Using VFP 9's SQL Commands

Session VFP223

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

VFP 9 includes quite a few enhancements to Visual FoxPro's SQL sub-language, including removing many of the limits in queries. The use of subqueries has been expanded, and there are some performance improvements, as well.

While some of the changes are easy to explain and easy to demonstrate, others address more unusual situations that don't occur frequently. This session will show you when and how to take advantage of these changes.

This session assumes familiarity with VFP's SQL commands: SELECT, INSERT, UPDATE and DELETE.

VFP 9 has no limits

The most basic change to queries in VFP 9 is the elimination of a number of restrictions. In earlier versions, for example, the total number of joins and subqueries was limited to nine; in VFP 9, there's no limit. Table 1 shows the limits related to queries that were removed or raised in VFP 9.

Table 1 No limits—SQL queries were limited in a number of ways in earlier versions of Visual FoxPro. Many of those limits were lifted in VFP 9.

Description	Limit in VFP 8 (and earlier)	Limit in VFP 9
Total number of joins and subqueries	9	No limit
Number of UNIONs	9	No limit
Number of tables and aliases referenced	30	No limit
Number of items listed in IN clause	24	Based on SYS(3055) setting
Nesting level for subqueries	1	No limit

For everyday queries, most of us never ran into these limits. But each of them can pose problems in certain situations. For example, the limit on the number of joins can make it difficult to consolidate data in a fully normalized database. The limit on the number of items in an IN clause case be a problem when filter conditions are generated by an automatic process.

Lots of tables

When data is fully normalized, the old limits on joins, as well as those on the number of tables and aliases referenced, can make it difficult to pull together all the information for a particular entity. For example, consider a database that contains information about people with the ability to hold multiple addresses, phone numbers and email addresses for each. Add a way to distinguish different types and locations of addresses (voice, fax, personal, business, etc.) When you want to gather all the information for one individual, the number of tables involved (whether actual tables or alternate uses of a few tables) in the query can be quite large.

Figure 1 shows a database (Contacts.DBC, included in the materials for this session) that stores contact information. The Person table has the name and birth date for each person. ContactItem contains information about a single contact item (address, phone number, email, URL), using a link to the ItemType look-up table to indicate which kind of item it is. 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 whether it's a personal or business item. 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..

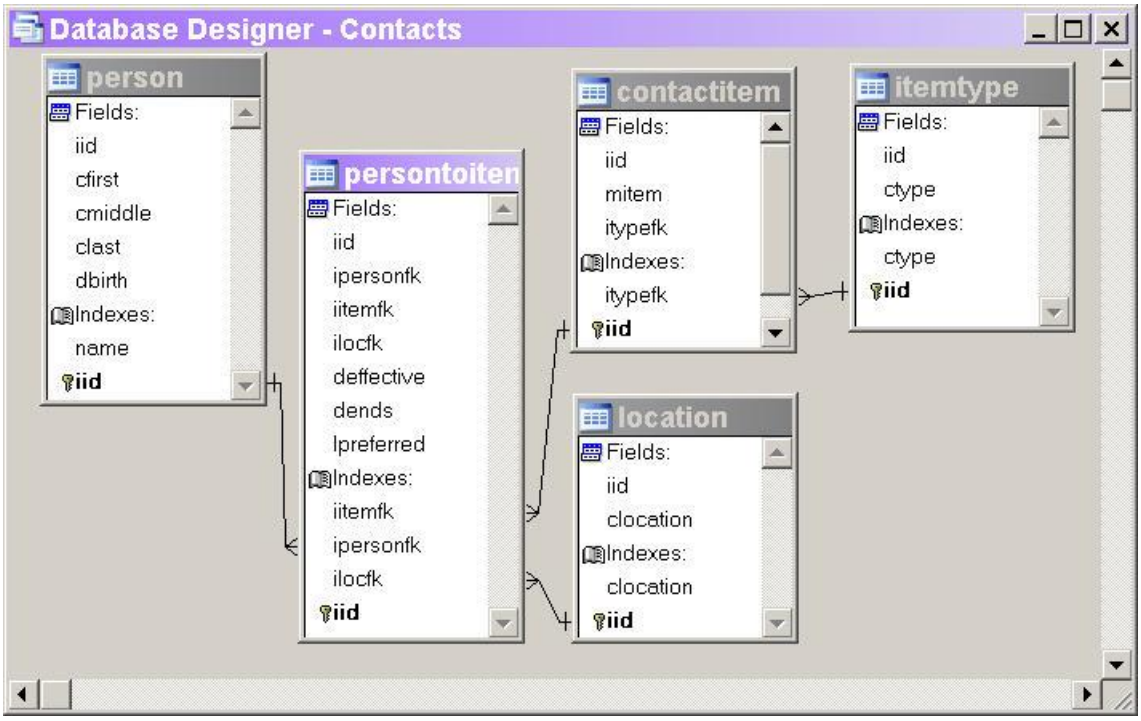

Figure 1 Contacts database—This database stores information about people and all their contact information, including addresses, phone numbers, email addresses and web addresses.

To collect all the personal contact information for each person requires a fairly complex join in which most of the tables are used several times. Listing 1 shows a query (SelectContacts.PRG in the session materials) that produces the desired result, putting the preferred address, phone, email and URL for a person into a single record.

Listing 1 Gathering personal data—To collect personal contact information for each person requires a complex query.

```
SELECT cFirst, cMiddle, cLast, ;
       Address.mItem Address, Phone.mItem Phone, ;
       Email.mItem Email, Web.mItem 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 ;
    INTO CURSOR PersonalContacts
```

This query lists 17 different aliases and performs 16 joins; it can't be run in VFP 8 and earlier versions, where it generates error 1805, "SQL: Too many subqueries." But VFP 9 executes it without a problem. (Actually, coming up with the correct query to produce the desired results took a fair amount of trial and error.)

Nearly unlimited IN operator

In earlier versions of VFP, the IN (list of items) operator was limited to 24 items in the list. While VFP 9 doesn't entirely remove the limit, it instead gives you control over it through the SYS(3055) function. Even without manipulating SYS(3055), the limit is significantly higher than in earlier versions. In my testing, I could include 154 items before I had to raise SYS(3055).

