_line_items: none
Rushmore optimization level for table products: none
Joining table customer and table orders using index tag Customer_i
Joining intermediate result and table order_line_items using index tag
Order_id
Joining table products and intermediate result using temp index
```

The tables here are joined in exactly the reverse order from what we'd expect based on the structure of the query. The sizes of the tables and the tags available lead the VFP engine to believe this is the optimal join order. (Optimization of filter conditions is none for all tables because the query contains no filter conditions.)

For optimization, it doesn't matter whether you use nested or sequential joins. It appears that FoxPro uses the syntax to understand the desired result, but then joins the tables in the most efficient order.

Adding an outer join changes the optimization result. If we make the join between Customer and Orders in the example above into a left outer join so all customers appear in the result, ShowPlan produces this output:

```
Rushmore optimization level for table orders: none
Rushmore optimization level for table order_line_items: none
Rushmore optimization level for table products: none
Joining table orders and table order_line_items using index tag
Order_id
Joining table products and intermediate result using temp index
Rushmore optimization level for table customer: none
Rushmore optimization level for intermediate result: none
```

## Joining table customer and intermediate result using temp index

With outer joins involved, the nested vs. sequential issue is a little more significant since FoxPro can no longer rearrange the joins as it pleases. However, my tests show that the join style used makes little difference in most cases. However, when tables of vastly different sizes are involved in an outer join, the choice of style might make a difference since it may determine the order in which the tables are actually joined.

## Using ShowPlan information

The output from ShowPlan can help us to optimize queries. The simplest case, of course, is to add a tag or modify a condition so that it uses an existing tag, as in the UPPER(Company\_Name) example above.

However, there are times when you know more about your data than VFP does. In such cases, you may want to insist that joins be performed in a certain order. The FORCE clause of SELECT lets you specify that joins must be performed in the order listed rather than having the optimizer try to figure out the best choice. You can also use parentheses around join clauses to force some joins to be performed before others.

You can also use FORCE when you've determined the optimal order for joins. Arrange the query so that the joins are listed in that order and add the FORCE clause to prevent the VFP engine from trying to figure out which way to do things. This saves the time the optimizer would require to figure out which order to use.

## Optimization *can* be a problem

One of the ways FoxPro optimizes queries is by taking a shortcut. If a query is fully optimizable, involves a single table, has no calculated fields and no grouping, and puts its results in a cursor, VFP simply filters the source table. This saves the time needed to actually create the cursor. You can tell when VFP has done so by checking DBF() for the result cursor. For example, with DELETED OFF, the following query:

```
SELECT First_Name, Last_Name ;  
FROM Employee ;  
WHERE Group_Id = " 3" ;  
INTO CURSOR Group3
```

creates a cursor that is simply a filtered view of the Employee table. DBF("Group3") returns something like:

```
C:\PROGRAM FILES\MICROSOFT VISUAL FOXPRO  
8\SAMPLES\TASTRADE\DATA\EMPLOYEE.DBF
```

Cursors that are really filtered tables can't be used in certain operations and can give bad results in others. In addition, VFP doesn't remember to apply a filter of DELETED(=).F. to the original table when a query is run with DELETED ON and no real cursor is created. If DELETED is subsequently turned OFF, the filtered cursor contains the deleted records of the original that otherwise meet its criteria. This can give wrong results.

In FoxPro 2.x and VFP 3.0x, the only solution was to force the query to be less than fully optimizable. In VFP 5 and later, the NOFILTER clause forces VFP to create a "real" cursor. For example, after executing:

```
SELECT First_Name, Last_Name ;  
FROM Employee ;  
WHERE Group_Id = " 3" ;  
INTO CURSOR Group3 NOFILTER
```

a check of DBF("Group3") shows a temporary file such as:

```
C:\TEMP\08334986.TMP
```

VFP 7 introduced the READWRITE clause that allows you to create editable cursors with SELECT. When READWRITE is included, VFP always creates a real cursor, whether the NOFILTER clause is present or not.

Because of the bug involving deleted records, it's best to use NOFILTER or READWRITE any time VFP might fail to create a real cursor. Make exceptions only when the speed gain from filtering far outweighs the risk of mishandling deleted records.

## Summary

SQL-SELECT provides an extremely powerful mechanism for collecting data. Getting started with simple queries is easy. But complex queries call for a methodical approach with plenty of testing along the way. SYS(3054) provides a means for understanding what's going on under the hood and tuning queries for optimal performance.

With the tools available and some practice, anyone can make the most of SQL-SELECT. With the enhancements to the Query Designer and View Designer in VFP 8, doing so is easier than ever.

## Acknowledgements

Andy Neil, Steve Sawyer, Anders Altberg and Chin Bae all helped me assimilate and understand the join syntax. Jim Slater tested my optimization results. Mark Wilden pointed out the problem with deleted records and filtered cursors. Andy Neil reviewed an earlier version of these session notes and suggested several improvements. Anders Altberg helped me understand the issues involved with filtering fields added by an outer join. Gene Goldhammer, Aleksey Tsingauz, and Fabian Winternitz of the Microsoft Fox team helped me to grasp VFP 8's changes to the Query Designer and View Designer. Thanks to all of them and anyone I may have missed for making these notes better.

Some of the material in these notes is excerpted from *Hacker's Guide to Visual FoxPro 7.0* by Tamar E. Granor, Ted Roche, Doug Hennig and Della Martin, Hentzenwerke Publishing, 2002.

*Copyright, 2003, Tamar E. Granor, Ph.D.*