Everything You Wanted to Know About ODBC *

(* But Were Afraid to Ask)

Doug Hennig
Stonefield Software Inc.
Email: doug@doughennig.com
Corporate Web sites: stonefieldquery.com
stonefieldsoftware.com
Personal Web site : DougHennig.com
Blog: DougHennig.BlogSpot.com
Twitter: DougHennig

ODBC (Open DataBase Connectivity) has been around since the early 1990s, but there's more to it than just connecting to and querying a database. This session looks at all things ODBC, such as whether to use a DSN or a connection string, how differences between drivers affect your code, how different settings in VFP can impact a query, and many more topics.

Introduction

According to Wikipedia (https://en.wikipedia.org/wiki/Open_Database_Connectivity), "Open Database Connectivity (ODBC) is a standard application programming interface (API) for accessing database management systems (DBMS)." ODBC allows development platforms such as VFP to read from or write to database engines such as Microsoft SQL Server, Oracle, Access, PostgreSQL, MySQL, and so on without knowing much about how those engines function. It also allows, with some wrinkles, the same code to access any of these databases, making switching data stores relatively easy.

Many VFP developers are interested in accessing data other than VFP DBFs. (In this document, I'll refer to VFP tables as "local" and databases accessed via ODBC as "remote.") There are many reasons for this, such as:

- File security: DBFs are simply files on disk, and as such they can be copied (such as by a thief or disgruntled employee), encrypted (such as by ransomware), or deleted (accidentally or on purpose). With many remote databases, this is much more difficult or even impossible to do.
- Data security: the contents of DBF files can be viewed with a utility that can read DBF files, even something as simple as Notepad. Most remote database engines support security so only authorized users can access the data.
- Size: VFP DBFs cannot exceed 2 GB, although with VFP Advanced (http://baiyujia.com/vfpadvanced/default.asp), it's possible to go beyond that. Most remote database engines do not have that limitation.
- Performance: while VFP has been touted as "the fastest desktop database on the planet," accessing DBFs over a LAN or WAN can have significant performance issues. One of the reasons is that VFP typically loads as much of the CDX into memory as possible so it can use Rushmore to optimize queries. With remote databases, only the result of the SQL command is returned to the workstation, so potentially much less data travels down the wire.

This document looks at ODBC as it relates to VFP development in detail. However, I'm not going to discuss CursorAdapters or remote views but will instead focus on the so-called "SQL passthrough" (or SPT) functions. I'm also not going to discuss OLE DB, another API for accessing remote databases, for a few reasons:

- There are no built-in functions in VFP for accessing OLE DB. You must use COM or CursorAdapters.
- OLE DB's status is murky. At one point, Microsoft announced it was being deprecated. Then they announced it was undeprecated (https://en.wikipedia.org/wiki/OLE_DB).
- In my experience, ODBC is significantly faster than OLE DB. That may be because of the first point rather than an API issue, but the cause is unimportant for our purposes.

ODBC basics

To access a remote database using ODBC, you must use an ODBC driver specific for the database engine. Windows comes with some ODBC drivers, including Microsoft Access, Microsoft Excel (yes, you can access Excel spreadsheets using ODBC, although, weirdly, Windows comes with the driver for older XLS files and not for the newer XLSX format), and SQL Server. Other drivers can be downloaded, such as those for newer versions of Access and Excel (https://www.microsoft.com/en-ca/download/details.aspx?id=13255) or other database engines such as MySQL.

Note that an ODBC driver is not the same as an OLE DB provider and the connection strings (discussed in the next section) are different, so don't mix them up. If the connection string has "provider=" in it, it's an OLE DB not ODBC connection string and won't work with ODBC functions.

One thing to be aware of is the platform of the driver. Some drivers are available as 32-bit only, some as 64-bit only, and some come with both types. As a 32-bit application, VFP can only use a 32-bit driver, so be sure to take that into consideration when looking for a driver for your remote database.

ODBC connection properties are set with SQLSETPROP() and read with SQLGETPROP(). ODBC behavior properties are set with CURSORSETPROP() and read with CURSORGETPROP(). We'll look at various properties throughout this document.

Connecting to an ODBC data source

Before you can access an ODBC data source such as a SQL Server database, you must connect to it. There are two ways to connect: using a connection string (sometimes called a "DSN-less connection") or using a Data Source Name (DSN).

Connection string

Connect to an ODBC data source using a connection string with the SQLSTRINGCONNECT() function. This function has the following syntax:

```
nHandle = sqlstringconnect(cConnectionString)
```

You can also call SQLSTRINGCONNECT() with no parameters to display a "Select Connection or Data Source" dialog but that would only normally be used from the Command window rather than in an application for obvious reasons. You can also pass a logical parameter indicating whether the connection can be shared; I discuss that in the **Sharing connections** section.

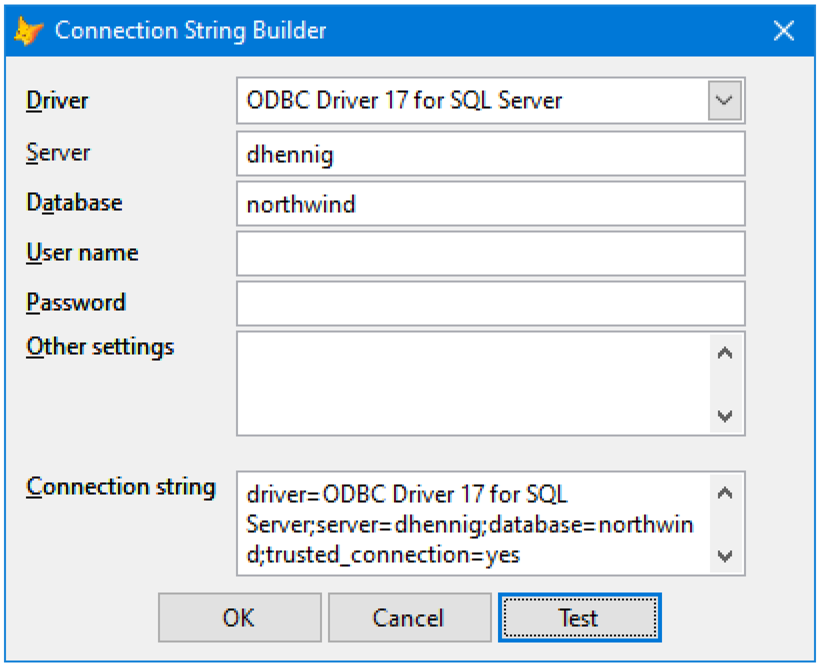
The connection string consists of name/value pairs in the format "name=value". Name/value pairs are separated with semi-colons. Here are some of the name/value pairs (names aren't case-sensitive but values may be):

- ODBC driver: this is required. Specify the driver using "driver=" followed by the name of driver. Note that you must spell the driver name exactly as it appears in the ODBC Administrator dialog. For example, the name of the Microsoft Excel driver is "Microsoft Excel Driver (*.xls, *.xlsx, *.xlsm, *.xlsb)"; you cannot leave off the "(*.xls, *.xlsx, *.xlsm, *.xlsb)" part and all the spaces are important.
- Server: most but unfortunately not all drivers use the "server" keyword to specify the server to connect to for server-based databases such as SQL Server.
- Database: in the case of a file-based database such as Excel, Access, or VFP, this is the path and name of the file to connect to. For server-based databases, it's the database name. There's a lot of variability in the name of this setting: many drivers, including SQL Server, use "database" as the keyword, Excel and Access use "dbq," and others use a variety of names.
- Username and password: the username and password to connect with, specified as "uid=*username*;pwd=*password.*" In the case of SQL Server, if the user connects using Windows authentication, omit these settings and instead add "trusted_connection=yes."

