

Chapter 6

Extending Remote Views with SQL Pass Through

As you learned in Chapter 4, remote views make it very easy to access remote data. Remote views are actually wrappers for SQL pass through, handling the tasks of managing connections, detecting changes, formatting UPDATE commands and submitting them to the server. In this chapter, we're going to take an in-depth look at SQL pass through. We'll see how to connect to the server, submit queries, manage transactions and more.

There is no doubt about it, remote views offer the easiest way to access remote data. However, with ease of use comes less flexibility. Visual FoxPro contains another powerful mechanism for manipulating remote data called SQL pass through (SPT). SQL pass through provides all the functionality of remote views and more.

With SQL pass through you can:

- Execute queries other than SELECT.
- Access back-end-specific functionality.
- Fetch multiple result sets in a single call.
- Execute stored procedures.

However:

- There is no graphical user interface.
- You must manually manage connections.
- Result sets are read-only by default and must be configured to be updatable.

The flexibility that SQL pass through allows makes it a powerful tool. It is important for client/server developers to understand it thoroughly.

Connecting to the server

In order to use SQL pass through, you must make a connection to the server. Unlike remote views, Visual FoxPro does not manage the connection. The developer must manually make the connection, configure its behavior, and then disconnect when the connection is no longer needed. Connection management is very important because making a connection consumes substantial time and resources on the client and server.

There are two functions that are used to establish the connection with the remote server: `SQLConnect()` and `SQLStringConnect()`.

The SQLConnect() function

There are two ways to use the SQLConnect() function to connect to a remote data source. The first requires that you supply the name of a data source as defined in the ODBC Data Source Administrator applet of the control panel. The following example creates a connection to a remote server using the ODBCpubs DSN:

```
LOCAL hConn
hConn = SQLConnect("ODBCpubs", "sa", "")
```

The second way to use SQLConnect() is to supply the name of a Visual FoxPro connection that was created using the CREATE CONNECTION command. As you saw in Chapter 4, "Remote Views," the CREATE CONNECTION command stores the metadata that Visual FoxPro needs to connect to a remote data source. The following example creates a Visual FoxPro connection named VFPPUBS and then connects to the database described by the connection:

```
LOCAL hConn
CREATE DATABASE ctemp
CREATE CONNECTION vfppubs ;
  DATASOURCE "ODBCpubs" ;
  USERID "sa" ;
  PASSWORD ""
hConn = SQLConnect("vfppubs")
```

The SQLStringConnect() function

The other function that can be used to establish a connection to a remote data source is SQLStringConnect(). Unlike SQLConnect(), SQLStringConnect() requires a single parameter, a string of semicolon-delimited options that describes the remote data source and optional connections settings.

The valid options are determined by the requirements of the ODBC driver. Specific requirements for each ODBC driver can be found in that ODBC driver's documentation.

Table 1 lists some commonly used connection string options for SQL Server 7.0.

Table 1. Some common SQL Server 7.0 connection string options.

Option	Description
DSN	References an ODBC DSN.
Driver	Specifies the name of the ODBC driver to use.
Server	Specifies the name of the SQL Server to connect to.
UID	Specifies the login ID or username.
PWD	Specifies the password for the given login ID or username.
Database	Specifies the initial database to connect to.
APP	Specifies the name of the application making the connection.
WSID	The name of the workstation making the connection.
Trusted_Connection	Specifies whether the login is being validated by the Windows NT Domain.

Not all of the options listed in Table 1 have to be used for each connection. For instance, if you specify the *Trusted_Connection* option and connect to SQL Server using NT Authentication, there's no reason to use the *UID* and *PWD* options since SQL Server would invariably ignore them.

The following code demonstrates some examples of using `SQLStringConnect()`.



From this point forward, substitute the name of your server for the string <MyServer> in code examples.

```
LOCAL hConn
hConn = SQLStringConnect("Driver=SQL Server;Server=<MyServer>;"+
    "UID=sa;PWD=;Database=pubs")
hConn = SQLStringConnect("DSN=ODBCPubs;UID=sa;PWD=;Database=pubs")
hConn = SQLStringConnect("DSN=ODBCPubs;Database=pubs;Trusted_Connection=Yes")
```

Handling connection errors

Both the `SQLConnect()` and `SQLStringConnect()` functions return a connection handle. If the connection is established successfully, the handle will be a positive integer. If Visual FoxPro failed to make the connection, the handle will contain a negative integer. A simple call to the `AERROR()` function can be used to retrieve the error number and message. The following example traps for a failed connection and displays the error number and message using the Visual FoxPro `MESSAGEBOX()` function. **Figure 1** shows an example of the error message.



Visual FoxPro returns error 1526 for all errors against a remote data source. The fifth element of the array returned by `AERROR()` contains the remote data source-specific error.

```
#define MB_OKBUTTON      0
#define MB_STOPSIGNICON 16

LOCAL hConn
hConn = SQLConnect("ODBCPubs", "bad_user", "")
IF (hConn < 0)
    LOCAL ARRAY laError[1]
    AERROR(laError)
    MESSAGEBOX( ;
        laError[2], ;
        MB_OKBUTTON + MB_STOPSIGNICON, ;
        "Error " + TRANSFORM(laError[5]))
ENDIF
```

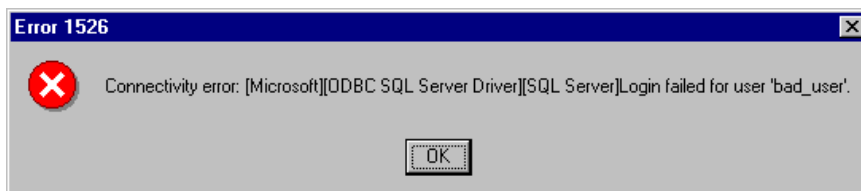


Figure 1. The error message returned from SQL Server 7.0 when trying to establish a connection using an unknown login.

Disconnecting

As mentioned previously, the developer is responsible for connection management when using SQL pass through. It is very important that a connection be released when it is no longer needed by the application because connections consume valuable resources on the server, and the number of connections may be limited by licensing constraints.

You break the connection to the remote data source using the `SQLDisconnect()` function. `SQLDisconnect()` takes one parameter, the connection handle created by a call to either `SQLConnect()` or `SQLStringConnect()`. `SQLDisconnect()` returns a 1 if the connection was correctly terminated and a negative value if an error occurred.

The following example establishes a connection to SQL Server 7.0 and then drops the connection:

```
LOCAL hConn,lnResult
*hConn = SQLStringConnect("Driver=SQL Server;Server=<MyServer>;"+
    UID=sa;PWD=;Database=pubs")
hConn = SQLConnect("ODBCpubs", "sa", "")
IF (hConn > 0)
    MESSAGEBOX("Connection established")
    lnResult = SQLDisconnect(hConn)
    IF lnResult < 0
        MESSAGEBOX("Disconnect failed")
    ENDIF && lnResult < 0
ENDIF && hConn > 0
```

If the parameter supplied to `SQLDisconnect()` is not a valid connection handle, Visual FoxPro will return a run-time error (#1466). Currently there is no way to determine whether a connection handle is valid without attempting to use it.



To disconnect all SQL pass through connections, you can pass a value of zero to `SQLDisconnect()`.

Accessing metadata

VFP has two SQL pass through functions that return information about the database you've connected to. The first, `SQLTables()`, returns a result set containing information about the tables and views in the database. The second, `SQLColumns()`, returns a result set containing information about a specific table or view.