When you control the query yourself, the limit on the IN operator isn't generally a problem. You can usually find another approach to avoid a large IN clause. One solution is to store the list of values to a cursor and do a join with that cursor. For example, this query:

```
SELECT cFirst, cLast ;
   FROM Person ;
   WHERE UPPER(cLast) IN ("BLACK", "BROWN", "GREEN", "SILVER", "WHITE")
```

could be replaced with:

```
CREATE CURSOR Names (cName C(25))
INSERT INTO Names VALUES ("BLACK")
INSERT INTO Names VALUES ("BROWN")
INSERT INTO Names VALUES ("GREEN")
INSERT INTO Names VALUES ("SILVER")
INSERT INTO Names VALUES ("WHITE")
```

```
SELECT cFirst, cLast ;
   FROM Names ;
     JOIN Person ;
       ON UPPER(cLast) = RTRIM(cName)
```

However, you don't always have the chance to write this kind of code. In particular, other applications that access VFP data through OLE DB may generate queries that use the IN operator and offer no chance to code around it.

The session materials include a form (ChooseByAreaCode.SCX) that uses the Contacts database and presents a list of all the area codes in use. You can select as many as you want and click a button to display all the voice phone numbers in ContactItem in those area codes. The form (Figure 2) shows the query, the length of the query statement and the number of items in the IN clause.

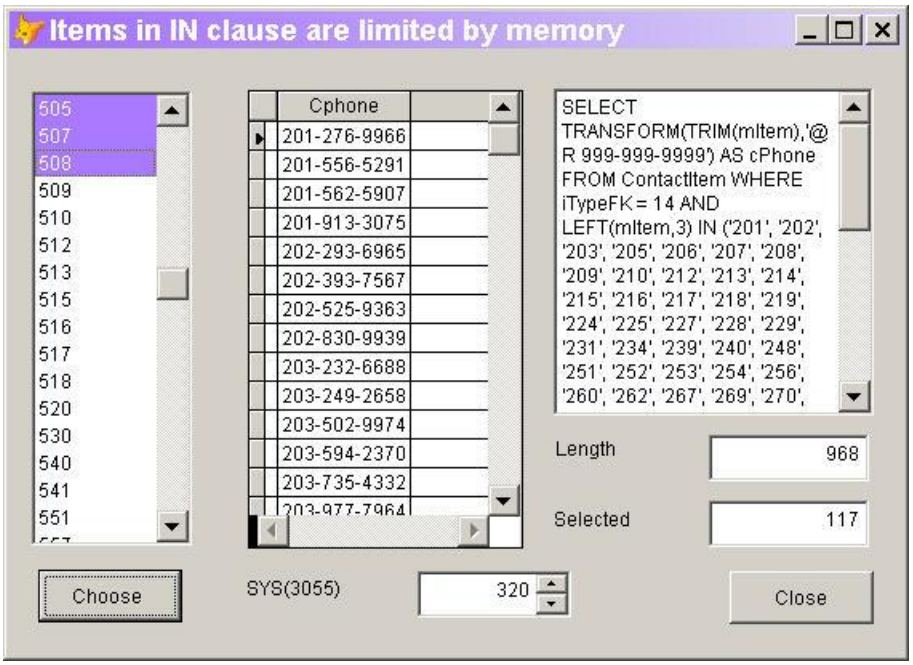

Figure 2 Nearly unlimited IN—The IN operator is no longer limited to 24 items. You can control the number of items with SYS(3055).

The form also lets you manipulate the value of SYS(3055) so you can experiment with the setting needed for different numbers of items.

Nested Subqueries

The ability to use subqueries (a query within a query) makes it possible to get some results with a single query that would otherwise require multiple queries. Perhaps the most common query involving a subquery is finding all the records in one table that are not in another. For example, this query (using the TasTrade database that comes with VFP) gets a list of companies in the Customer table who have placed no orders:

```
SELECT Company_Name ;
   FROM Customer ;
   WHERE Customer_ID NOT IN ;
```

```
      (SELECT Customer_ID FROM Orders) ;
   INTO CURSOR NoSales
```

Another common use for a subquery is to perform aggregation and then allow the main query to retrieve additional information about the aggregate results. For example, you can use a subquery to get a list of the products included in each customer's most recent order, as in Listing 2. (The query in Listing 2 is included in the session materials as CustProducts.PRG.) The subquery in this example is *correlated*, meaning that it uses a field from a table that's not listed in the subquery itself, only in the main query. In the example, OrdLast.Customer_ID is used in the WHERE clause of the subquery, but OrdLast is an alias for Orders defined in the main query.

Listing 2. The subquery here finds each customer's most recent order. Those results are used to get the list of products included in that order.

```
SELECT CustLast.Customer_ID, Product_ID ;
   FROM Order_Line_Items OLILast;
     JOIN Orders OrdLast;
       ON OLILast.Order_ID = OrdLast.Order_ID ;
     JOIN Customer CustLast;
       ON OrdLast.Customer_ID = ;
          CustLast.Customer_ID ;
   WHERE OrdLast.Order_Date = ;
     (SELECT MAX(Order_Date) ;
        FROM Orders Ord ;
        WHERE OrdLast.Customer_ID = ;
              Ord.Customer_ID ) ;
   INTO CURSOR CustProducts
```

In VFP 8 and earlier, subqueries cannot be nested. That is, the WHERE clause of a subquery can't use another subquery. VFP 9 allows nested subqueries, increasing the number of questions you can answer with a single query.

Suppose you want to find out what products a customer included on its first order, but not its most recent. While you could use the cursor created in Listing 2 in another query, in VFP 9, you can do the whole job with a single query, shown in Listing 3, and included in the session materials as ProductFirstNotLast.PRG.