There are lots of other settings depending on the specific driver. A great resource for connection strings is the well-named https://www.connectionstrings.com; it lists dozens of database engines and provides syntax and examples for different types of drivers for each. Note: be sure to use the ODBC connection strings because OLE DB and .NET connection strings are also listed but won't work with ODBC.

It's not normally a good idea to hard-code a connection string into your application because servers, usernames, and passwords can change, and it's a big security no-no. Instead, store the connection string somewhere your application can read it from, such as a table or INI file. You should store it as an encrypted value (using, for example, Craig Boyd's VFPEncryption library) since it could contain a username, password, or other sensitive information.

If you want to be flexible, add a way the user (or at least an administrator or IT person) can specify the connection string. It's unlikely they'll be experts at entering connection strings, so I like to use a Connection String Builder dialog (provided with the sample files accompanying this document; see **Figure 1**) that allows the user to choose the driver from a combobox and has a Test button so they can be sure they entered a valid connection string.

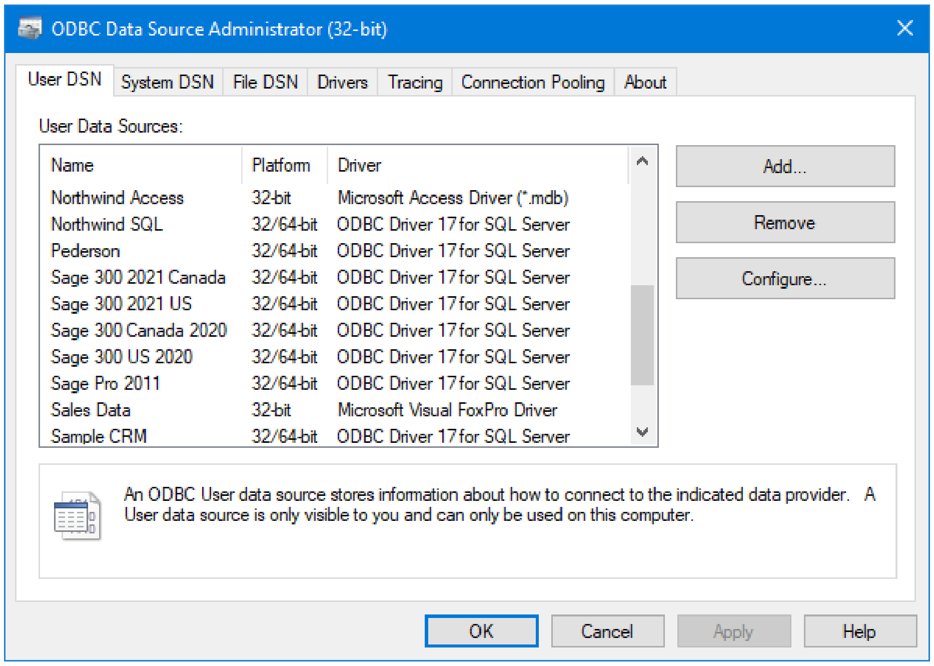


Driver	ODBC Driver 17 for SQL Server
Server	dhennig
Database	northwind
User name	
Password	
Other settings	
Connection string	driver=ODBC Driver 17 for SQL Server;server=dhennig;database=northwind;trusted_connection=yes

Figure 1. Providing a connection string builder makes it easier for the user to specify the connection string for their database.

DSN

DSNs are created using the ODBC Administrator (**Figure 2**). Use the 32-bit version of this tool. Newer versions of Windows show which version is which (32-bit vs. 64-bit) when you click the Start button and type "ODBC" but if you don't see that, the one you need to run is C:\Windows\SysWOW64\ODBCAD32.exe.



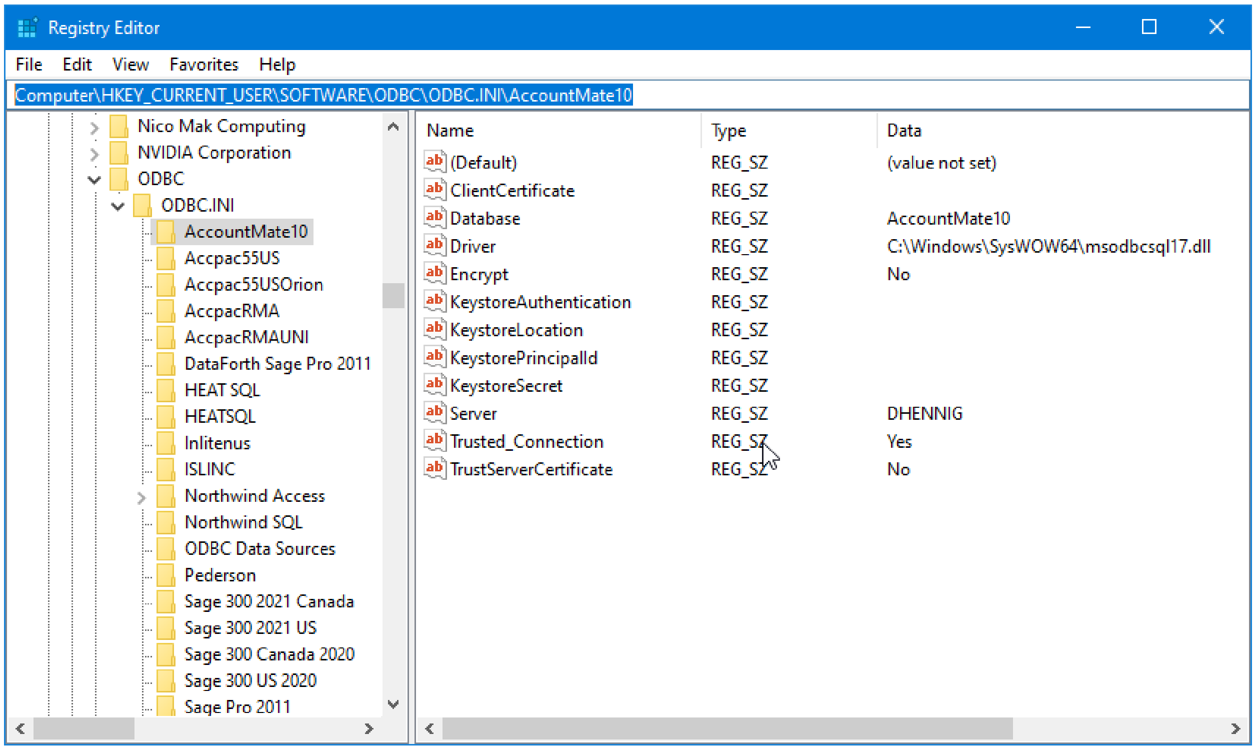
Name	Platform	Driver
Northwind Access	32-bit	Microsoft Access Driver (*.mdb)
Northwind SQL	32/64-bit	ODBC Driver 17 for SQL Server
Pederson	32/64-bit	ODBC Driver 17 for SQL Server
Sage 300 2021 Canada	32/64-bit	ODBC Driver 17 for SQL Server
Sage 300 2021 US	32/64-bit	ODBC Driver 17 for SQL Server
Sage 300 Canada 2020	32/64-bit	ODBC Driver 17 for SQL Server
Sage 300 US 2020	32/64-bit	ODBC Driver 17 for SQL Server
Sage Pro 2011	32/64-bit	ODBC Driver 17 for SQL Server
Sales Data	32-bit	Microsoft Visual FoxPro Driver
Sample CRM	32/64-bit	ODBC Driver 17 for SQL Server

An ODBC User data source stores information about how to connect to the indicated data provider. A User data source is only visible to you and can only be used on this computer.

Figure 2. The ODBC Administrator creates ODBC DSNs.

There are three kinds of DSNs:

- User DSNs are specific to the Windows account the current user is logged in as. They're stored in the Windows Registry under HKEY_CURRENT_USER\SOFTWARE\ODBC\ODBC.INI (see **Figure 3**).