The SQLTables() function

The following example demonstrates using SQLTables() to retrieve information about the user tables and views within the SQL Server demo database pubs. **Figure 2** shows a portion of the information returned in a VFP Browse window, and **Table 2** lists the definitions of the columns within the result set.

```
LOCAL hConn, lnResult
hConn = SQLConnect("odbcpubs", "sa", "")
lnResult = SQLTables(hConn, "TABLE", "VIEW")
SQLDisconnect(hConn)
```

Table_cat	Table_schem	Table_name	Table_type	Remarks
pubs	dbo	authors	TABLE	NULL
pubs	dbo	discounts	TABLE	NULL
pubs	dbo	dtproperties	TABLE	NULL
pubs	dbo	employee	TABLE	NULL
pubs	dbo	jobs	TABLE	NULL
pubs	dbo	pub_info	TABLE	NULL
pubs	dbo	publishers	TABLE	NULL
pubs	dbo	roysched	TABLE	NULL
pubs	dbo	sales	TABLE	NULL
pubs	dbo	stores	TABLE	NULL
pubs	dbo	titleauthor	TABLE	NULL
pubs	dbo	titles	TABLE	NULL
pubs	dbo	sysalternates	VIEW	NULL
pubs	dbo	sysconstraints	VIEW	NULL
pubs	dbo	syssegments	VIEW	NULL
pubs	dbo	titleview	VIEW	NULL
pubs	INFORMATION_SCHEMA	CHECK_CONSTRAINTS	VIEW	NULL
pubs	INFORMATION_SCHEMA	COLUMN_DOMAIN_USAGE	VIEW	NULL
pubs	INFORMATION_SCHEMA	COLUMN_PRIVILEGES	VIEW	NULL
pubs	INFORMATION_SCHEMA	COLUMNS	VIEW	NULL
pubs	INFORMATION_SCHEMA	CONSTRAINT_COLUMN_USAGE	VIEW	NULL

Figure 2. The results of calling SQLTables() on the pubs database.

Table 2. The description of the columns in the result set.

Column	Description
Table_cat	Object qualifier. However, In SQL Server 7.0, <i>Table_cat</i> contains the name of the database.
Table_schema	Object owner.
Table_name	Object name.
Table_type	Object type (TABLE, VIEW, SYSTEM TABLE or another data-store-specific identifier).
Remarks	A description of the object. However, SQL Server 7.0 does not return a value for <i>Remarks</i> .

The SQLColumns() function

SQLColumns() returns a result set containing information about each column in the specified table. This function returns different results depending on a third, optional parameter: "FOXPRO" or "NATIVE." The "NATIVE" option formats the result set with information specific to the remote data source, whereas specifying "FOXPRO" formats the result set with column information describing the Visual FoxPro cursor that contains the retrieved data. **Table 3** lists the columns that are returned by SQLColumns() using the "FOXPRO" option. **Table 4** lists the columns returned by SQLColumns() when using "NATIVE" and when attached to a SQL Server database. Different results are possible depending on the remote data source.

Table 3. A description of the columns returned by the FOXPRO option.

Column	Description
Field_name	Column name
Field_type	Visual FoxPro data type
Field_len	Column length
Field_dec	Number of decimal places

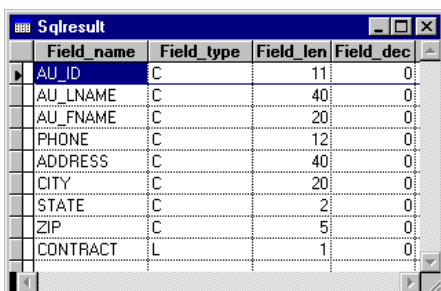
Table 4. A description of the columns returned from SQL Server using SQLColumns() and the NATIVE option.

Column	Description
Table_cat	SQL Server database name
Table_schema	Object owner
Table_name	Object name
Column_name	Column name
Data_type	Integer code for the ODBC data type
Type_name	SQL Server data type name
Column_size	Display requirements (character positions)
Buffer_length	Storage requirements (bytes)
Decimal_digits	Number of digits to the right of the decimal point
Num_prec_radix	Base for numeric data types
Nullable	Integer flag for nullability
Remarks	SQL Server always returns NULL
Column_def	Default value expression
SQL_data_type	Same as <i>Data_type</i> column
SQL_datetime_sub	Subtype for datetime data types
Character_octet_length	Maximum length of a character or integer data type
Ordinal_position	Ordinal position of the column (starting at 1)
Is_nullable	Nullability indicator as a string (YES NO)
SS_data_type	SQL Server data type code

The following example demonstrates using the SQLColumns() function to retrieve information about the authors table in the pubs database. **Figure 3** shows a Browse window containing a result set formatted with the FOXPRO option. **Figure 4** shows a subset of the columns returned by the NATIVE option.


```
LOCAL hConn, lnResult
hConn = SQLConnect("odbcpubs", "sa", "")
```

```
lnResult = SQLColumns(hConn, "authors", "FOXPRO")
BROWSE NORMAL && display the results
lnResult = SQLColumns(hConn, "authors", "NATIVE")
BROWSE NORMAL && display the results
SQLDisconnect(hConn)
```



Field_name	Field_type	Field_len	Field_dec
AU_ID	C	11	0
AU_LNAME	C	40	0
AU_FNAME	C	20	0
PHONE	C	12	0
ADDRESS	C	40	0
CITY	C	20	0
STATE	C	2	0
ZIP	C	5	0
CONTRACT	L	1	0

Figure 3. The results of calling `SQLColumns()` with the `FOXPRO` option.



Table_cat	Table_schm	Table_name	Column_name	Type_name	Column_size	Decimal_digs	Column_def	Ordinal_position	Is_nullable
pubs	dbo	authors	au_id	id	11		NULL, NULL	1	NO
pubs	dbo	authors	au_lname	varchar	40		NULL, NULL	2	NO
pubs	dbo	authors	au_fname	varchar	20		NULL, NULL	3	NO
pubs	dbo	authors	phone	char	12		NULL, (UNPadded)	4	NO
pubs	dbo	authors	address	varchar	40		NULL, NULL	5	YES
pubs	dbo	authors	city	varchar	20		NULL, NULL	6	YES
pubs	dbo	authors	state	char	2		NULL, NULL	7	YES
pubs	dbo	authors	zip	char	5		NULL, NULL	8	YES
pubs	dbo	authors	contract	bit	1		0, NULL	9	NO

Figure 4. A subset of the columns returned by `SQLColumns()` with the `NATIVE` option. (See Table 4 for a complete list of columns.)

Submitting queries

Most interactions with the remote server will be through the `SQLExec()` function. `SQLExec()` is the workhorse of the SQL pass through functions. You'll use it to submit `SELECT`, `INSERT`, `UPDATE` and `DELETE` queries, as well as calls to stored procedures. If the statement is successfully executed, `SQLExec()` will return a value greater than zero that represents the number of result sets returned by the server (more on multiple result sets later). A negative return value indicates an error. As discussed previously, you can use `AERROR()` to retrieve information about the error. It's also possible for `SQLExec()` to return a value of zero (0), but only if queries are being submitted asynchronously. We'll look at asynchronous queries in a later section.

Queries that return a result set

As with the `SQLTables()` and `SQLColumns()` functions, result sets returned by a query submitted using `SQLExec()` are stored in a Visual FoxPro cursor. Also like the `SQLTables()`

and `SQLColumns()` functions, the name of the result set will be `SQLRESULT` unless another name is specified in the call to `SQLExec()`.

For example, the following call to `SQLExec()` runs a `SELECT` query against the `authors` table in the `pubs` database.



From this point forward, examples may not include the code that establishes the connection.

```
lnResults = SQLExec(hConn, "SELECT * FROM authors")
```

To specify the name of the result set rather than accept the default, “`SQLRESULT`,” specify the name in the third parameter of the `SQLExec` statement. The following example uses the same query but specifies that the resultant cursor should be called `authors`:

```
lnResult = SQLExec(hConn, "SELECT * FROM authors", "authors")
```

Figure 5 shows the Data Session window with the single `authors` cursor open.