Listing 3. Nesting subqueries lets you solve more problems with a single query rather than a series of queries.

```
SELECT Customer.Company_Name, Product_ID ;
  FROM Order_Line_Items ;
    JOIN Orders ;
      ON Order_Line_Items.Order_ID = Orders.Order_ID ;
    JOIN Customer ;
      ON Orders.Customer_ID = Customer.Customer_ID ;
    WHERE Orders.Order_Date = ;
      (SELECT MIN(Order_Date) ;
         FROM Orders Ord ;
          WHERE Orders.Customer_ID=Ord.Customer_ID );
      AND Product_ID NOT IN ;
        (SELECT Product_ID ;
           FROM Order_Line_Items OLILast;
             JOIN Orders OrdLast;
               ON OLILast.Order_ID = OrdLast.Order_ID ;
             JOIN Customer CustLast;
               ON OrdLast.Customer_ID = ;
                  CustLast.Customer_ID ;
```

```
          WHERE OrdLast.Order_Date = ;
            (SELECT MAX(Order_Date) ;
               FROM Orders Ord ;
               WHERE OrdLast.Customer_ID = ;
                     Ord.Customer_ID ) );
  INTO CURSOR FirstNotLast
```

More flexible subqueries

The previous section looked at one way subqueries have been improved in VFP 9. In fact, the VFP team did a lot of work with subqueries for this version. In VFP 8 and earlier, subqueries could be used only in the WHERE clause, that is, only in filtering data. In addition to removing the limit on nesting, VFP 9 also allows you to use subqueries in the field list and the FROM clause of a query and in the SET clause of the SQL UPDATE command. It also accepts a GROUP BY clause in correlated subqueries (subqueries that refer to one or more fields from tables in the main query, like the subquery in Listing 2). Finally, VFP 9 also allows you to use the TOP n clause in subqueries, as long as those subqueries are not correlated.

Derived tables—Subqueries in the FROM clause

The result of a subquery in the FROM clause is called a *derived table*. One use for a derived table is retrieving additional data when you use an aggregate function.

Using the TasTrade database that comes with VFP, consider the problem of finding out about each customer's most recent order. If all you want is the customer id and the order date, it's easy:

```
SELECT Customer_ID, MAX(Order_Date) ;
  FROM Orders ;
  GROUP BY 1 ;
  INTO CURSOR MostRecentOrder
```

This query is simple and works in every version of VFP. But suppose you want not just the customer id, but additional information about the order, such as the company name and contact person for the company and the name of the shipper. Prior to VFP 9, you need to use a correlated subquery, two queries in sequence or a very ugly subquery. Listing 4 (MostRecentCorrelated.PRG in the session materials) shows the solution using a correlated subquery, while Listing 5 (MostRecentTwoQueries.PRG) shows the two queries in sequence technique, and Listing 6 (MostRecentSubquery.PRG) shows the ugly subquery approach. What makes the subquery in Listing 6 so ugly is the need to combine the customer id and the most recent date into a single field for comparison.

Listing 4 Correlated subquery—Finding data associated with an aggregate is one reason to use a correlated subquery.

```
SELECT Orders.Order_ID, Customer.Company_Name as Cust_Name, ;
       Shippers.Company_Name AS Ship_Name, Orders.Order_Date ;
  FROM Orders ;
    JOIN Customer ;
      ON Orders.Customer_ID = Customer.Customer_ID ;
    JOIN Shippers ;
      ON Orders.Shipper_ID = shippers.Shipper_ID ;
  WHERE Orders.Order_Date = ;
    (SELECT MAX(Order_Date) ;
       FROM Orders Ord WHERE Orders.Customer_ID=Ord.Customer_ID );
```

```
  ORDER BY Cust_Name ;
  INTO CURSOR MostRecentOrders
```

Listing 5. Two queries, one result—Another way to find the data associated with an aggregate result is to use two queries in sequence. The first query does the grouping, then the second query joins with that result.

```
SELECT Customer_ID, MAX(Order_Date) as Order_Date ;
  FROM Orders ;
  GROUP BY 1 ;
  INTO CURSOR RecentOrder

SELECT Orders.Order_ID, Customer.Company_Name as Cust_Name, ;
       Shippers.Company_Name AS Ship_Name, Orders.Order_Date ;
  FROM Orders ;
    JOIN RecentOrder ;
      ON Orders.Customer_ID = RecentOrder.Customer_ID ;
      AND Orders.Order_Date = RecentOrder.Order_Date ;
    JOIN Customer ;
      ON Orders.Customer_ID = Customer.Customer_ID ;
    JOIN Shippers ;
      ON Orders.Shipper_ID = shippers.Shipper_ID ;
  ORDER BY Cust_Name ;
  INTO CURSOR MostRecentOrders
```

Listing 6. Same result, but hard to maintain—Yet another way to find data associated with an aggregate result is to use a subquery in the WHERE clause. The resulting query is hard to maintain.

```
SELECT Orders.Order_ID, Customer.Company_Name as Cust_Name, ;
       Shippers.Company_Name AS Ship_Name, Orders.Order_Date ;
  FROM Orders ;
    JOIN Customer ;
      ON Orders.Customer_ID = Customer.Customer_ID ;
    JOIN Shippers ;
      ON Orders.Shipper_ID = shippers.Shipper_ID ;
  WHERE Orders.Customer_ID + DTOS(Orders.Order_Date) in ;
    (SELECT Customer_ID + DTOS(MAX(Order_Date)) ;
       FROM Orders GROUP BY Customer_ID );
  ORDER BY Cust_Name ;
  INTO CURSOR MostRecentOrders
```

VFP 9 gives you a better alternative than any of these. You can perform the subquery on the fly right in the join clause. Putting the subquery into the join clause means it doesn't have to be correlated; in some cases, that means it will run faster. The query in Listing 7 (MostRecentOrderDetails.PRG in the session materials) uses this approach. Here, the subquery has moved into the JOIN clause, and doesn't need to combine the customer id and most recent date. Instead, the ON portion of the JOIN compares the two fields separately, much like the two query solution in Listing 5.