Computer\HKEY_CURRENT_USER\SOFTWARE\ODBC\ODBC.INI\AccountMate10

- Nico Mak Computing
- NVIDIA Corporation
- ODBC
  - ODBC.INI
    - AccountMate10
    - Accpac55US
    - Accpac55USOrion
    - AccpacRMA
    - AccpacRMAUNI
    - DataForth Sage Pro 2011
    - HEAT SQL
    - HEATSQL
    - Inlitenus
    - ISLINC
    - Northwind Access
    - Northwind SQL
    - ODBC Data Sources
    - Pederson
    - Sage 300 2021 Canada
    - Sage 300 2021 US
    - Sage 300 Canada 2020
    - Sage 300 US 2020
    - Sage Pro 2011

Name	Type	Data
(Default)	REG_SZ	(value not set)
ClientCertificate	REG_SZ	
Database	REG_SZ	AccountMate10
Driver	REG_SZ	C:\Windows\SysWOW64\msodbcsql17.dll
Encrypt	REG_SZ	No
KeystoreAuthentication	REG_SZ	
KeystoreLocation	REG_SZ	
KeystorePrincipalId	REG_SZ	
KeystoreSecret	REG_SZ	
Server	REG_SZ	DHENNIG
Trusted_Connection	REG_SZ	Yes
TrustServerCertificate	REG_SZ	No

Figure 3. User DSNs are stored in the Windows Registry.

- System DSNs are available to all user accounts. 32-bit DSNs are stored in the Windows Registry under HKEY_LOCAL_MACHINE\SOFTWARE\WOW6432Node\ODBC\ODBC.INI.
- File DSNs are stored in a INI file with a DSN extension. Here's the content of a file DSN for the Northwind SQL Server database on my machine:

```
[ODBC]
DRIVER=ODBC Driver 17 for SQL Server
TrustServerCertificate=No
DATABASE=Northwind
WSID=DHENNIG
APP=Microsoft® Windows® Operating System
Trusted_Connection=Yes
SERVER=dhennig
```

  In my experience, file DSNs are rarely used.

Note that 32-bit applications can access both 32-bit and 64-bit user DSNs, but only 32-bit system DSNs.

Connect to a database with a DSN using either SQLCONNECT() or SQLSTRINGCONNECT() with "dsn=*DataSourceName*" (along with any other name/value pairs such as username and password) as the connection string. The syntax for SQLCONNECT() is:

```
nHandle = sqlconnect(cConnectionName | cDSN [, cUserID [, cPassword ]])
```

You can also call SQLCONNECT() with no parameters to display a "Select Connection or Data Source" dialog but that would only normally be used from the Command window rather than in an application. You can also pass a logical parameter indicating whether the connection can be shared; I discuss that in the **Sharing connections** section.

Pass a connection name to use a connection defined in a database container. Otherwise, specify the DSN to use and optionally the username and password.

To connect to a file DSN, instead of SQLCONNECT(), use SQLSTRINGCONNECT('FileDSN=*path*'), where *path* is the location and name of the file.

A DSN name can be hard-coded in your application, meaning someone has to create it on the user's computer, or you could prompt the user for the one to use by displaying a list of existing DSNs. To do that, use the GetDataSources method of the SFRegistryODBC class in SFRegistry.vcx (discussed in more detail in the **SFRegistryODBC** section and included in the sample files accompanying this document) to fill an array and then display the array to the user in, for example, a combobox. **Figure 4** shows an example.

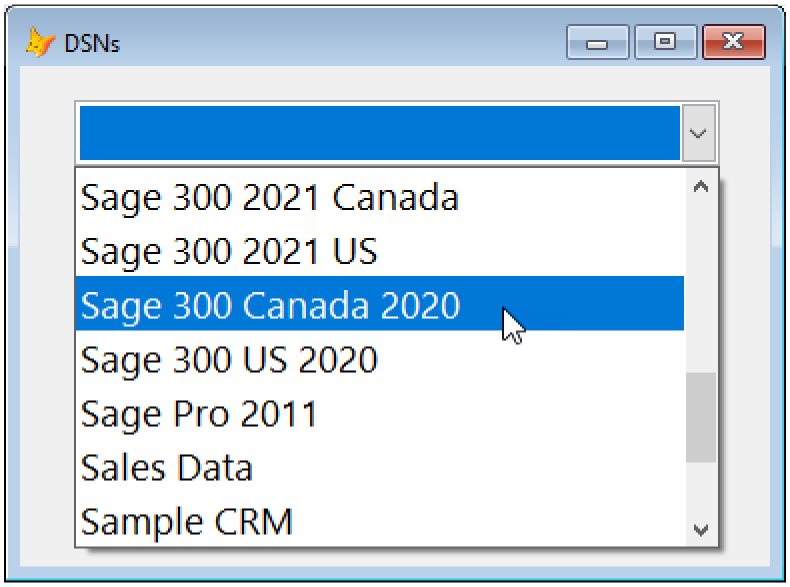


Figure 4. You can display a list of DSNs for the user to choose from.

Which one to use

Which is better: a DSN or a connection string? There are pros and cons to both.

- A DSN hides the settings for connection. The user or their IT staff creates the DSN using the ODBC Administrator so your application doesn't know the name of the server, the database, or other settings. If something changes, such as the database being moved to a new server, they just need to update the DSN and don't have to change anything in your application.
- The ODBC Administrator may require administrative permissions and certainly requires a higher level of knowledge than many users have.
- Specifying a connection string also requires a higher level of knowledge than many users have, although using a connection string builder can help.
- A DSN must be created on every workstation whereas a connection string can exist in a table or INI file in a common folder on a server.
- I don't know if it's still true since I haven't worked with the Pervasive database engine in a long time, but at one time, Pervasive didn't support a DSN-less connection; you had to create a DSN and use it in a SQLCONNECT() or SQLSTRINGCONNECT() statement. Other database engines may have similar limitations.
- A connection string is more flexible, since, depending on the driver, it's possible to specify additional settings that aren't exposed as options in the UI of the driver in the ODBC Administrator.

Connecting

The return value of both SQLCONNECT() or SQLSTRINGCONNECT() is an ODBC statement handle if the connection succeeded or -1 if it failed. VFP developers sometimes refer to it as a connection handle, but technically that isn't correct. Internally, VFP calls an ODBC API function to connect to an ODBC data source and receives a connection handle as the return value. To access the data source, it passes the connection handle to another ODBC API function which returns a statement handle. A statement handle is like a child of a connection handle—every connection handle, which indicates a unique connection to a database engine, can have multiple statement handles associated with it, but every statement handle has a single connection handle associated with it. The difference between statement and connection handles will become more apparent in the **Sharing connections** section.

What VFP returns to us is a simple numeric value: the first connection has a value of 1, the second is 2, and so on. Although you don't really need it, you can get the internal statement and connection handles from a statement handle using SQLGETPROP():

```
messagebox('Connection handle: ' + transform(sqlgetprop(lnHandle, 'ODBChdbc')) + ;
   chr(13) + 'Statement handle: ' + transform(sqlgetprop(lnHandle, 'ODBChstmt'))
```

Always check the return value for SQLCONNECT() or SQLSTRINGCONNECT() and use AERROR() to determine what went wrong if it's negative. For example:

```
lnHandle = sqlconnect('Northwind')
```

```
if lnHandle < 0
   aerror(laError)
   messagebox(laError[3])
   return
endif lnHandle < 0
sqldisconnect(lnHandle)
```

This is demo code, so it uses MESSAGEBOX() to display what went wrong. In a real application, you'd probably log this somewhere and display a more meaningful message to the user.