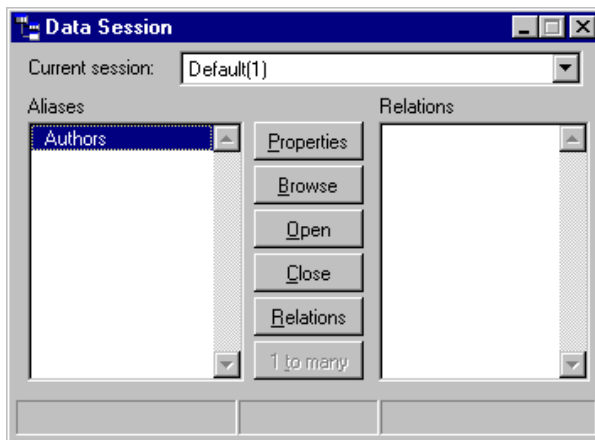


Figure 5. The Data Session window showing the single `authors` cursor.

Retrieving multiple result sets

As discussed in Chapter 3, “Introduction to SQL Server 7.0,” submitting a query to a remote server can be a very expensive operation. The server must parse the query and check for syntax errors, verify that all referenced objects exist, optimize the query to determine the best way to solve it, and then compile and execute. Luckily for us, Visual FoxPro and ODBC provide a means to gain an “economy of scale” when submitting queries. It is possible to submit multiple queries in a single call to `SQLExec()`, as in the following example:

```
lcSQL = "SELECT * FROM authors; SELECT * FROM titles"
lnResults = SQLExec(hConn, lcSQL)
```


SQLExec() returns a value (stored in lnResults in this example) containing the number of cursors returned.

Now you might be wondering what names Visual FoxPro assigns to each cursor. In the preceding example, the results of the first query (SELECT * FROM authors) will be placed into a cursor named Sqlresult. The results from the second query will be placed into a cursor named Sqlresult1. **Figure 6** shows the Data Session window with the two cursors open.

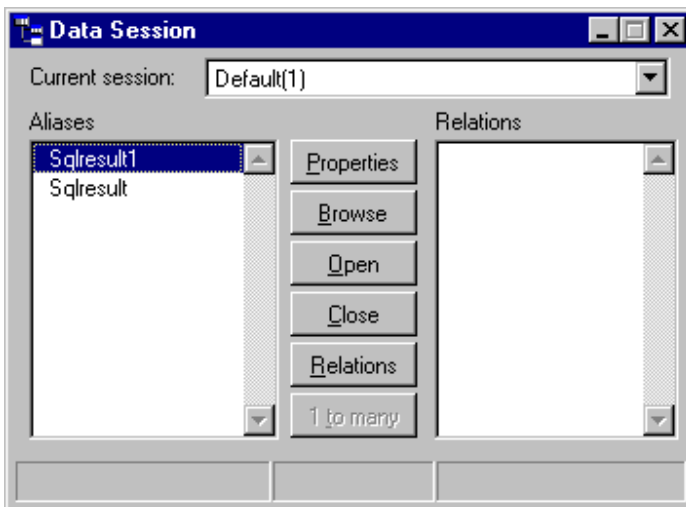


Figure 6. The Data Session window showing the two cursors *Sqlresult* and *Sqlresult1*.

Visual FoxPro's default behavior is to wait until SQL Server has returned all the result sets and then return the result sets to the application in a single action. Alternately, you can tell Visual FoxPro to return the result sets one at a time as each one is available. This behavior is controlled by a Connection property, *BatchMode*. If *BatchMode* is True (the default), Visual FoxPro returns all result sets at once; else, if False, Visual FoxPro returns the result sets one at a time.

Use the SQLSetProp() function to manipulate connection settings. The following example changes the *BatchMode* property to False, causing Visual FoxPro to return results sets one at a time:

```
lnResult = SQLSetProp(hConn, 'BatchMode', .F.)
```

As usual, a positive return result indicates success.

SQLSetProp() has a sibling, SQLGetProp(), that returns the current value of a connection property. The following code checks that the *BatchMode* property was set correctly:

```
llBatch = SQLGetProp(hConn, 'BatchMode')
```

When *BatchMode* is False, Visual FoxPro automatically returns only the first result set. The developer must request that Visual FoxPro return additional result sets by calling the

SQLMoreResults() function. SQLMoreResults() returns zero (0) if the next result set is not ready, one (1) if it is ready, two (2) if there are no more result sets to retrieve, or a negative number if an error has occurred.

The following example demonstrates the SQLMoreResults() function. In this example, we're going to retrieve information about a specific book by submitting queries against the titles, authors, titleauthor and sales tables.

```
*-- Get information about the book
lcSQL = "SELECT * FROM titles WHERE title_id = 'TC7777'" + ";"

*-- Retrieve the authors
lcSQL = lcSQL + ;
"SELECT * " + ;
"FROM authors INNER JOIN titleauthor " + ;
"ON authors.au_id = titleauthor.au_id " + ;
"WHERE titleauthor.title_id = 'TC7777'"

*-- Retrieve sales information
lcSQL = lcSQL + "SELECT * FROM sales WHERE title_id = 'TC7777'"

lnResult = SQLSetProp(hConn, "BatchMode", .F.)
lnResult = SQLExec(hConn, lcSQL, 'TitleInfo')

DO WHILE .T.
  lnResult = SQLMoreResults(hConn)
  DO CASE
    CASE lnResult < 0
      *-- Error condition

    CASE lnResult = 0
      *-- No result sets are ready

    CASE lnResult = 2
      *-- All result sets have been retrieved
      EXIT

    OTHERWISE
      *-- Process retrieved result set
  ENDCASE
ENDDO
```

It is important to realize that SQLMoreResults() must continue being called until it returns a two (2), meaning no more result sets. If any other SQL pass through function is issued before SQLMoreResults() returns 2, Visual FoxPro will return the error shown in **Figure 7**.



The preceding statement is not entirely true. You can issue the SQLCancel() function to terminate any waiting result sets, but we haven't introduced it yet.



Figure 7. The results of trying to issue another SQL pass through function while processing result sets in non-batch mode.

Queries that modify data

INSERT, UPDATE and DELETE queries are submitted in the same way as SELECT queries. The following example increases the price of all books in the titles table of the pubs database by 10 percent:

```
InResult = SQLExec(hConn, "UPDATE titles SET price = price * 1.1")
```

In this example, SQLExec() executes a data modification query rather than a SQL SELECT statement. Therefore, it returns a success indicator (1 for successful execution or a negative number in the event of an error), rather than the number of result sets. If the query successfully updates zero, one, or one million rows, SQLExec() will return a value of one. (A query is considered successful if the server can parse and execute it.)

To determine the number of rows updated, use the SQL Server global variable @@ROWCOUNT, which performs the same function as Visual FoxPro's _TALLY global variable. After executing a query, @@ROWCOUNT contains the number of rows affected by the query. The value of @@ROWCOUNT can be retrieved by issuing a SELECT query:

```
InResult = SQLExec(hConn, "SELECT @@ROWCOUNT AS AffectedRows", "status")
```

Note that the value of @@ROWCOUNT is returned as a column named "AffectedRows" in the first (and only) row of a cursor named "Status," not to the variable InResult.

Unlike _TALLY, @@ROWCOUNT is not truly global. It is one of several variables that are scoped to the connection. Therefore, the value of @@ROWCOUNT must be retrieved on the same connection that executed the query. If you execute the query on one connection and retrieve the value of @@ROWCOUNT from another connection, the result will not be accurate.

Also, @@ROWCOUNT is reset after each statement. If you submit multiple queries, @@ROWCOUNT will contain the affected row count for the last query executed.

Parameterized queries

You previously read about using parameters in views to filter the query. You can use this same mechanism with SQL pass through.

Using a parameterized query might seem unnecessary at first. After all, since you pass the query as a string, you have complete control over its creation. Consider the following example:

```
FUNCTION GetTitleInfo(tcTitleID, tcCursor)
LOCAL lcQuery, hConn
llcQuery = "SELECT * FROM titles WHERE title_id = '" + tcTitleID + "'"
hConn = SQLConnect("odbcpubs", "sa", "")
lnResult = SQLExec(hConn, lcQuery, tcCursor)
SQLDisconnect(hConn)
RETURN .T.
```

Creating the query using the technique from the previous example works in most situations. However, there are situations where using a parameterized query makes more sense. For example, when different back ends impose different requirements for specifying literal values, it is easier to allow Visual FoxPro to handle the conversion. Consider dates. Visual FoxPro requires date literals to be specified in the form {^1999-12-31}. SQL Server, on the other hand, does not recognize {^1999-12-31} as a date literal. Instead you would have to use a literal similar to '12/31/1999' or '19991231' (the latter being preferred).