Listing 7. Derived table for aggregates—This query uses a subquery in the FROM clause (a derived table) to solve the problem of finding data associated with aggregates.

```
SELECT Orders.Order_ID, Customer.Company_Name as Cust_Name, ;
       Shippers.Company_Name AS Ship_Name, Orders.Order_Date ;
  FROM Orders ;
    JOIN (SELECT Customer_ID, MAX(Order_Date) as Order_Date ;
        FROM Orders CheckOrderDate ;
        GROUP BY 1) RecentOrder ;
      ON Orders.Customer_ID = RecentOrder.Customer_ID ;
```

```
      AND Orders.Order_Date = RecentOrder.Order_Date ;
    JOIN Customer ;
      ON Orders.Customer_ID = Customer.Customer_ID ;
    JOIN Shippers ;
      ON Orders.Shipper_ID = shippers.Shipper_ID ;
  ORDER BY Cust_Name ;
  INTO CURSOR MostRecentOrders
```

Be aware that a subquery in the FROM clause can't be correlated, that is, it may not refer to fields of tables used in the main query, only to fields of the tables it lists. All subqueries in the FROM clause are executed before the main query, thus it's not yet clear what records are in the result.

Computing fields with a subquery

In addition to supporting derived tables, VFP 9 lets you put subqueries in the field list of a query. That is, you can use a subquery to compute a field to appear in the result. A subquery used this way must return a single field and no more than a single record.

Why would you do this? Why not include the expression in the main query and add any necessary tables? As with derived tables, this approach is handy when grouping is involved. Working with the TasTrade data, suppose you want to find the total value of the orders placed for each customer in a particular year. Along with that, you want a great deal of customer information, including the address, phone number and fax number.

Clearly to compute the total value of orders for a customer, you need to group data from Order_Line_Items by customer. You can extract multiple fields from Customer along the way as long as you add them to the GROUP BY clause. (Prior to VFP 8, you could include additional fields, even without putting them in the GROUP BY clause. In VFP 8, you can do so by issuing SET ENGINEBEHAVIOR 70.) Listing 8 (CustomerTotalGrouped.PRG in the session materials) shows a query that retrieves the customer id, company name, address, phone and fax information along with the total value of that customer's orders.

Listing 8 Using GROUP BY to get parent data—In a grouped query, you can add fields from a parent table as long as you add them to the GROUP BY clause.

```
SELECT Customer.Customer_ID, Customer.Company_Name, ;
       Customer.Address, Customer.City, Customer.Region, ;
       Customer.Postal_Code, Customer.Phone, Customer.Fax, ;
       SUM(Quantity*Unit_Price) AS yTotal;
FROM Customer ;
  LEFT JOIN Orders ;
    JOIN Order_Line_Items;
      ON Orders.Order_ID = Order_Line_Items.Order_ID ;
    ON Customer.Customer_ID = Orders.Customer_ID ;
    AND BETWEEN(Order_Date,DATE(m.nYear,1,1),DATE(m.nYear,12,31)) ;
  GROUP BY 1, 2, 3, 4, 5, 6, 7, 8;
INTO CURSOR CustomerTotal
```

However, adding so many fields to the GROUP BY clause slows the query down. One alternative that works in this situation is to remove the fields from the GROUP BY clause and instead wrap them in MAX() or MIN(). As with grouping, since these fields are the same for all records in the group, using MAX() or MIN() doesn't change the results. This version, shown in Listing 9

(CustomerTotalMax.PRG), is somewhat faster than listing all the fields in the GROUP BY clause.

Listing 9. Removing fields from GROUP BY—Another approach to retrieving parent data in a grouped query is to wrap the extra fields in MAX() or MIN().

```
SELECT Customer.Customer_ID, MAX(Customer.Company_Name), ;
       MAX(Customer.Address), MAX(Customer.City), MAX(Customer.Region), ;
       MAX(Customer.Postal_Code), MAX(Customer.Phone), MAX(Customer.Fax), ;
       SUM(Quantity*Unit_Price) AS yTotal ;
FROM Customer ;
  LEFT JOIN Orders ;
    JOIN Order_Line_Items;
      ON Orders.Order_ID = Order_Line_Items.Order_ID ;
    ON Customer.Customer_ID = Orders.Customer_ID ;
    AND BETWEEN(Order_Date,DATE(m.nYear,1,1),DATE(m.nYear,12,31)) ;
  GROUP BY 1;
  INTO CURSOR CustomerTotal
```

But the ability to use a subquery in the field list provides an even simpler and more efficient solution to this problem. We can compute the total value of the orders in a subquery, allowing the main query to refer only to the parent table. Not only do we eliminate the extra fields in GROUP BY and the extra calls to aggregate functions, but we can also eliminate the outer join . Listing 10 (CustomerTotal.PRG) shows the query.

Listing 10. Subquery in field list—Using a subquery in the field list simplifies the problem of showing additional fields from the parent table.

```
SELECT Customer.Customer_ID, Customer.Company_Name, ;
       Customer.Address, Customer.City, Customer.Region, ;
       Customer.Postal_Code, Customer.Phone, Customer.Fax, ;
       (SELECT SUM(Quantity*Unit_Price) ;
          FROM Orders ;
            JOIN Order_Line_Items;
              ON Orders.Order_ID = Order_Line_Items.Order_ID ;
          WHERE BETWEEN(Order_Date,DATE(m.nYear,1,1),DATE(m.nYear,12,31)) ;
            AND Customer.Customer_ID=Orders.Customer_ID ) as yTotal ;
  FROM Customer ;
  INTO CURSOR CustomerTotal
```

In my tests, using the queries shown, the subquery version (Listing 10) was about 12% faster than the version with calls to MAX() (Listing 9), which in turn was about 12% faster than listing all the parent fields in the GROUP BY clause (Listing 8). The more fields from the parent in the field list, the greater the advantage of the subquery version.

Computing replacements in UPDATE

The third new place you can use subqueries is the SET clause of the UPDATE command. That is, you can use a subquery to compute the value to which a field is to be set. However, when you use this approach, the UPDATE command cannot include a subquery in the WHERE clause. In addition, you're limited to a single subquery in the SET clause, so you can't use this approach to compute the values of multiple fields.

In this example, imagine that you have a data warehouse (SalesByProduct) for TasTrade data that tells you how many of each product were sold and the dollar amount of those sales. It's designed

to hold data for a single month and you want to update it at the end of the month. (Code to create the data warehouse as a cursor is included in the session materials as CreateWarehouse.PRG.)