You usually don't want a login dialog to appear, so before connecting, use SQLSETPROP(0, 'DispLogin', 3). The first parameter is the statement handle; pass 0, meaning this applies to all connections. For the third parameter, 1 means display the login dialog if information is missing, 2 means always display the dialog, and 3 means never display the dialog. Some database engines require the login dialog, so use 2 in that case.

You can control how long VFP waits for a connection to be made with SQLSETPROP(0, 'ConnectTimeout', *value in seconds*). The default is 15 seconds.

If you leave the connection open for the life of the application (see the next section), you must store the connection handle somewhere. A common design is to use a global object to connect to the database and store the handle in a property of the object. See the **Managing connections** section for an example of a class that provides this capability.

Once you've connected, you can get the connection string, user name, and password used for a particular connection using SQLGETPROP(nHandle, 'ConnectString'), SQLGETPROP(nHandle, 'UserID'), and SQLGETPROP(nHandle, 'Password').

Disconnecting

It's important to release a connection once you no longer need it. Connections are expensive resources: they are time-consuming to open, they take up resources on both the workstation and server, and may be limited by the number of licenses available for the database engine. To release a connection, pass SQLDISCONNECT() the statement handle of the connection:

```
nResult = sqldisconnect(nHandle)
```

Pass 0 for nHandle to release all connections. The return value is 1 if the connection is successfully released, -1 if there is a connection level error, or -2 if there is an environment level error. In practice, most developers don't bother checking the return value. You cannot disconnect a connection that's busy executing a query or is in manual transaction mode (discussed later in this document).

Should you connect to a database, issue a query, then disconnect, or keep the handle open for the life of the application? In earlier days, the latter was preferred because connecting to a database took some time. Modern database engines often use connection pooling: closing a connection doesn't actually close the connection but rather puts it into a pool of open connections. Opening a connection returns an existing connection from the pool and only opens a new one if none are available. See https://docs.microsoft.com/en-us/sql/odbc/reference/develop-app/driver-manager-connection-pooling for more information on connection pooling. Since not all drivers support pooling (see the Connection Pooling page of the ODBC Administrator to configure pooling), it's best to test with your specific driver to determine whether the connect-query-disconnect or the keep-connection-open strategy works best.

SQLSETPROP(nHandle, 'IdleTimeout', *value in minutes*) specifies how long to wait after the last time a connection is used before automatically disconnecting. The default is 0, which mean don't disconnect idle connections.

Sharing connections

Every time you use SQLCONNECT() or SQLSTRINGCONNECT() to connect to an ODBC data source, VFP opens a new connection to the database engine. As mentioned in the previous section, connections are expensive resources. Fortunately, VFP allows you to declare a connection as shared.

SQLSTRINGCONNECT() and SQLCONNECT() accept an optional parameter to indicate whether the connection should be shareable:

```
nHandle = sqlstringconnect(cConnectionString, [lShareable])
nHandle = sqlconnect(cConnectionName | cDSN [, cUserID [, cPassword ]], [lShareable])
```

If lShareable is True, these functions create a new connection and return a new statement handle for that connection, but flag the connection as being shareable.

To create a new statement handle using an existing connection (that is, share the connection handle), pass the existing statement handle to SQLCONNECT():

```
nNewHandle = sqlconnect(nHandle)
```

Remote views can share connections with SPT functions by specifying CONNSTRING nHandle in the USE command.

You can determine if a connection is shared using SQLGETPROP(nHandle, 'Shared').

SharedConn.prg (**Listing 1**), included in the files that accompany this document, shows how shared connections work. It displays the VFP statement handle and the internal ODBC connection and statement handles for several "connections."

Listing 1. This example shows how shared connections work.

```
close databases all
create database Remote
create connection NWConnection connstring 'driver=SQL Server;' + ;
   'server=dhennig;database=Northwind'
create sql view NWCustomers remote connection NWConnection share as ;
   select * from Customers
```

```
create sql view NWOrders remote connection NWConnection share as ;
   select * from Orders

* Create a shareable ODBC connection, and then another one it's shared with.

lnHandle1 = sqlstringconnect('dsn=Northwind', .T.)
lnHandle2 = sqlconnect(lnHandle1)

* Create a shareable connection from the DBC Connection object.

open database Remote
lnHandle3 = sqlconnect('NWConnection', .T.)

* Open the remote views; they should share the connection with lnHandle3.

use NWCustomers
use NWOrders in 0

* Display connection information.

clear
ShowConnection(lnHandle1)
ShowConnection(lnHandle2)
ShowConnection(lnHandle3)
ShowConnection(cursorgetprop('ConnectHandle', 'NWCustomers'))
ShowConnection(cursorgetprop('ConnectHandle', 'NWOrders'))
close databases all
erase Remote.d*
sqldisconnect(0)

function ShowConnection(tnHandle)
? 'VFP handle=' + transform(tnHandle) + ;
   ', ODBC connection=' + transform(sqlgetprop(tnHandle, 'ODBChdbc')) + ;
   ', ODBC statement=' + transform(sqlgetprop(tnHandle, 'ODBChstmt'))
```

Running this program displays results like:

```
VFP handle=1, ODBC connection=104545824, ODBC statement=104548616
VFP handle=2, ODBC connection=104545824, ODBC statement=104542648
VFP handle=3, ODBC connection=104542136, ODBC statement=104549824
VFP handle=4, ODBC connection=104542136, ODBC statement=104539896
VFP handle=5, ODBC connection=104542136, ODBC statement=104547480
```

Note that while every connection has a different VFP and ODBC statement handles, the first two have the same connection handle, as do the last three. Thus, although it appears we have five connections to SQL Server, SQL Server only sees two connections.

With shared connections, it's possible for one query to block another using the same shared connection. For example, if you run the following code, you'll get a "Connection is busy with results for another hstmt" error for the first SQLEXEC() statement. This is because it shares the same connection as the NWCustomers remote view, the FetchAsNeeded property of the remote view is True, and not all records for the view have been returned yet. By default, VFP doesn't allow another process to simultaneously retrieve data over the same connection. However, if you set the AllowSimultaneousFetch property for the view, either permanently using DBSETPROP() or for the current instance using CURSORSETPROP(), VFP allows this. So, the second SQLEXEC() succeeds in the example shown in **Listing 2** (taken from SimultaneousFetch.prg).

Listing 2. Shared connections can block each other unless you allow simultaneous fetching.

```
close databases all
create database Remote
create connection NWConnection connstring 'driver=SQL Server;' + ;
   'server=(local);uid=sa;pwd=;database=Northwind'
create sql view NWCustomers remote connection NWConnection share as ;
   select * from Customers
dbsetprop('NWCustomers', 'View', 'FetchAsNeeded', .T.)
dbsetprop('NWCustomers', 'View', 'FetchSize',     10)

* Open the NWCustomers view, then open a shared connection and try to get
* records from the connection. If it failed, show the error message.

use NWCustomers
lnHandle = sqlconnect('NWConnection', .T.)
lnResult = sqlexec(lnHandle, 'select * from Employees')
if lnResult < 0
   aerror(laError)
   messagebox(laError[2])
endif
sqldisconnect(lnHandle)
use in NWCustomers

* Now set the AllowSimultaneousFetch property to .T. and try again.

dbsetprop('NWCustomers', 'View', 'AllowSimultaneousFetch', .T.)
use NWCustomers
lnHandle = sqlconnect('NWConnection', .T.)
lnResult = sqlexec(lnHandle, 'select * from Employees')
if lnResult < 0
   aerror(laError)
   messagebox(laError[2])
endif
sqldisconnect(lnHandle)
close databases all
erase Remote.dbc
```

You can determine if a shared connection is busy using SQLGETPROP(nHandle, 'ConnectBusy').