The following code shows how the same query would be formatted for Visual FoxPro and SQL Server back ends:

```
*-- If accessing Visual FoxPro using the ODBC driver
lcQuery = ;
"SELECT * " + ;
"FROM titles " + ;
"WHERE pubdate BETWEEN {^1998-01-01} AND {^1998-12-31}"

*-- If accessing SQL Server
lcQuery = ;
"SELECT * " + ;
"FROM titles " + ;
"WHERE pubdate BETWEEN '19980101' AND '19981231'"
```

In this situation, Visual FoxPro converts the search arguments to the proper format automatically. The following example demonstrates this:

```
LOCAL ldStart, ldStop, lcQuery
ldStart = {^1998-01-01}
ldStop = {^1998-12-31}
lcQuery = ;
"SELECT * " + ;
"FROM titles " + ;
"WHERE pubdate BETWEEN ?ldStart AND ?ldStop"
```

The preceding query would work correctly against both Visual FoxPro and SQL Server.

There are other data types that also benefit from the use of parameterization. Visual FoxPro's Logical vs. SQL Server's Bit is another example. A literal TRUE is represented in Visual FoxPro as .T., while in Transact-SQL it is 1.

The advantage of parameterization

Parameterized queries provide an additional benefit: Parameterized queries execute more quickly than non-parameterized queries when the query is called repeatedly with different parameters. This performance benefit occurs because SQL Server does not have to parse, optimize and compile the query each time it is called—instead, it can reuse the existing execution plan with the new parameter values.

To demonstrate, we'll use the SQL Server Profiler, a utility that ships with SQL Server 7.0. SQL Server Profiler, described in greater detail in Chapter 8, "Errors and Debugging," is one of the best tools available for debugging and investigation. It logs events that occur on the server (such as the submission of a query or calling a stored procedure) and collects additional information.



SQL Server 6.5 has a similar utility called SQLTrace.

Figure 8 shows the output from the SQL Server Profiler for the following query, and **Figure 9** shows the output for the second one:

```
LOCAL llTrue,lnResult
lnResult = SQLExec(hConn, "SELECT * FROM authors WHERE contract = 1")
llTrue = .T.
lnResult = SQLExec(hConn, "SELECT * FROM authors WHERE contract = ?llTrue")
```

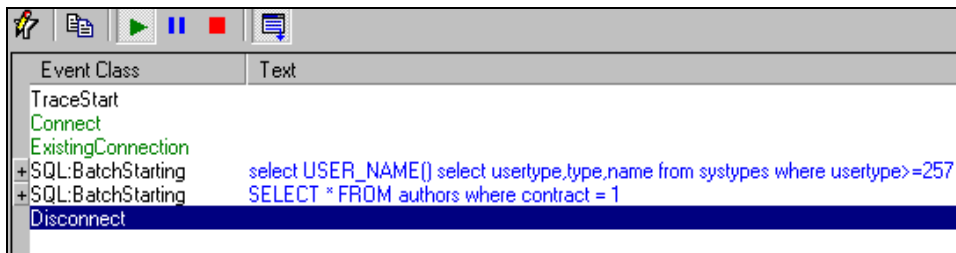


Figure 8. The SQL Server Profiler output for a non-parameterized query.

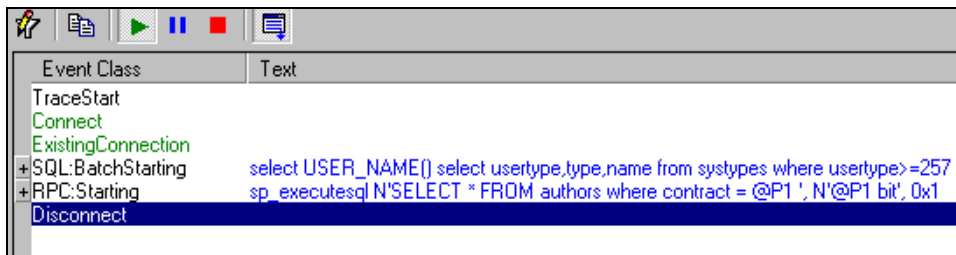


Figure 9. The SQL Server Profiler output for a parameterized query.

There is an important difference between how these two queries were submitted to SQL Server. As expected, the first query was passed straight through. SQL Server had to parse,

optimize, compile and execute the query. The next time the same query is submitted, SQL Server will have to parse, optimize and compile the query again before executing it.

The second query was handled quite differently. When Visual FoxPro (actually, ODBC) submitted the query, the *sp_executesql* stored procedure was used to identify the search arguments to SQL Server. The following is an excerpt from the SQL Server Books Online:

“*sp_executesql* can be used instead of stored procedures to execute a Transact-SQL statement a number of times when the change in parameter values to the statement is the only variation. Because the Transact-SQL statement itself remains constant and only the parameter values change, the Microsoft® SQL Server™ query optimizer is likely to reuse the execution plan it generates for the first execution.”

SQL Server will take advantage of this knowledge (the search arguments) by caching the execution plan (the result of parsing, optimizing and compiling a query) instead of discarding the execution plan, which is the normal behavior. The next time the query is submitted, SQL Server can reuse the existing execution plan, but with a new parameter value.

There is a tradeoff—calling a stored procedure has a cost, so you should not blindly write all your queries using parameters. However, the cost is worth incurring if the query is executed repeatedly. There are no magic criteria to base your decision on, but some of the things to consider are the number of times the query is called and the length of time between calls.

Making SQL pass through result sets updatable

By default, the cursor created to hold the results of a SQL pass through query is read-only. Actually, changes can be made to the data within the cursor, but Visual FoxPro won't do anything with the changes. This may sound familiar, as it's the same behavior exhibited by a non-updatable view. As it turns out, the same options that make a view updatable will also work on a cursor created by a SQL pass through query. The following example retrieves all the authors from the authors table and makes the *au_fname* and *au_lname* columns updatable:

```
InResult = SQLExec(hConn, "SELECT * FROM authors", "authors")
CURSORSETPROP("TABLES", "dbo.authors", "authors")
CURSORSETPROP("KeyFieldList", "au_id", "authors")
CURSORSETPROP("UpdatableFieldList", "au_lname, au_fname", "authors")
CURSORSETPROP("UpdateNameList", ;
  "au_id dbo.authors.au_id, " + ;
  "au_lname dbo.authors.au_lname, " + ;
  "au_fname dbo.authors.au_fname, " + ;
  "phone dbo.authors.phone, " + ;
  "address dbo.authors.address, " + ;
  "city dbo.authors.city, " + ;
  "state dbo.authors.state, " + ;
  "zip dbo.authors.zip, " + ;
  "contract dbo.authors.contract", "authors")
CURSORSETPROP("SendUpdates", .T., "authors")
```

Each property plays an important role in the creation of the commands sent to the server. Visual FoxPro will create an INSERT, UPDATE or DELETE query based on the operations performed on the cursor. The UpdatableFieldList property tells Visual FoxPro which columns

it needs to track changes to. The `Tables` property supplies the name of the remote table, and the `UpdateNameList` property has the name of the column in the remote table for each column in the cursor. `KeyFieldList` contains a comma-delimited list of the columns that make up the primary key. Visual FoxPro uses this information to construct the `WHERE` clause of the query. The last property, `SendUpdates`, provides a safety mechanism. Unless `SendUpdates` is marked `TRUE`, Visual FoxPro will not send updates to the server.

There are two other properties that you may want to include when making a cursor updatable. The first, `BatchUpdateCount`, controls the number of update queries that are submitted to the server at once. The default value is one (1), but increasing this property can improve performance. SQL Server will parse, optimize, compile and execute the entire batch of queries at the same time. The second parameter, `WhereType`, controls how Visual FoxPro constructs the `WHERE` clause used by the update queries. This also affects how conflicts are detected. Consult the Visual FoxPro online Help for more information on the `WhereType` cursor property.