To update the data, you can use the following UPDATE commands. Set nMonth and nYear to the month and year whose data you're collecting before running the code in Listing 11 (SubqueryInSet.PRG in the session materials) . ("Correlated Updates," later in these notes, provides a better solution to this problem.)

Listing 11 Subquery in UPDATE—You can use a subquery in the SET portion of the SQL UPDATE command to calculate the new value on the fly.

```
UPDATE SalesByProduct ;
  SET TotalSales = (;
    SELECT NVL(SUM(Quantity*Unit_Price),$0) ;
      FROM Order_Line_Items ;
        JOIN Orders ;
          ON Order_Line_Items.Order_ID = Orders.Order_ID ;
      WHERE MONTH(Order_Date) = nMonth AND YEAR(Order_Date) = nYear;
        AND Order_Line_Items.Product_ID = SalesByProduct.Product_ID)

UPDATE SalesByProduct ;
  SET UnitsSold = (;
    SELECT CAST(NVL(SUM(Quantity),0) AS N(12)) ;
      FROM Order_Line_Items ;
        JOIN Orders ;
          ON Order_Line_Items.Order_ID = Orders.Order_ID ;
      WHERE MONTH(Order_Date) = nMonth AND YEAR(Order_Date) = nYear;
        AND Order_Line_Items.Product_ID = SalesByProduct.Product_ID)
```

The second UPDATE in the example uses VFP 9's CAST() function, which lets you change data types on the fly, to ensure that the quantity field is the right type and size.

Correlated subqueries and grouping

VFP 8 prohibits the GROUP BY clause in correlated subqueries. Because correlation can give you the same effect as grouping in many cases (see Listing 4, for example), you may not have run into this limit.

But there are a few situations where the ability to use GROUP BY in a correlated subquery makes it easier to get the desired results. Fortunately, VFP 9 permits grouping in a correlated subquery.

For example, consider the case where you want a list of customers who have placed at least one order totaling more than a specified value and had that order shipped somewhere other than their address of record. (You might be checking into suspicious transactions.) In VFP 8 and earlier, extracting this information is tricky. You need two queries in sequence: the first collects information about orders over the specified amount, while the second compares the shipping address of those records to the customer address and extracts customer information. Listing 12 (SuspiciousTwoQueries.PRG in the session materials) shows one way to do this, with the threshold value set to $4000.

Listing 12. Finding customers with suspicious big orders—In VFP 8 and earlier, it takes two queries to get a list of customers who've placed large orders and shipped them somewhere other than their home office.

```
SELECT Orders.Customer_ID, Orders.Ship_to_Address ;
  FROM Orders ;
    JOIN Order_line_items ;
      ON orders.order_id=order_line_items.order_id ;
  GROUP BY 1, 2, Orders.Order_ID ;
  HAVING SUM(Quantity*Unit_Price)> 4000 ;
  INTO CURSOR BigOrders

SELECT Company_Name, Order_ID, Order_Date ;
  FROM Customer ;
    JOIN BigOrders ;
      ON Customer.Customer_ID = BigOrders.Customer_ID ;
  WHERE BigOrders.Ship_to_Address <> Customer.Address ;
  INTO CURSOR Suspicions
```

In VFP 9, the ability to group in correlated subqueries means we can find this result with one query. Listing 13 (Suspicious.PRG) shows the one-query version.

Listing 13. Finding suspicious orders—Using GROUP BY in a correlated subquery in VFP makes it possible to find customers with suspicious orders in a single query.

```
SELECT Company_Name, Ord.Order_ID, Ord.Order_Date ;
  FROM Customer ;
   JOIN Orders Ord;
     ON Customer.Customer_ID = Ord.Customer_ID ;
  WHERE Ord.Order_ID in (;
    SELECT Orders.Order_ID;
    FROM Orders ;
      JOIN Order_Line_Items ;
        ON Orders.Order_ID=Order_Line_Items.Order_ID ;
        AND Orders.Customer_ID=Customer.Customer_ID ;
        AND Orders.Ship_to_Address <> Customer.Address ;
    GROUP BY Orders.Order_ID ;
    HAVING SUM(Quantity*Unit_Price)> 4000 ) ;
  INTO CURSOR Suspicions
```

As with a number of other situations where a single query replaces two queries, the version in Listing 13 is faster than the one in Listing 12.

Using TOP n in subqueries

The TOP n clause of SELECT lets you return only the first n records (or first n% of the records) of the result set. While the MIN() and MAX() functions let you choose the single smallest or largest value in a given field, TOP n lets you choose multiple items. For example, you can use it to see the 10 most recent orders or the 30 most expensive products.

In VFP 8 and earlier, you can't use the TOP n clause in subqueries. VFP 9 changes that, permitting TOP n in subqueries unless the subquery is correlated.

For example, using the Contacts data, suppose you want to know the names of the people who have the 20 oldest contact items. In VFP 8 and earlier versions, this takes two queries, one to find the 20 oldest contact items and a second to find out who they belong to. One solution is shown in Listing 14 (OldestContactsTwoQueries.PRG in the session materials).

Listing 14. Finding the oldest contacts—In VFP 8 and earlier, you need two queries to find the people who have the oldest contact items.

```
SELECT TOP 20 iPersonFK ;
  FROM PersonToItem ;
  ORDER BY dEffective ;
  INTO CURSOR OldestItems

SELECT cFirst, cLast ;
  FROM Person;
  WHERE iID in ( ;
    SELECT iPersonFK FROM OldestItems) ;
  INTO CURSOR OldestContacts
```

VFP 9 allows subqueries to use the TOP n clause, as long as the subquery is not correlated. That means that the oldest contacts can be found with a single query, as in Listing 15 (OldestContacts.PRG).

Listing 15. Simpler oldest contacts—In VFP 9, you can include TOP n in an uncorrelated subquery.

```
SELECT cFirst, cLast ;
  FROM Person ;
  WHERE iID in ;
  (SELECT TOP 20 iPersonFK ;
    FROM PersonToItem ;
    ORDER BY dEffective) ;
  INTO CURSOR OldestContacts
```

Not only is the second version shorter, but my tests show that it's about 50% faster than the two-query version.