Managing connections

Keep the statement handle returned by SQLSTRINGCONNECT() or SQLCONNECT() in a variable or property of an object because it's needed for everything you'll do using ODBC. I like to use a manager class to connect and disconnect from the data source and store the connection handle. I instantiate the class, named SFConnectionManager contained in

SFData.vcx (included with the files accompanying this document), into a global object in my applications so it can be used anywhere.

Set the cConnString property to the connection string to use. Because the Connect method only connects if it hasn't already, you can call it any time you wish. The Disconnect method is automatically called when the object is destroyed so you don't have to call it manually unless you wish to. Here's an example of a program that uses SFConnectionManager:

```
set library to VFPEncryption.fll
goConn = newobject('SFConnectionManager', 'SFData.vcx')
goConn.cConnString = Decrypt(ReadINI(fullpath('Settings.ini'), 'Settings', ;
   'ConnectionString'), 'ConnStringKey')
if goConn.Connect()
   sqlexec(goConn.nHandle, 'select * from Customers')
   browse
endif goConn.Connect()
```

It reads the connection string from an INI file, decrypts it, saves it in the cConnString property of an SFConnectionManager object, calls its Connect method, and then uses its nHandle property for the SQLEXEC() statement.

Many applications open a single connection to an ODBC data source and use that connection for the duration of the application. However, there may be situations where you need to open additional connections for short periods of time, such as retrieving data from a different database. Rather than reissuing SQLCONNECT() or SQLSTRINGCONNECT() calls each time, use SQLIDLEDISCONNECT().

SQLIDLEDISCONNECT() temporarily disconnects from a data source but preserves the statement handle and original connection parameters. The syntax is:

```
nResult = sqlidledisconnect(nHandle)
```

Specify the statement handle for the connection or 0 to disconnect all connections. The return value is 1 if it's successful or -1 if it cannot disconnect. You cannot disconnect a connection that's busy executing a query or is in manual transaction mode.

If your application uses the statement handle again, VFP automatically reconnects to the data source using the original connection parameters. If the connection cannot be reestablished, error 1526 ("Connectivity error") occurs. As a result, you can reuse the connection at any time without worrying about whether it's still active.

SQLIDLEDISCONNECT() disconnects the statement handle, but does not release the connection to the data source until all its statement handles have been released. After calling SQLIDLEDISCONNECT(), you can use SQLGETPROP() to determine whether the connection has been released. The ODBChstmt property is 0 if the statement handle has been released; the ODBChdbc property is 0 if the connection to the data source has been released.

ASQLHANDLES() populates a one-dimensional array with a list of all active statement handles, and returns the number of handles in use:

```
nCount = asqlhandles(ArrayName [, nHandle])
```

If you pass the optional nHandle parameter, the array is populated with statement handles that share the same connection handle, including nHandle itself. You can use this, for example, to close all connections for a specific connection handle:

```
lnHandles = asqlhandles(laHandles, lnHandle)
for lnI = 1 to lnHandles
   sqldisconnect(laHandles[lnI])
next lnI
```

What happens if an existing connection goes down, such as if SQL Server is stopped and started again or you've connected to the data source over the Internet and there's a temporary disruption? Walter Meester posted an idea on Level Extreme (formerly known as the Universal Thread; https://www.levelextreme.com) about automatically reconnecting. His application executes SQLEXEC(nHandle, '') every 5 to 15 seconds (presumably with a timer). If it fails, he calls the following code:

```
ASQLHANDLES(aSqlhndls)
FOR EACH nOdbchdbc IN aSqlhndls
   =SQLIDLEDISCONNECT(nOdbchdbc)
ENDFOR
```

This causes an automatic reconnection the next time the connection is used. As John Ryan called it in his reply, "fiendishly clever."

Accessing an ODBC data source

The main function used to access an ODBC data source is SQLEXEC(). Its syntax is:

```
nResult = sqlexec(nHandle [, cSQLCommand [, cCursorName [, aCountInfo]]])
```

The cSQLCommand parameter is shown as optional but that's only the case if you've previously used SQLPREPARE() (discussed later) to prepare a command.

The return value is the number of result sets created if there's more than one (see the **Retrieving multiple result sets** section), 0 if it's still executing, 1 if it's finished, or -1 if it failed. Always check the return value and use AERROR() to determine what went wrong if it's negative. You could use SQLSETPROP(nHandle, 'DispWarnings', .T.) (the default is .F.) to display error messages, but that isn't suitable in an application because you can't control what's displayed to the user.

cSQLCommand is any valid SQL statement using the syntax of the database engine you're connected to. This is important to note because VFP syntax may differ from the syntax of the database engine. For example, this is a valid VFP SQL statement:

```
sele * from Orders ;
```

```
   where between(OrderDate, date() - 7, date()
```

However, it isn't valid when used with SQL Server for four reasons:

- "SELE" is an abbreviation for SELECT in VFP; SQL Server requires the full command.
- Semi-colons are statement terminators not line terminators in SQL Server. As with VFP, a SQL Server command can span several lines but a carriage return terminates lines not a semi-colon.
- There is no BETWEEN() function in SQL Server.
- There is no DATE() function in SQL Server.

The SQL Server equivalent is:

```
select * from Orders where OrderDate between DATEADD(day, 7, GETDATE()) and GETDATE()
```

or:

```
select * from Orders
   where OrderDate between DATEADD(day, 7, GETDATE()) and GETDATE()
```

or:

```
select * from Orders
   where OrderDate between ?ldDate1 and ?ldDate2
```

where ldDate1 and ldDate2 are variables containing DATE() – 7 and DATE(), respectively. I'll discussed using parameters in statements like this in the next section.

Use delimiters for objects with "illegal" names (for example, containing spaces or punctuation characters). The delimiters to use depend on the specific database engine; for example, SQL Server uses square brackets (such as [Order Details]), some versions of Access use double quotes (such as "Order Details"), and some versions of MySQL use reverse apostrophes (such as `Order Details`).

Creating a multi-line SQL statement is most easily accomplished using the TEXT command:

```
text to lcSQL noshow
select * from Orders
   where OrderDate between ?ldDate1 and ?ldDate2
endtext
lnResult = sqlexec(lnHandle, lcSQL)
```

You can specify how long VFP should wait for a query to execute using SQLSETPROP(nHandle, 'QueryTimeout', *value in seconds*). The default is 0, meaning there is no timeout. Related to this is SQLSETPROP(nHandle, 'WaitTime', *value in milliseconds*), which specifies how long VFP waits before checking if the SQL statement has completed; the default is 100 ms.

SQLSETPROP(nHandle, 'PacketSize', *value in bytes*) specifies the size of network packets used by the connection. The default is 4096. You may find that adjusting this value gives better performance.

Besides using the CAST() function in a SQL statement, you have some control over how remote data types map to VFP data types. Use CURSORSETPROP('MapVarchar', .T., 0) prior to using SQLEXEC() to specify that Varchar data types should come into VFP as Varchar (the default is .F., meaning they come into VFP as Character). Use CURSORSETPROP('MapVarbinary', .T., 0) to specify that Binary and Varbinary data types should come into VFP as Varbinary or Blob (the default is .F., meaning they come into VFP as Character). Although I'm not sure why you'd want to, you can use CURSORSETPROP('UseMemoSize', *value*, 0) to specify that fields with fewer than 255 characters (the default) come into VFP as Memo.