Calling stored procedures

One of the things you don't normally do with a remote view is call a stored procedure. For that, you typically use SQL pass through. Calling a stored procedure via SQL pass through is just like submitting any other query: The `SQLExec()` function does all the work.

The following example demonstrates calling a SQL Server stored procedure. The stored procedure being called, `sp_helpdb`, returns information about the databases residing on the attached server. Note that we have the same ability to rename the result set returned by the query/stored procedure.

```
lnResult = SQLExec(hConn, "EXECUTE sp_helpdb", "dbinfo")
```

The preceding example uses the Transact-SQL `EXECUTE` command to call the stored procedure. You can also call stored procedures using the ODBC escape syntax, as demonstrated in the following example:

```
lnResult = SQLExec(hConn, "{CALL sp_helpdb}")
```

Using the ODBC escape syntax offers two small advantages. First, ODBC will automatically convert the statement to the format required by the back end you are working with—as long as that back end supports directly calling stored procedures. Second, it is an alternate way to work with `OUTPUT` parameters.

Handling input and output parameters

Stored procedures, like Visual FoxPro's procedures and functions, can accept parameters. An example is `sp_helpprotect`, which returns information about user permissions applied to a specific database object. The following code calls the `sp_helpprotect` stored procedure to obtain information about user permissions applied to the `authors` table. The result set will contain all the users and roles that have been given explicit permissions on the `authors` table, and whether those permissions were granted or denied.

```
lnResult = SQLExec(hConn, "EXECUTE sp_helprotect 'authors'")
```

Using the ODBC calling convention is slightly different from calling a Visual FoxPro function, as shown here:

```
lnResult = SQLExec(hConn, "{CALL sp_helprotect ('authors')}")
```

Additional parameters are separated by commas:

```
lnResult = SQLExec(hConn, "EXECUTE sp_helprotect 'authors', 'guest'")
lnResult = SQLExec(hConn, "{CALL sp_helprotect ('authors', 'guest')}")
```

Output parameters

There is another way to get information from a stored procedure: output parameters. An output parameter works the same way as a parameter passed to a function by reference in Visual FoxPro: The stored procedure alters the contents of the parameter, and the new value will be available to the calling program.

The following Transact-SQL creates a stored procedure in the Northwind database that counts the quantity sold of a particular product within a specified date range:

```
LOCAL lcQuery, lnResult
lcQuery = ;
"CREATE PROCEDURE p_productcount " + ;
"@ProductId INT, " + ;
"@StartDate DATETIME, " + ;
"@EndDate DATETIME, " + ;
"@QtySold INT OUTPUT " + ;
"AS " + ;
"SELECT @QtySold = SUM(od.Quantity) " + ;
"FROM Orders o INNER JOIN [Order Details] od " + ;
"ON o.OrderId = od.OrderId " + ;
"WHERE od.ProductId = @ProductId " + ;
"AND o.OrderDate BETWEEN @StartDate AND @EndDate "

*-- hConn must be a connection to the Northwind database
lnResult = SQLExec(hConn, lcQuery)
```

The stored procedure accepts four parameters: the ID of the product, the start and end points for a date range, and an output parameter to return the total quantity sold.

The following example shows how to call the stored procedure and pass the parameters:

```
LOCAL lnTotalCnt, lcQuery
lnTotalCnt = 0
lcQuery = "EXECUTE p_ProductCount 72, '19960701', '19960801', ?@lnTotalCnt"
lnResult = SQLExec(hConn, lcQuery)
```

You can also call the p_ProductCount procedure using ODBC escape syntax, as in the following code:


```
lcQuery = "{CALL p_productcount (72, '19960701', '19960801', ?@lnTotalCnt)}"
lnResult = SQLExec(hConn, lcQuery)
```



Because SQL Server returns result codes and output parameters in the last packet sent from the server, output parameters are not guaranteed to be available until after the last result set is returned from the server—that is, until `SQLExec()` returns a one (1) while in Batch mode or `SQLMoreResults()` returns a two (2) in Non-batch mode.

Transaction management

A transaction groups a collection of operations into a single unit of work. If any operation within the transaction fails, the application can cause the data store to undo (that is, reverse) all the operations that have already been completed, thus keeping the integrity of the data intact.

Transaction management is a powerful tool, and the Visual FoxPro community was pleased to see its introduction into Visual FoxPro.

In Chapter 3, “Introduction to SQL Server 7.0,” we looked at transactions within SQL Server and identified two types: implicit (or Autocommit) and explicit. To review, implicit transactions are individual statements that commit independently of other statements in the batch. In other words, the changes made by one statement are not affected by the success or failure of a statement that executes later. The following example demonstrates transferring funds from a savings account to a checking account:

```
lnResult = SQLExec(hConn, ;
  "UPDATE account " + ;
  "SET balance = balance - 100 " + ;
  "WHERE ac_num = 14356")

lnResult = SQLExec(hConn, ;
  "UPDATE account " + ;
  "SET balance = balance + 100 " + ;
  "WHERE ac_num = 45249")
```

Even if the two queries are submitted in the same `SQLExec()` call, as in the following example, the two queries commit independently of each other:

```
lnResult = SQLExec(hConn, ;
  "UPDATE account " + ;
  "SET balance = balance - 100 " + ;
  "WHERE ac_num = 14356" + ;
  ";" + ;
  "UPDATE account " + ;
  "SET balance = balance + 100 " + ;
  "WHERE ac_num = 45249")
```

Each query is independent of the other. If the second fails, nothing can be done to undo the changes made by the first except to submit a correcting query.

On the other hand, an explicit transaction groups multiple operations and allows the developer to undo all changes made by all operations in the transaction if any one operation fails.

In this section, we're going to look at the SQL pass through functions that manage transactions: `SQLSetProp()`, `SQLCommit()` and `SQLRollback()`.

SQL pass through doesn't have a function to start an explicit transaction. Instead, explicit transactions are started by setting the connection's Transactions property to a two (2) or `DB_TRANSMANUAL` (from `Foxpro.h`). The following example shows how to use the `SQLSetProp()` function to start a manual (Visual FoxPro term) or explicit (SQL Server term) transaction:

```
#include FOXPRO.h
lnResult = SQLSetProp(hConn, "TRANSACTIONS", DB_TRANSMANUAL)
```

Enabling manual transaction does not actually start a transaction. The transaction actually starts only when the first query is submitted. After that, all queries submitted on the connection will participate in the transaction until the transaction is terminated. You will see exactly how this works in Chapter 11, "Transactions." Regardless, if everything goes well and no errors occur, you can commit the transaction with the `SQLCommit()` function:

```
lnResult = SQLCommit(hConn)
```

If something did go wrong, the transaction can be rolled back and all operations reversed with the `SQLRollback()` function:

```
lnResult = SQLRollback(hConn)
```

Manual transactions can only be disabled by calling `SQLSetProp()` to set the Transactions property back to 1. If you do not reset the Transactions property to 1, the next query submitted on the connection automatically causes another explicit transaction to be started.

Taking all that into account, the original example can be rewritten as follows:

```
#include FOXPRO.h
...
lnResult = SQLSetProp(hConn, "TRANSACTIONS", DB_TRANSMANUAL)
lnResult = SQLExec(hConn, ;
    "UPDATE account " + ;
    "SET balance = balance - 100 " + ;
    "WHERE ac_num = 14356" + ;
    ";" + ;
    "UPDATE account " + ;
    "SET balance = balance + 100 " + ;
    "WHERE ac_num = 45249")

IF (lnResult != 1)
    SQLRollback(hConn)

    *-- Relay error message to the user
ELSE
    SQLCommit(hConn)
```

```

ENDIF

SQLSetProp(hConn, "TRANSACTIONS", 1)
RETURN (lnResult = 1)

```

The code in the preceding example wraps the UPDATE queries within the explicit transaction and handles an error by rolling back any changes that may have occurred.