Correlated updates

In addition to supporting subqueries in the SET clause, the SQL UPDATE command in VFP 9 has a new FROM clause that allows you to draw the update data from another table. This gives you what you might call *correlated updates*.

The example in Listing 11 has one serious drawback. You have to use a separate UPDATE command for each field you want to change. Using the FROM clause, we can accomplish the same result with a query followed by an UPDATE command. The code in Listing 16 (CorrelatedUpdate.PRG in the session materials) computes the new values and stores them in a cursor, then references that cursor in the UPDATE command.

Listing 16 Correlated UPDATE—The new FROM clause in SQL UPDATE lets you draw the replacement values from another table.

```
SELECT Order_Line_Items.Product_ID, ;
       SUM(Quantity*Order_Line_Items.Unit_Price) AS TotalSales, ;
       SUM(Quantity) AS UnitsSold ;
  FROM Order_Line_Items ;
    JOIN Orders ;
      ON Order_Line_Items.Order_ID = Orders.Order_ID ;
      AND MONTH(Order_Date) = nMonth AND YEAR(Order_Date) = nYear ;
  GROUP BY 1 ;
  INTO CURSOR MonthlySales

UPDATE SalesByProduct ;
  SET SalesByProduct.TotalSales = NVL(MonthlySales.TotalSales, $0), ;
      SalesByProduct.UnitsSold = NVL(MonthlySales.UnitsSold, 0) ;
```

```
  FROM SalesByProduct ;
    LEFT JOIN MonthlySales ;
      ON SalesByProduct.Product_ID = MonthlySales.Product_ID
```

Note that you can not only specify that values come from another table, but you can actually perform joins in the FROM clause to put together the list of values. In Listing 16, the outer join ensures that the records for products not sold in the specified month are set to 0.

In fact, the FROM clause of UPDATE supports subqueries (derived tables), so you can do this entire operation in a single UPDATE command, as shown in Listing 17 (CorrelatedUpdateSubquery.PRG).

Listing 17 Correlated UPDATE with subquery—Instead of running a query ahead of time to compute the results, you can use a subquery in the FROM clause of an UPDATE command.

```
UPDATE SalesByProduct ;
  SET SalesByProduct.TotalSales = NVL(MonthlySales.TotalSales, $0), ;
      SalesByProduct.UnitsSold = NVL(MonthlySales.UnitsSold, 0) ;
  FROM SalesByProduct ;
    LEFT JOIN (;
      SELECT Order_Line_Items.Product_ID, ;
             SUM(Quantity*Order_Line_Items.Unit_Price) as TotalSales, ;
             SUM(Quantity) AS UnitsSold ;
        FROM Order_Line_Items ;
          JOIN Orders ;
            ON Order_Line_Items.Order_ID = Orders.Order_ID ;
            AND (MONTH(Order_Date) = nMonth AND YEAR(Order_Date) = nYear) ;
        GROUP BY 1) AS MonthlySales ;
      ON SalesByProduct.Product_ID = MonthlySales.Product_ID
```

Correlated DELETEs

In VFP 8 and earlier, the SQL DELETE command lets you list only one table. While you can use subqueries in the WHERE clause, deleting records based on information in other tables can be tricky. VFP 9 allows you to list multiple tables in DELETE's FROM clause, joining them according to the usual rules. This provides a much cleaner way to perform "correlated deletion," that is, deletion from one table based on data in one or more other tables.

The syntax for a correlated DELETE is a little confusing. If the FROM clause of DELETE contains more than one table, you must specify the target table for the deletion between DELETE and FROM:

```
DELETE [Target] FROM Table1 [JOIN Table2 …]
```

Use the local alias of the target table between DELETE and FROM. This may be the name of the table, but if you assign a local alias to the table in the FROM clause, use that instead. (Note that the same rules apply for UPDATE, when the table being updated is also included in the FROM clause, as in Listing 17.)

The Contacts database tracks the effective date and end date for each contact item in the PersonToItem table. To remove all ContactItem records that are no longer in use by anyone takes two steps in VFP 8, as shown in Listing 18 (UncorrelatedDelete.PRG).

Listing 18 Two-step deletion—To delete obsolete contact items, first figure out which ones they are, then delete them. In VFP 8, it takes a query followed by a SQL DELETE.

```
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)
```

In VFP 9, the task can be accomplished with one DELETE command. The code in Listing 19 (CorrelatedDelete.PRG) deletes the same records as the code in Listing 18, but you can incorporate the query to figure out which records are obsolete right into the DELETE command.

Listing 19 Correlated DELETE—VFP 9 lets you list multiple tables in the DELETE command, though you have to tell it which one to delete from.

```
DELETE ContactItem FROM ContactItem ;
  JOIN (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() ) ;
    AS ExpDates ;
    ON ContactItem.iID = ExpDates.iItemFK
```

A more perfect UNION

The UNION clause of SELECT lets you combine the results of several queries into a single result set. In VFP 8, the rules for UNION were loosened, making the clause easier to use. VFP 9 offers two more improvements to UNION and one restriction.

Use names in ORDER BY with UNION

In earlier versions of VFP, when you use the UNION clause to combine multiple queries into a single result, the ORDER BY clause can list only the positions of the fields in the field list. You can't refer to fields by name, even if the field has the same name in every query in the UNION. This makes such queries hard to read and hard to maintain, since the ORDER BY list must be corrected if the field list changes.

In VFP 9, you can use field names in the ORDER BY clause of a UNIONed query. The field names you use are the ones in the result set. Be aware that when the names of corresponding fields in the UNION are different, the result draws the field name from the last query in the UNION.

Listing 20 (AllCompanies.PRG in the session materials) is a simple example that uses this capability. Even with this straightforward query, using the field names increases readability considerably.

Listing 20 Field names in ORDER BY—VFP 9 allows you to use the field names of the result set in the ORDER BY clause of a UNIONed query.

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

Insert data from UNIONed result

VFP 8 introduced the ability to populate a table or cursor directly from a query result with the addition of the INSERT INTO … SELECT syntax. This made it possible to compute results and add them in a single step.