Parameterized statements

Note the syntax for using a parameterized value in the previous example: "?" followed by an expression. For a variable, you can just use the variable name. Surround more complex expressions with parentheses. For example, in the following code, SQLEXEC() calls the GetCustomer function and passes the result as the parameter value for the SQL SELECT statement:

```
sqlexec(lnHandle, 'select * from Customers where CustomerID = ?(GetCustomer())')

function GetCustomer
return inputbox('Customer ID', 'Select Customer', 'ALFKI')
```

Use parameterized statements rather than constructing a SQL statement with inserted variable values. There are at least three good reasons for this:

- With a parameterized statement, you don't have to worry about how to format different data types into literal values or put single quotes around strings. For example, because ldDate1 and ldDate2 contain date values, which can't be mixed with character values, here's how you'd have to construct the SQL statement so the dates are in a format SQL Server understands:

  ```
  lcDate1 = transform(dtos(ldDate1), '@R 9999-99-99')
  lcDate2 = transform(dtos(ldDate2), '@R 9999-99-99')
  text to lcSQL noshow textmerge
  select * from Orders
    where OrderDate between '<<lcDate1>>' and '<<lcDate2>>'
  endtext
  ```

- A parameterized statement may be optimized by the database engine so the second and subsequent times it's used are executed much faster than the first (simply inserting new values into the previously analyzed statement).
- Constructing a SQL statement from user-supplied values can lead to SQL injection attacks as illustrated by the famous cartoon from XKCD (https://xkcd.com/327) shown in **Figure 5**.

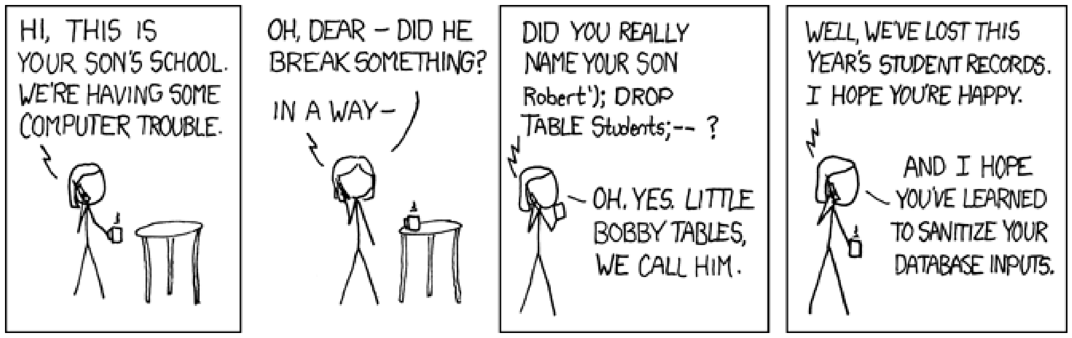

HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH, YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

Figure 5. This cartoon illustrates the dangers of constructing a SQL statement using user-supplied values.

I use a helper class to make it easier to deal with parameterized statements; see the **SFDataManager** section later in this document.

Literal values

Sometimes you'll use literal values in SQL statements. See **Table 1** for rules for using literal values by data type. Note that these are general ODBC rules; check the documentation for your specific database engine for variations. For example, with SQL Server, you can specify a date literal without the "{d" and "}" (for example, "WHERE OrderDate > '2021-01-13'").

Table 1. Rules for literal values.

Date type	Format	Example
String	Surrounded with single quotes	WHERE City = 'Rome'
Date	{d 'YYYY-MM-DD'}	WHERE OrderDate > {d '2021-01-13'}
DateTime	{t 'YYYY-MM-DD HH:MM:SS'} with the time in 24-hour format	WHERE Entered < {t '2021-01-13 17:26:00'}
Boolean	0 for false, 1 for true	WHERE Active = 1
All numeric types	The number	WHERE Cost > 500

Calling stored procedures

To call a stored procedure, use a statement in the format:

```
exec ProcName [parameter1, [parameter2, … ]]
```

For example, this calls the Sales By Year stored procedure, passing it two dates. Note the square brackets surrounding the procedure name since it has spaces in it:

```
ldDate1 = {^1996-01-01}
ldDate2 = {^1996-12-31}
sqlexec(lnHandle, 'exec [Sales By Year] ?ldDate1, ?ldDate2')
```

For a stored procedure with an output parameter, use ?@*VariableName*; after the statement has executed, the variable contains the output value.

Naming the result set

Some SQL statements, such as INSERT, UPDATE, and DELETE, don't return a result set so no cursor is created. For those that do, such as SELECT, the resulting cursor is named SQLResult by default. You can specify the name to use with the cCursorName parameter in the SQLEXEC() statement. For example:

```
sqlexec(lnHandle, 'select * from Customers', 'CustomerCursor')
```

Whether you specify a name or not, any existing cursor with the name to be used is closed before the new cursor is created.

Retrieving multiple result sets

Some database engines support a command consisting of multiple SQL statements separated by semi-colons. In that case, the SQLEXEC() return value indicates the number of result sets returned and the cursors are named with an incrementing number suffix to the name of the first cursor. This code creates two cursors, one named MyCursor which contains customers and one named MyCursor1 which contains employees:

```
lnResult = sqlexec(lnHandle, 'select * from Customers; select * from Employees', ;
   'MyCursor')
```

By default, the SQLEXEC() command doesn't return until all statements have executed. However, you can use SQLSETPROP() to set the BatchMode property to .F. (the default is .T.), causing SQLEXEC() to return result sets as they're created. With BatchMode turned off, SQLEXEC() returns the first result set. To get the next result set, call SQLMORERESULTS():

```
nResult = sqlmoreresults(nHandle [, cCursorName] [, aCountInfo]])
```

Because you can optionally specify a cursor name, you can name each cursor individually rather than all using the same base name.

The return value is 0 if the statement is still executing, 1 if it's finished, 2 if there are no more result sets, and -1 if an error occurred. You have to ensure SQLMORERESULTS() returns 2 before using any other SPT function or closing the connection or you'll get an "Invalid call issued while executing a SQLMORERESULTS() sequence" error.

The code in **Listing 3**, taken from MultipleResults.prg, shows using multi-statement commands with BatchMode turned on and off.

Listing 3. MultipleResults.prg shows how to execute multi-statement commands.

```
lnHandle = sqlstringconnect('driver=SQL Server;server=dhennig;' + ;
   'database=Northwind')

* Issue a multi-statement command.
```

```
sqlsetprop(lnHandle, 'BatchMode', .T.)
lnResult = sqlexec(lnHandle, 'select * from Customers; select * from Employees', ;
   'MyCursor')
messagebox(lnResult)
select MyCursor
browse
select MyCursor1
browse
close databases

* Now do it with BatchMode turned off.

sqlsetprop(lnHandle, 'BatchMode', .F.)
lnResult = sqlexec(lnHandle, ;
   'select * from Customers; select * from Employees; select * from Products', ;
   'Customers')
messagebox(lnResult)
browse
llDone   = .F.
lcCursor = 'Employees'
do while not llDone
   lnResult = sqlmoreresults(lnHandle, lcCursor)
   messagebox(lnResult)
   do case

* Still executing: do nothing.

      case lnResult = 0

* Done executing the current statement: see the results and set the name for
* the next cursor.

      case lnResult = 1
         browse
         lcCursor = 'Products'

* Done all statements: exit the loop.

      case lnResult = 2
         llDone = .T.

* Error.

      otherwise
         aerror(laError)
         messagebox(laError[3])
   endcase
enddo while not llDone

sqlsetprop(lnHandle, 'BatchMode', .T.)
sqldisconnect(lnHandle)
```

When you run this, a cursor named MyCursor displays customers, then one named MyCursor1 shows Employees. Then three cursors are displayed: one named Customers, one named Employees, and one named Products.

Getting a count of the records affected by SPT commands

SQLEXEC() and SQLMORERESULTS() have an optional parameter that provides information about the number of records affected by a SQL command. The aCountInfo parameter is the name of an array to populate with row count information. If the array doesn't exist, it's created. The array contains two columns, one row for each SQL command you issue that doesn't return a record set, and one row for each record set returned. The contents of each row depend on what happened. **Table 2** shows the possible values.

Table 2. The contents of the array filled by SQLEXEC() or SQLMORERESULTS().