Binding connections

Sometimes it's necessary for two or more connections to participate in the same transaction. This scenario can occur when dealing with components in a non-MTS environment. To accommodate this need, SQL Server provides the ability to bind two or more connections together. Once bound, the connections will participate in the same transaction.

If multiple connections participate in one transaction, any of the participating connections can begin the transaction, and any participating connection can end the transaction.

Connection binding is accomplished by using two stored procedures: *sp_getbindtoken* and *sp_bindsession*. First, execute *sp_getbindtoken* against the first connection to obtain a unique identifier (the bind token, as a string) for the connection. Next, pass the bind token to *sp_bindsession*, which is executed against another connection. The second function binds the two connections. The following example demonstrates the entire process:

```

LOCAL hConn1, hConn2, hConn3, lnResult, lcToken
lcToken = ""
hConn1 = SQLConnect("odbcpubs", "sa", "")
hConn2 = SQLConnect("odbcpubs", "sa", "")
hConn3 = SQLConnect("odbcpubs", "sa", "")

lnResult = SQLExec(hConn1, "EXECUTE sp_getbindtoken ?@lcToken")
lnResult = SQLExec(hConn2, "EXECUTE sp_bindsession ?lcToken")
lnResult = SQLExec(hConn3, "EXECUTE sp_bindsession ?lcToken")

SQLDisconnect(hConn1)
SQLDisconnect(hConn2)
SQLDisconnect(hConn3)

```

In the example, three connections are established to the server. In the first call to *sp_getbindtoken*, you get the bind token. You must use the ? and @ symbols with the lcToken variable because the binding token returns through an OUTPUT parameter. You then pass the bind token to the second and third connections by calling *sp_bindsession*.

Asynchronous processing

So far, every query that we've sent to the server was sent synchronously. Visual FoxPro paused until the server finished processing the query and returned the result set to Visual FoxPro. There are times, however, when you may not want Visual FoxPro to pause while the query is running. For example, you may want to provide some feedback to the user to indicate that the application is running and has not locked up, or you may want to provide the ability to cancel a query mid-stream. To prevent Visual FoxPro from pausing, submit the query asynchronously. Just remember that this approach makes the developer responsible for determining when the query processing is finished.

Switching to asynchronous processing is not complicated. There is a connection property, *Asynchronous*, that determines the mode. If *Asynchronous* is set to *FALSE* (the default), all queries will be sent synchronously. Note that *Asynchronous* is a Visual FoxPro connection property and therefore is scoped to the connection.

The following example demonstrates making a connection and then using the `SQLSetProp()` function to change to asynchronous mode:

```
hConn = SQLConnect("odbcpubs", "sa", "")
lnResult = SQLSetProp(hConn, "ASYNCHRONOUS", .T.)
```

There is absolutely no difference between submitting a query in synchronous mode and submitting a query in asynchronous mode. There is, however, a difference in the way you detect that the query has completed. If a query is submitted in synchronous mode, `SQLExec()` will return a positive value indicating the number of result sets returned or a negative value indicating an error. In asynchronous mode, that still holds true, but `SQLExec()` can return a zero (0) indicating that the query is still being processed. It is up to the developer to poll the server to determine when the query has been completed.

Polling the server is quite easy. It requires resubmitting the query again. ODBC and the server realize that the query is not being resubmitted, that this is merely a status check. In fact, you can simply pass an empty string for the subsequent `SQLExec()` calls. Regardless, the following example shows one way to structure this process:

```
LOCAL llDone, lnResult
llDone = .F.
DO WHILE !llDone
    lnResult = SQLExec(hConn, "EXECUTE LongQuery")
    llDone = (lnResult != 0)
ENDDO
```

The loop will stay engaged as long as `SQLExec()` returns a zero (0) identifying the query as still being processed.

In the preceding example, the stored procedure being called does not actually run any queries. The stored procedure `LongQuery` simply uses the Transact-SQL `WAITFOR` command with the `DELAY` option to pause for a specific period of time (two minutes in this case) before proceeding. The code for `LongQuery` is shown here:

```
lcQuery = ;
[CREATE PROCEDURE longquery AS ] + ;
    [WAITFOR DELAY '00:02:00']
*-- hConn should be a connection to the pubs database
lnResult = SQLExec(hConn, lcQuery)
```

The following program demonstrates calling the `LongQuery` stored procedure in asynchronous mode and trapping the Escape key, which is the mechanism provided to terminate the query:

```
LOCAL hConn, lcQuery, llCancel, lnResult
lcQuery = "EXECUTE LongQuery"
```

```

hConn = SQLConnect("odbcpubs", "sa", "")
SQLSetProp(hConn, "ASYNCHRONOUS", .T.)

SET ESCAPE ON
ON ESCAPE llCancel = .T.

WAIT WINDOW "Press Esc to cancel the query" NOWAIT NOCLEAR

llCancel = .F.
lnResult = 0
DO WHILE (!llCancel AND lnResult = 0)
    lnResult = SQLExec(hConn, lcQuery)
    DOEVENTS
ENDDO

WAIT CLEAR

IF (llCancel AND lnResult = 0)
    WAIT WINDOW "Query being cancelled..." NOWAIT NOCLEAR
    SQLCancel(hConn)
    WAIT WINDOW "Query cancelled by user"
ELSE
    IF (lnResult > 0)
        WAIT WINDOW "Query completed successfully!"
    ELSE
        WAIT WINDOW "Query aborted by error"
    ENDIF
ENDIF

SQLDisconnect(hConn)

```

If the user presses the Escape key, the ON ESCAPE mechanism sets the local variable *llCancel* to TRUE, terminating the WHILE loop. The next IF statement tests whether the query was canceled before the results were returned to Visual FoxPro. If so, the SQLCancel() function is used to terminate the query on the server.

Asynchronous queries are a bit more complicated to code, but they permit the user to cancel a query, which adds polish to your applications.

Connection properties revisited

Throughout this chapter you have seen connection properties used to configure the behavior of SQL pass through. Visual FoxPro has two ways to configure default values for connection properties. The first and perhaps easiest to use is the Remote Data tab of the Options dialog (see **Figure 10**). Note that unless the defaults are written to the registry using the Set As Default button, changes made using the Remote Data tab will affect all new connections but will only persist until the current Visual FoxPro session is terminated.

SQLSetProp() can also be used to configure connection defaults through the special connection handle zero (0)—the environment handle. Unlike the Remote Data tab of the Options dialog, the changes made to the environment handle using SQLSetProp() cannot be made to persist beyond the current session. However, you can configure connection properties explicitly using SQLSetProp() as part of your application startup routine.

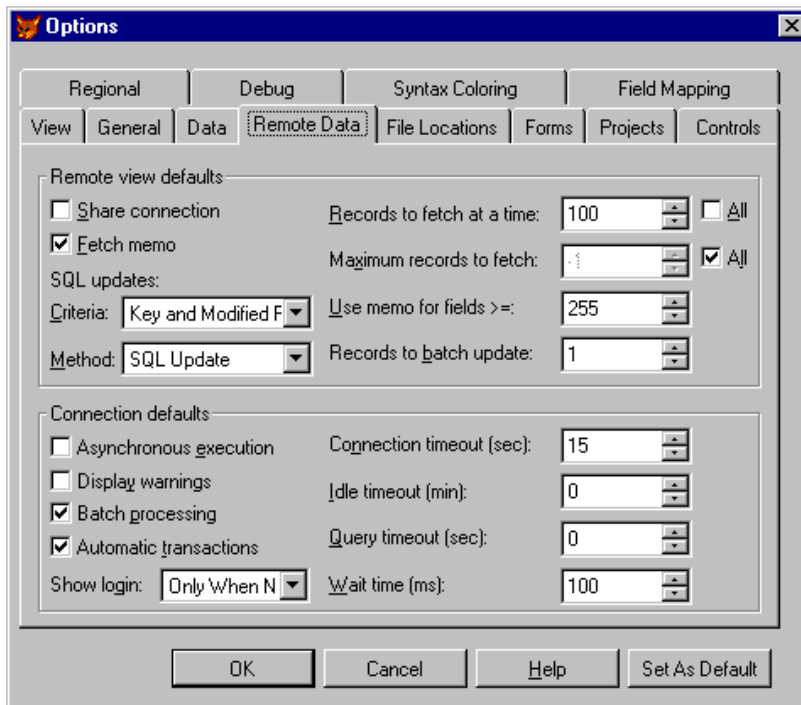


Figure 10. The Remote Data tab of the Options dialog.