VFP 9 adds another capability to that syntax: the query used can include the UNION clause. This means you can consolidate even more data and add it to a table or cursor in one step.

For example, suppose you have a data warehouse for TasTrade, containing the annual sales for each employee by product as well as an annual total for each employee. (Note that this data warehouse is different than the one described in "Computing Replacements in UPDATE.")

You can compute the sales for each product by each employee in a specified year with a single query; similarly, you can compute the totals for each employee for a year with one query. However, collecting both the product-specific and the total data requires either two queries or a query involving a UNION. If you want to add a year's worth of data to the warehouse, you can do it with the INSERT in Listing 21 (WarehouseUnion.PRG in the session materials).

Listing 21 INSERT from UNIONed SELECT—VFP 8 added the ability to INSERT directly from a query result; VFP 9 extends it to queries involving a UNION. Be sure to assign a value to the variable nYear before running this example.

```
CREATE CURSOR Warehouse (Product_ID C(6), Employee_ID C(6), ;
                         nYear N(4), ;
                         nTotalSales Y, nUnitsSold N(8))

INSERT INTO Warehouse ;
SELECT CrossProd.Product_ID, ;
       CrossProd.Employee_ID, ;
       m.nYear as nYear, ;
       NVL(nUnitsSold, 0), ;
       NVL(nTotalSales, $0);
   FROM (SELECT Employee.Employee_ID, ;
                Products.Product_ID ;
      FROM Employee, Products) AS CrossProd ;
     LEFT JOIN ( ;
      SELECT Product_ID, Employee_ID, ;
             SUM(Quantity) AS nUnitsSold, ;
             SUM(Quantity * Unit_Price) AS nTotalSales ;
        FROM Orders ;
          JOIN Order_Line_Items ;
            ON Orders.Order_ID = ;
               Order_Line_Items.Order_ID ;
        WHERE YEAR(Order_Date) = m.nYear ;
        GROUP BY Product_ID, Employee_ID ) ;
        AS AnnualSales ;
```

```
      ON CrossProd.Employee_ID = ;
         AnnualSales.Employee_ID ;
      AND CrossProd.Product_ID = AnnualSales.Product_ID ;
UNION ;
SELECT "Total" AS Product_ID, Employee.Employee_ID, ;
       m.nYear AS nYear, ;
       CAST(NVL(SUM(Quantity),0) as N(12)) ;
         AS nUnitsSold, ;
       NVL(SUM(Quantity * Unit_Price), $0) ;
         AS nTotalSales ;
   FROM Orders ;
     JOIN Order_Line_Items ;
       ON Orders.Order_ID = Order_Line_Items.Order_ID ;
      AND YEAR(Order_Date) = m.nYear ;
     RIGHT JOIN Employee ;
       ON Orders.Employee_ID = Employee.Employee_ID ;
   GROUP BY Employee.Employee_ID ;
ORDER BY 2, 1
```

The first query in this UNION uses an unusual technique, a cross-product or cartesian join. We need to ensure that a record is added for every employee/product combination, but there's no direct relationship between employees and products. So the query uses a derived table that contains one record for each employee/product combination; it then does an outer join with actual sales results to find the appropriate data to insert.

No parentheses with UNION

Although it wasn't really syntactically correct, earlier versions of VFP didn't object if queries in a UNION were enclosed in parentheses. In VFP 9, a single query in a UNION can be surrounded by parentheses, but putting parentheses around multiple queries in a UNION raises a new error, error 2196. Listing 22 shows a query that works in VFP 8, but fails in VFP 9, due to the new rule. The query is included in the session materials as UnionParens.PRG.

Listing 22. Putting parentheses around multiple queries in a UNION raises a new error.

```
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 )
```

According to the Fox team, using parentheses around multiple UNIONs can cause incorrect results.

Combining DISTINCT and ORDER BY

VFP allows you to order query results by any field from the source tables; fields in the ORDER BY list don't have to be in the field list. In VFP 9, this is no longer true for queries that use SELECT DISTINCT. For example, this query executes in VFP 8, but it raises error 1808 ("SQL: ORDER BY clause is invalid.") in VFP 9:

```
SELECT Distinct Customer_ID;
      FROM Orders ;
```

```
ORDER BY Order_Date
```

This behavior is affected by SET ENGINEBEHAVIOR. (See “Turn off new behavior” later in these notes.)

Optimization changes

VFP 9 includes a couple of changes to improve the performance of your queries as well as a new function that makes testing optimization easier. There's also one change that may be a problem for multi-lingual applications.

Fully optimize LIKE with "%"

The LIKE operator lets you compare strings. If a condition in a SQL command includes cField LIKE cString, the specified field is compared to the specified character string character by character. Unlike the = operator, if cString is shorter than cField, the two do not match, unless a wildcard character is used. The LIKE operator supports two wildcard characters—use "_" to represent a single unknown character and "%" to represent 0 or more unknown characters. For example, you can find all the customers in TasTrade whose names begin with the letter "P" using this query:

```
SELECT Customer_ID, Company_Name ;
  FROM Customer ;
  WHERE UPPER(Company_Name) LIKE "P%" ;
  INTO CURSOR PCompanies
```

Earlier versions of VFP could not fully optimize that query. LIKE "string%" expressions could only be partially optimized. VFP 9 fully optimizes such expressions. (Full optimization applies only when the % wildcard is at the end of the character string.)

My testing showed mixed results regarding the effect of this optimization. For many queries, versions using LIKE and = were equally fast in both VFP 8 and VFP 9. However, in one situation, the case when VFP takes a shortcut and simply filters the original table, optimization of LIKE made a significant difference. VFP does so for a query that involves a single table, has no calculated fields and is fully optimizable. Filtering the source table rather than creating an actual file on disk saves considerable time. Optimization of LIKE means the VFP engine can take this approach with some additional queries.

Better speed for TOP n

When you use the TOP n or TOP n PERCENT clause to return only a subset of the records that otherwise match the query conditions, VFP has to figure out which records are at the top of the list. When you specify the TOP n of a large set, that process can take considerable time. VFP 9 improves performance in that situation.