Result	Column 1	Column 2
Command succeeded and does not return a result set (e.g., INSERT, UPDATE)	Empty string	Number of records affected, or -1 if not available (still executing)
Command failed	"0"	-1
Command returned a result set	Cursor name	Number of records affected, or -1 if not available (still executing)

If you execute a command that returns multiple result sets and the BatchMode property of the connection is .T., the array contains one row for each result set. If BatchMode is .F., each call to SQLMORERESULTS() replaces the previous contents of the array with a single row describing the new result set.

Here's an example that shows how the array is useful to determine how many records are affected by UPDATE and DELETE commands:

```
sqlexec(lnHandle, 'create table Testing (Name varchar(1), Type int)')
sqlexec(lnHandle, "insert into Testing values ('A', 1)")
sqlexec(lnHandle, "insert into Testing values ('B', 2)")
sqlexec(lnHandle, "insert into Testing values ('C', 1)")
sqlexec(lnHandle, "insert into Testing values ('D', 1)")
sqlexec(lnHandle, "insert into Testing values ('E', 2)")

sqlexec(lnHandle, 'update Testing set Type = 3 where Type = 2; ' + ;
   'delete from Testing where Type = 1', '', laCount)
messagebox('Number of records updated: ' + transform(laCount[1, 2]) + chr(13) + ;
   'Number of records deleted: ' + transform(laCount[2, 2]))
```

The message shows that two records were updated and three were deleted.

Asynchronous processing

By default, VFP waits until the ODBC data source has finished processing the SQL statements before continuing execution. One problem with that is a long-running query: VFP is essentially frozen until it completes and there's no way for the user to stop it.

To process a SQL statement asynchronously, use SQLSETPROP() to set the Asynchronous property to .T. In that case, SQLEXEC() returns without necessarily finishing the processing; a return value of 0 means the statement is still processing.

To check whether processing has completed, call SQLEXEC() again with either the same command or an empty string. When it no longer returns 0 (or -1 indicating an error), the command is done. To cancel execution before it's complete, call SQLCANCEL().

Listing 4 is an example, taken from Async,prg, that queries a table with more than 100,000 records, so it'll take a little while to execute.

Listing 4. Asynchronous processing allows you to display progress to the user and allow them to cancel the execution.

```
lnHandle = sqlstringconnect('driver=SQL Server;server=dhennig;' + ;
   'database=ISLINC')
sqlsetprop(lnHandle, 'Asynchronous', .T.)
cursorsetprop('FetchSize', 1000, 0)
set escape on
llCancel = .F.
on escape llCancel = .T.
clear
llDone  = .F.
lnCount = 0
do while not llDone and not llCancel
   lnResult = sqlexec(lnHandle, 'select * from ARIBH', 'ARIBH', laCount)
   if laCount[1, 2] > 0
      lnCount = lnCount + laCount[1, 2]
   endif laCount[1, 2] > 0
   do case

* Still executing.

      case lnResult = 0
         ? 'Records retrieved: ' + transform(lnCount)

* An error.

      case lnResult < 0
         aerror(laError)
         messagebox(laError[3])
         llDone = .T.

* We're done.

      otherwise
         messagebox('Done')
```

```
          ? 'Records retrieved: ' + transform(lnCount)
          browse
          llDone = .T.
   endcase
   doevents
enddo while not llDone ...

* Cancel execution if the user pressed Esc.

if llCancel
   sqlcancel(lnHandle)
   messagebox('Cancelled')
endif llCancel

on escape
close databases
sqlsetprop(lnHandle, 'Asynchronous', .F.)
sqldisconnect(lnHandle)
```

As the program runs, it displays "Records retrieved: 1000," "Records retrieved: 2000," and so on. Once it's done, it shows the final record count and displays the cursor in a BROWSE window. Pressing Escape before it's done cancels the execution.

Another way to determine how many records have been retrieved is CURSORGETPROP('RecordsFetched'). Also, CURSORGETPROP('FetchIsComplete') returns .F. until processing is done and then returns .T. The functions are more useful with remote views since there is no array containing a count of the records retrieved so far.

Preparing SQL statements

Many database engines analyze the SQL statement to come up with an execution plan and then execute that plan. The analysis can take some time, so if a SQL statement is executed many times, you may get better performance by preparing it first. SQLPREPARE() sends a SQL statement to the database engine to compile but not execute it. Here's the syntax:

```
nResult = sqlprepare(nHandle, cSQLCommand, [cCursorName])
```

To execute it, call SQLEXEC() without passing the SQL statement. Here's an example:

```
sqlprepare(lnHandle, 'select * from Orders ' + ;
   'where OrderDate between ?ldDate1 and ?ldDate2')

ldDate1 = {^1996-07-01}
ldDate2 = {^1996-07-31}
sqlexec(lnHandle)
browse

ldDate1 = {^1996-08-01}
ldDate2 = {^1996-08-31}
sqlexec(lnHandle)
browse
```

Note that the ldDate1 and ldDate2 variables don't exist when SQLPREPARE() is called; they don't need to since the command isn't executed. The two BROWSE windows that appear show July and August orders, respectively.

Limiting how many records are retrieved

You can limit the number of records retrieved by a query without changing the SQL statement (such as adding a TOP clause) by using CURSORSETPROP('MaxRecords', nCount). Set nCount to the number of records to retrieve, -1 to retrieve all records, or 0 to execute the query but return no records.

Why would you want to do this? Here are a couple of scenarios where this makes sense:

- In Stonefield Query, the user can test if the relationship they defined between two tables works by clicking a Test button. The Click method of this button creates a SQL statement similar to:

  ```
  SELECT Table1.Field1, Table2.Field2
     FROM Table1
     JOIN Table2 ON Table1.Field1 = Table2.Field2
  ```

  However, we don't want to retrieve all records from this query; the tables may contain a lot of records and we only need a few to see if the join is correct. Using TOP N may not be feasible because it may require an ORDER BY clause and not all database engines support it (MySQL uses LIMIT and Oracle uses FETCH ROWS ONLY, for example). Setting the nCount parameter to 10 ensures we only retrieve a representative sample.

- You want to create a cursor with the desired structure but no records when a form is initially displayed for data binding purposes, and then once the user enters some filter criteria, recreate the cursor with the desired records. You could also, of course, simply use something like SELECT * FROM MyTable WHERE 0 = 1.

Detecting delayed memo fetching

If you use CURSORSETPROP('FetchMemo', .F.) to perform delayed memo fetching, the contents of memo fields are not retrieved initially. Instead, VFP retrieves the contents of a memo field for a given row the first time it is needed, such as when it's displayed in a form or used in an expression. The tradeoff is a small delay the first time the memo field is needed for a given row versus a longer delay when the database is initially queried.

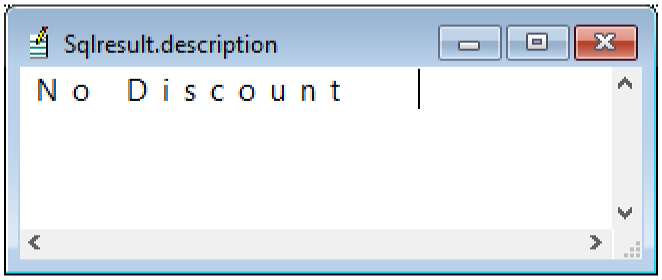
You can use ISMEMOFETCHED() to determine if the specified memo field in the current record of the cursor has been retrieved. Calling ISMEMOFETCHED() doesn't cause the memo field to be fetched, so it can be used safely to determine whether a delay may occur before you try to access the memo field's contents.

Map remote Unicode data to ANSI in memo fields

Use SYS(987, .T.) to map remote Unicode data to ANSI when retrieved into a memo field.