Other connection properties

Earlier in this chapter, we explored several connection properties: Batch Mode, Asynchronous and Transactions. There are many more connection properties. The Help for `SQLSetProp()` lists 17 different properties, some of which are seldom, if ever, used (for example, `ODBCdbc`). However, some are worthy of mention.

The `DispLogin` property

The `DispLogin` connection property controls whether Visual FoxPro prompts the user with the ODBC login dialog. Table 5 lists the possible values and their descriptions.

Table 5. The legal values for the `DispLogin` connection property.

Numeric value	Constant from <code>Foxpro.h</code>	Description
1 (Default)	<code>DB_PROMPTCOMPLETE</code>	Visual FoxPro will display the ODBC connection dialog if any required connection information is missing.
2	<code>DB_PROMPTALWAYS</code>	Client is always prompted with the ODBC connection dialog.
3	<code>DB_PROMPTNEVER</code>	Client is never prompted.

The following example demonstrates the effect of DispLogin.



The following example will not work correctly if your ODBC DSN is configured for NT Authentication.

```
InResult = SQLSetProp(0, 'DISPLOGIN', 1) && Set to the default value
hConn = SQLConnect("odbcpubs", "bad_user", "")
```

You should be prompted with the ODBC connection dialog similar to **Figure 11**.



Figure 11. Visual FoxPro prompting with the ODBC connection dialog due to missing login information.

If you execute the following code, you'll get a different result. Visual FoxPro will not prompt with the ODBC connection dialog, and SQLConnect() will return a -1.

```
InResult = SQLSetProp(0, 'DISPLOGIN', 3) && never prompt
hConn = SQLConnect("odbcpubs", "bad_user", "")
```

It is highly recommended that you always use the “never prompt” option for this property, as you should handle all logins to the server through your own code instead of this dialog.

The ConnectionTimeout property

The ConnectionTimeout property specifies the amount of time (in seconds) that Visual FoxPro waits while trying to establish a connection with a remote data source. Legal values are 0 to 600. The default is 15 seconds. You may want to adjust this value upward when connecting to a server across a slow connection.

The QueryTimeout property

The QueryTimeout property specifies the amount of time (in seconds) that Visual FoxPro waits for a query to be processed. Legal values are 0 to 600. The default is 0 seconds (wait indefinitely). This property could be used as a governor to terminate long-running queries before they tie up important server resources.

The IdleTimeOut property

The IdleTimeOut property specifies the amount of time (in seconds) that Visual FoxPro will allow a previously active connection to sit idle before being automatically disconnected. The default is 0 seconds (wait indefinitely).

Remote views vs. SQL pass through

Developers have strong opinions about the usefulness of remote views. Many members of the Visual FoxPro community feel that remote views carry too much overhead and that the best performance is achieved by using SQL pass through. In this section, we'll again use the SQL Profiler to capture and evaluate the commands submitted to SQL Server.



Refer to the topic "Monitoring with SQL Server Profiler" in the SQL Server Books Online for more information on using the SQL Server Profiler.

SQL pass through

Figure 12 shows the commands sent to SQL Server for the following query:

```
lnResult = SQLExec(hConn, ;
  "UPDATE authors " + ;
  "SET au_lname = 'White' " + ;
  "WHERE au_id = '172-32-1176'")
```



Figure 12. The commands captured by SQL Profiler for a simple UPDATE query.

Notice that nothing special was sent to the server—the query was truly “passed through” without any intervention by Visual FoxPro. When SQL Server received the query, it proceeded with the normal process: parse, name resolution, optimize, compile and execute.

Figure 13 shows the commands sent to SQL Server for a query that uses parameters. This is the same query as before, except this time we're using parameters in place of the literals 'White' and '172-32-1176.'

```
LOCAL lcNewName, lcAuID
lcNewName = "White"
lcAuID = "172-32-1176"
lnResult = SQLExec(hConn, ;
  "UPDATE authors " + ;
  "SET au_lname = ?lcNewName " + ;
  "WHERE au_id = ?lcAuID")
```



```
DBSetProp('V_AUTHORS', 'View', 'UseMemoSize', 255)
DBSetProp('V_AUTHORS', 'View', 'FetchSize', 100)
DBSetProp('V_AUTHORS', 'View', 'MaxRecords', -1)
DBSetProp('V_AUTHORS', 'View', 'Tables', 'dbo.authors')
DBSetProp('V_AUTHORS', 'View', 'Prepared', .F.)
DBSetProp('V_AUTHORS', 'View', 'CompareMemo', .T.)
DBSetProp('V_AUTHORS', 'View', 'FetchAsNeeded', .F.)
DBSetProp('V_AUTHORS', 'View', 'FetchSize', 100)
DBSetProp('V_AUTHORS', 'View', 'Comment', "")
DBSetProp('V_AUTHORS', 'View', 'BatchUpdateCount', 1)
DBSetProp('V_AUTHORS', 'View', 'ShareConnection', .F.)

!* Field Level Properties for V_AUTHORS
* Props for the V_AUTHORS.au_id field.
DBSetProp('V_AUTHORS.au_id', 'Field', 'KeyField', .T.)
DBSetProp('V_AUTHORS.au_id', 'Field', 'Updatable', .F.)
DBSetProp('V_AUTHORS.au_id', 'Field', 'UpdateName', 'dbo.authors.au_id')
DBSetProp('V_AUTHORS.au_id', 'Field', 'DataType', "C(11)")
* Props for the V_AUTHORS.au_lname field.
DBSetProp('V_AUTHORS.au_lname', 'Field', 'KeyField', .F.)
DBSetProp('V_AUTHORS.au_lname', 'Field', 'Updatable', .T.)
DBSetProp('V_AUTHORS.au_lname', 'Field', 'UpdateName', 'dbo.authors.au_lname')
DBSetProp('V_AUTHORS.au_lname', 'Field', 'DataType', "C(40)")
* Props for the V_AUTHORS.au_fname field.
DBSetProp('V_AUTHORS.au_fname', 'Field', 'KeyField', .F.)
DBSetProp('V_AUTHORS.au_fname', 'Field', 'Updatable', .T.)
DBSetProp('V_AUTHORS.au_fname', 'Field', 'UpdateName', 'dbo.authors.au_fname')
DBSetProp('V_AUTHORS.au_fname', 'Field', 'DataType', "C(20)")
* Props for the V_AUTHORS.phone field.
DBSetProp('V_AUTHORS.phone', 'Field', 'KeyField', .F.)
DBSetProp('V_AUTHORS.phone', 'Field', 'Updatable', .F.)
DBSetProp('V_AUTHORS.phone', 'Field', 'UpdateName', 'dbo.authors.phone')
DBSetProp('V_AUTHORS.phone', 'Field', 'DataType', "C(12)")
* Props for the V_AUTHORS.address field.
DBSetProp('V_AUTHORS.address', 'Field', 'KeyField', .F.)
DBSetProp('V_AUTHORS.address', 'Field', 'Updatable', .F.)
DBSetProp('V_AUTHORS.address', 'Field', 'UpdateName', 'dbo.authors.address')
DBSetProp('V_AUTHORS.address', 'Field', 'DataType', "C(40)")
* Props for the V_AUTHORS.city field.
DBSetProp('V_AUTHORS.city', 'Field', 'KeyField', .F.)
DBSetProp('V_AUTHORS.city', 'Field', 'Updatable', .F.)
DBSetProp('V_AUTHORS.city', 'Field', 'UpdateName', 'dbo.authors.city')
DBSetProp('V_AUTHORS.city', 'Field', 'DataType', "C(20)")
* Props for the V_AUTHORS.state field.
DBSetProp('V_AUTHORS.state', 'Field', 'KeyField', .F.)
DBSetProp('V_AUTHORS.state', 'Field', 'Updatable', .F.)
DBSetProp('V_AUTHORS.state', 'Field', 'UpdateName', 'dbo.authors.state')
DBSetProp('V_AUTHORS.state', 'Field', 'DataType', "C(2)")
* Props for the V_AUTHORS.zip field.
DBSetProp('V_AUTHORS.zip', 'Field', 'KeyField', .F.)
DBSetProp('V_AUTHORS.zip', 'Field', 'Updatable', .F.)
DBSetProp('V_AUTHORS.zip', 'Field', 'UpdateName', 'dbo.authors.zip')
DBSetProp('V_AUTHORS.zip', 'Field', 'DataType', "C(5)")
* Props for the V_AUTHORS.contract field.
DBSetProp('V_AUTHORS.contract', 'Field', 'KeyField', .F.)
DBSetProp('V_AUTHORS.contract', 'Field', 'Updatable', .F.)
DBSetProp('V_AUTHORS.contract', 'Field', 'UpdateName', 'dbo.authors.contract')
DBSetProp('V_AUTHORS.contract', 'Field', 'DataType', "I")
```