In my tests, I didn't see a difference until I was working with a very large table. Choosing the TOP 20 out of a table of nearly 75,000 records showed no performance difference. When I looked for the TOP 20 in a table of over a million records, however, VFP 9 finished in about one-third the time of VFP 8.

Along with making TOP n calculations faster, the behavior of TOP n queries has changed slightly. In earlier versions of VFP, a query with a TOP n clause could return more than n records due to ties in the data. VFP 9 never returns more than the exact number of records specified by the TOP n clause. (See "Turning off new behavior" later in these notes for the exception to this rule.) When there are ties, it appears that VFP chooses records in physical order from the group with the same value.

FOR with DELETED() is optimized

Earlier versions of VFP could not use any index that included a FOR condition (a filter) for optimization. VFP 9 makes a special exception for indexes with a filter of FOR DELETED() or FOR NOT DELETED(). Tags with either of those filters can be used for optimization.

Logging optimization results

The SYS(3054) function was introduced in VFP 5. It gives you information about how FoxPro is optimizing a SQL command. It's been improved several times and now offers a great deal of data about the optimization process. However, it's still hard to use SYS(3054) to gather information about query performance through an entire program or application.

Enter SYS(3092). This new function lets you direct SYS(3054) output to a log file. By itself, SYS(3054) can send output only to the active window or a variable. With SYS(3092), you can collect data about a whole series of queries and examine it at your leisure.

The syntax is:

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

The cFileName parameter specifies the name (including path) of the log file. Use lAdditive to specify whether an existing file is overwritten. The default is to overwrite an existing file.

To turn off logging and make the log file available for reading, pass the empty string as the cFileName parameter.

The function returns the name of the active log file. Note the the new log file is set before the value is returned, so to save the name of an old log file before changing it, you must call the function once with no additional parameters, and then call it again, passing the new value.

When you turn on logging with SYS(3092), the output from SYS(3054) is still echoed to the active window or stored to a specified variable.

Once you establish a log file with SYS(3092), use SYS(3054) as you normally would and run the queries you want to test. When you're done testing, reset SYS(3054) and then issue SYS(3092, "") to stop logging. You can then examine the log file to see your optimization results.

The information in the log file is most useful if you pass either 2 or 12 as the second parameter to SYS(3054). Added in VFP 7, those settings include the query itself in the output before reporting on optimization.

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, 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 another in a series of changes 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.

SELECT from buffered tables

Since VFP 3, FoxPro developers have been frustrated by the behavior of SELECT with buffering. When a buffered table is used in a query, VFP uses the actual table on disk, not the open buffered version. This means that query results don't reflect uncommitted changes to the data.

This behavior follows naturally from the normal behavior of queries. In general, whether a table is open or not, when it's listed in the FROM clause of a query, the VFP engine opens it again in a new work area.

In some situations, it would be really handy to be able to pull data from a buffered table with a query. In VFP 8 and earlier, you have to turn to Xbase commands (such as CALCULATE) instead.

VFP 9 gives you the option of looking at the buffered data. Add the new WITH (Buffering=.T.) clause to a query and it uses an available buffer rather than the table on disk. Listing 23 shows a query that counts the number of customers in each country using buffered data. (The session materials include QueryWithBuffering.PRG, which demonstrates the effect of the WITH clause.)

Listing 23. You can query buffered data using the new WITH (Buffering=.T.) clause.

```
SELECT Country, CNT(*) ;
   FROM Customer WITH (Buffering = .T.) ;
   GROUP BY Country ;
   INTO CURSOR BufferedCount
```

The WITH clause applies to a single table. If the query lists multiple tables for which you want to use buffered data, include a WITH statement for each.

One big warning. If you're using row buffering, the query commits the changes to the current row. This actually makes sense as the query moves the record pointer in the buffered table. In earlier versions, where queries operated against the data on disk, the record pointer in the buffer didn’t move, but when you query the buffer itself, the record pointer does move.

You can also control the behavior of queries with buffered tables globally. The new SET SQLBUFFERING command lets you specify whether queries draw from disk or from buffers by default. The WITH (Buffering = lExpr) clause overrides the current setting for a particular table and query. SET SQLBUFFERING is scoped to the data session. Use SET(“SQLBUFFERING”) to query the current setting.

Turn off new behavior

The significant changes to VFP's SQL engine introduced in VFP 8 caused problems for some existing applications. Rather than force developers to change working code or be stuck in VFP 7, the Fox team added the SET ENGINEBEHAVIOR command, which allows you to turn off the VFP 8 changes. While it's not a good idea to use it all the time, the command provides a flexible solution for existing applications.

Most of the changes to the SQL engine in VFP 9 are unlikely to cause compatibility problems. However, there are a few items that may be an issue for some applications, so the Fox team added a new setting to SET ENGINEBEHAVIOR.

As noted in "Better speed for TOP n" earlier in these notes, in VFP 9, a TOP n query now returns exactly n records; in the case of ties, it may discard some of the tied results.

When a query includes one of the aggregate functions (CNT(), SUM(), AVG(), MIN() or MAX()), but has no GROUP BY clause, VFP 9 always returns a single record. If no records meet the join and filter conditions, the result record has the null value for all fields. In earlier versions, such a query returned an empty result.

Finally, all fields listed in the ORDER BY clause of a query using SELECT DISTINCT must be included in the field list of the query.

To turn off these behaviors, SET ENGINEBEHAVIOR to 80 or 70. The default is SET ENGINEBEHAVIOR 90, which enables the new behaviors.

The bottom line

Many of the SQL changes in VFP 9 increase compatibility with the SQL-92 standard. They also provide more tools for manipulating your data as needed. While some of the changes aren't likely to have a strong impact on your day-to-day work, you'll probably find that others come in handy over and over. The ability to extract some data with a single query, where previously two queries were needed, not only tends to speed up the code, but also makes it easier to define views for those tasks.

My thanks to Aleksey Tsingauz of Microsoft, who reviewed these notes and made many useful suggestions. Thanks also to my co-authors for “New in Nine: Visual FoxPro’s Latest Hits,” who made additional suggestions. These notes are based on Chapter 8 of that book, available from www.hentzenwerke.com.