Figure 15 shows the commands captured by SQL Profiler when the remote view was opened.

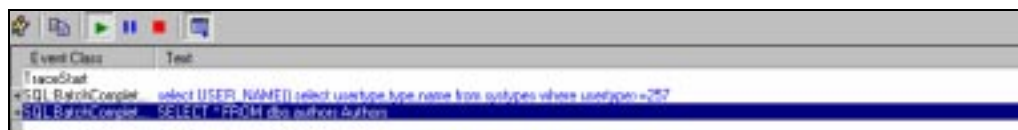


Figure 15. Opening the remote view.

As with the simple SQL pass through query, nothing special is sent to the server.

Figure 16 shows the results of changing a single row and issuing a TABLEUPDATE().

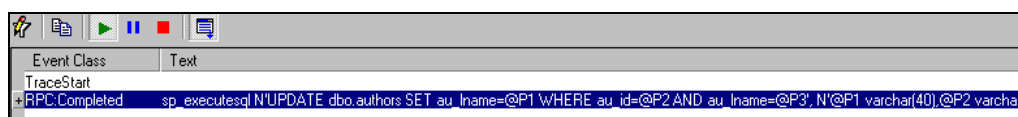


Figure 16. After changing one row and calling TABLEUPDATE().

Just like the parameterized SQL pass through, Visual FoxPro (and ODBC) uses *sp_executesql* to make the updates. In fact, modifying multiple rows and issuing a TABLEUPDATE() results in multiple calls to *sp_executesql* (see **Figure 17**).



Figure 17. The commands sent to SQL Server when multiple rows of a remote view are modified and sent to the server with a single call to TABLEUPDATE(.T.).

This is exactly the situation for which *sp_executesql* was created. The *au_lname* column of the first two rows was modified. SQL Server will be able to reuse the execution plan from the first query when making the changes for the next, eliminating the work that would have been done to prepare the execution plan (parse, resolve, optimize and compile) for the second query.

What have we learned? Overall, remote views and SQL pass through cause the same commands to be sent to the server for roughly the same situations, so the performance should be similar. Given these facts, the decision to use one over the other must be made based on other criteria.

Remote views are a wrapper for SQL pass through and, hence, a handholding mechanism that handles the detection of changes and the generation of the commands to write those changes back to the data store. Anything that can be done with a remote view can also be done using SQL pass through—although it may require more work on the part of the developer. However, the converse is not true. There are commands that can only be submitted using SQL pass through. Returning multiple result sets is the most obvious example.

Remote views require the presence of a Visual FoxPro database, which might be a piece of baggage not wanted in a middle-tier component. On the other hand, the simplicity of

remote views makes them a very powerful tool, especially when the query is static or has consistent parameters.

Using remote views and SPT together

In most cases, you don't have to choose between using remote views vs. SQL pass through. Combining the two in a single application is a very powerful technique. All the SQL pass through functions, including `SQLExec()`, `SQLCommit()`, `SQLRollback()`, `SQLGetProp()` and `SQLSetProp()`, can be called for existing connections. So if a connection to the server is established by a remote view, then you can use the same connection for SQL pass through.

To determine the ODBC connection handle for any remote cursor, use `CURSORGETPROP()`:

```
hConn = CURSORGETPROP("ConnectHandle")
```

In the following example, the previously described `v_authors` view is opened, and then its connection is used to query the titles table:

```
USE v_authors
hConn = CURSORGETPROP("ConnectHandle", "v_authors")
lnResult = SQLExec(hConn, "SELECT * FROM titles")
```

If your application uses remote views with a shared connection, then by using this technique you can use a single ODBC connection throughout the application for views and SQL pass through. The following sections give some brief examples of how combining remote views with SQL pass through can enhance your applications.



It is impossible to allow views to use a connection handle that was acquired by a SQL pass through statement. Therefore, to share connections between views and SQL pass through statements, you must open a view, acquire its connection, and then share it with your SQL pass through commands.

Transactions

Even if remote views suffice for all your data entry and reporting needs, you will need SQL pass through for transactions. Transactions are covered in greater detail in Chapter 11, "Transactions."

Stored procedures

Consider the example of a form that firefighters use to report their activity on fire incidents. It uses 45 different views, all of which share a single connection, for data entry. However, determining which firefighters are on duty when the alarm sounds is too complicated for a view. A stored procedure is executed with `SQLExec()` to return the primary keys of the firefighters who are on duty for a particular unit at a specific date and time. The result set is scanned and the keys are used with a parameterized view that returns necessary data about each firefighter.

Filter conditions

Suppose you give a user the ability to filter the data being presented in a grid or report. You can either bring down all the data and then filter the result set, or let the server filter the data by sending it a WHERE clause specifying the results the user wants. The latter is more efficient at run time, but how do you implement it? Do you write different parameterized views for each possible filter condition? Perhaps, if there are only a few. But what if there are 10, 20 or 100 possibilities? Your view DBC would quickly become unmanageable.

We solved this problem by creating a single view that defines the columns in the result set, but does not include a WHERE clause. The user enters all of his or her filter conditions in a form, and when the OK button is clicked, all the filter conditions are concatenated into a single, giant WHERE clause. This WHERE clause is tacked onto the end of the view's SQL SELECT, and the resulting query is sent to the back end with `SQLExec()`. Here's an example with a simple WHERE clause looking for a specific datetime:

```
*-- Open a view and use it as a template
USE myview IN 0 ALIAS template
lnHandle = CURSORGETPROP("ConnectHandle", "template")

*-- Get the SQL SELECT of the template
lcSelect = CURSORGETPROP("SQL", "template")

*-- Create a WHERE clause and add it to the SQL
lcWhere = " WHERE alarmtime = '" + lcSomeDatetime + "'"
lcSelect = lcSelect + lcWhere

*-- Execute the new query
lnSuccess = SQLExec(lnHandle, lcSelect, "mycursor")
```

You can even make the new result set updatable by simply copying some of the properties from the updatable view used as a template:

```
*-- Copy update properties to the new cursor
SELECT mycursor
CURSORSETPROP("Buffering", 5)
CURSORSETPROP("Tables", CURSORGETPROP("Tables", "template"))
CURSORSETPROP("UpdateNameList", CURSORGETPROP("UpdateNameList", "template"))
CURSORSETPROP("UpdatableFieldList", CURSORGETPROP("UpdatableFieldList",
"template"))
CURSORSETPROP("SendUpdates", .T.)
```

Summary

In this chapter, we explored the capabilities of SQL pass through and how to use it effectively. The next chapter takes a look at building client/server applications that can scale down as well as up, allowing you to use a single code base for all your customers.