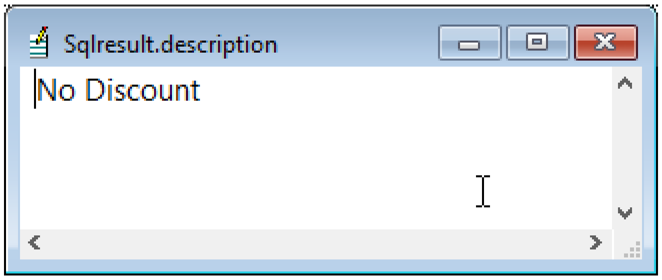
Call SYS(987) without the parameter to obtain the current setting. Once a cursor is opened, changing the SYS(987) setting does not affect future fetches by that cursor. This setting is global to all data sessions.

Figure 6 shows how Unicode data appears in VFP. **Figure 7** shows the result of using SYS(987, .T.).



```
N o   D i s c o u n t
```

Figure 6. Unicode data as it appears in VFP.



No Discount

Figure 7. Unicode data is automatically converted to ANSI using SYS(987).

Tracing

Some database engines, such as SQL Server, have the ability to trace all connections and queries to the engine. However, this is also built into the ODBC Administrator. Choose the Tracing page, specify a path for the log file, and click the Start Tracing Now button. Note that you must do this prior to starting your application as it only logs for applications started after tracing has begun. Click Stop Tracing Now to turn off tracing. I'm not sure how useful tracing is as it shows all of the C-style API calls; for example:

```
VFPA            5e58-b20  ENTER SQLAllocEnv
        HENV *             0x0097A0A0

VFPA            5e58-b20  EXIT  SQLAllocEnv  with return code 0 (SQL_SUCCESS)
        HENV *             0x0097A0A0 ( 0x06BC40B8)

VFPA            5e58-b20  ENTER SQLAllocConnect
        HENV               0x06BC40B8
        HDBC *             0x0019D56C
```

Transactions

VFP supports transactions against remote data. Interestingly, there isn't the equivalent of a BEGIN TRANSACTION command. Instead, use SQLSETPROP(nHandle, 'Transactions', 2) (or use the DB_TRANSMANUAL constant from FoxPro.h) to enable manual transactions and SQLSETPROP(nHandle, 'TRANSACTIONS', 1) (or use the DB_TRANSAUTO constant from FoxPro.h) to disable them. Once you've enable manual transactions, all queries participate in a transaction until you issue SQLCOMMIT(nHandle) to commit the transaction or SQLROLLBACK(nHandle) to roll back the transaction. Both functions return 1 if successful or -1 if not, in which case you should use AERROR() to determine what went wrong. Here's an example:

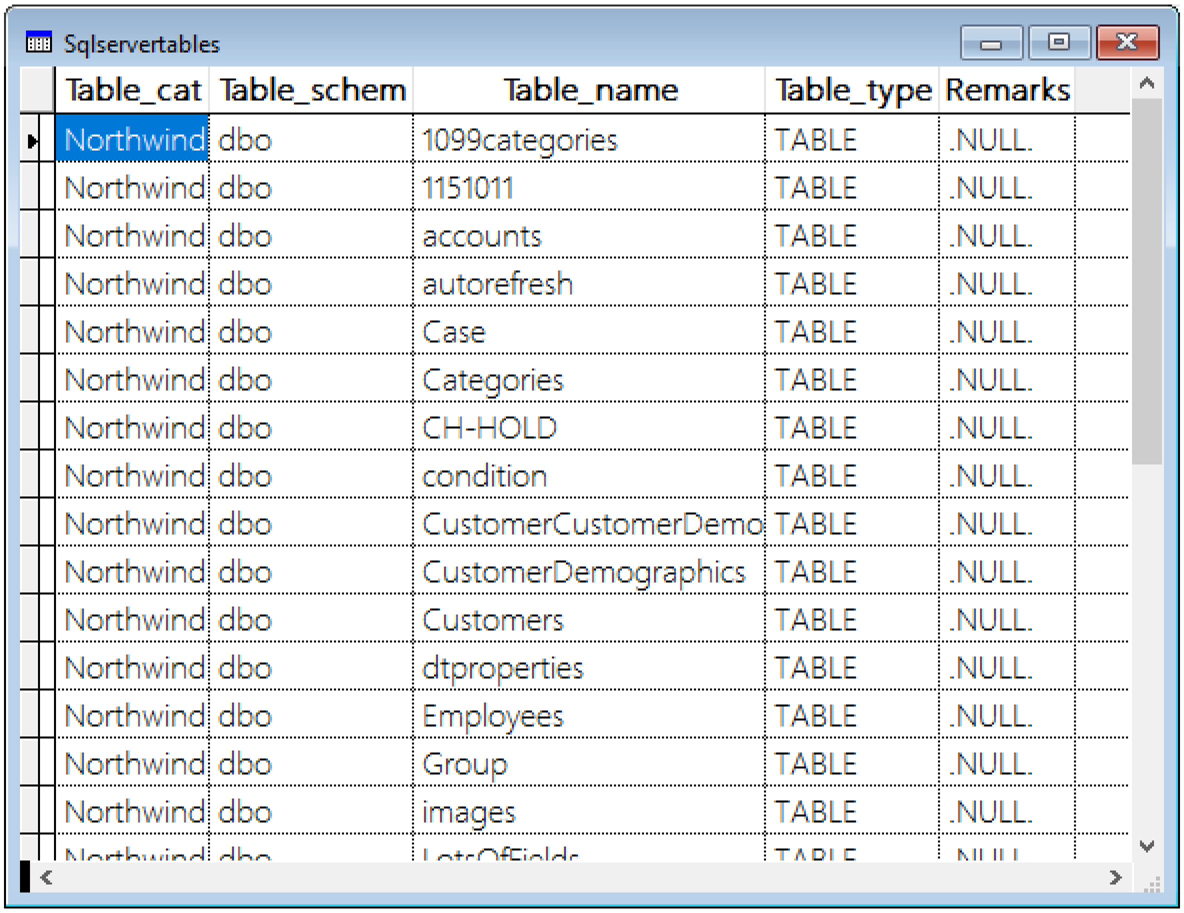
```
sqlsetprop(lnHandle, 'Transactions', 2)
text to lcSQL noshow
UPDATE Account SET Balance = Balance – 100 WHERE Account = ?lnAccount1 ;
UPDATE Account SET Balance = Balance + 100 WHERE Account = ?lnAccount2
endtext
llResult = sqlexec(lnHandle, lcSQL) > 0
do case
   case not llResult
      sqlrollback(lnHandle)
   case sqlcommit(lnHandle) < 0
      aerror(laError)
* do something about the error
endcase
sqlsetprop(lnHandle, 'Transactions', 1)
```

Use SQLSETPROP(nHandle, 'DisconnectRollback', .T.) to rollback a transaction if the connection is closed if a transaction is pending; the default is .F., meaning the transaction is committed.

Metadata discovery

VFP has two functions that retrieve metadata about an ODBC data source:

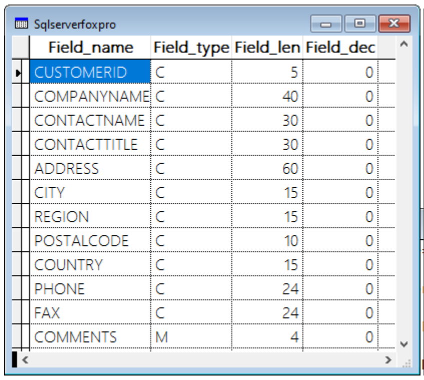
- SQLTABLES(nHandle [, cTableTypes] [, cCursorName]) creates a cursor containing information about the tables and optionally views. Pass "TABLE," "VIEW," "SYSTEM TABLE," a combination of those in a single comma-separated string (for example, "'TABLE', 'VIEW'"; note the values must be surrounded by single quotes), or an empty string for all types. **Figure 8** shows an example.



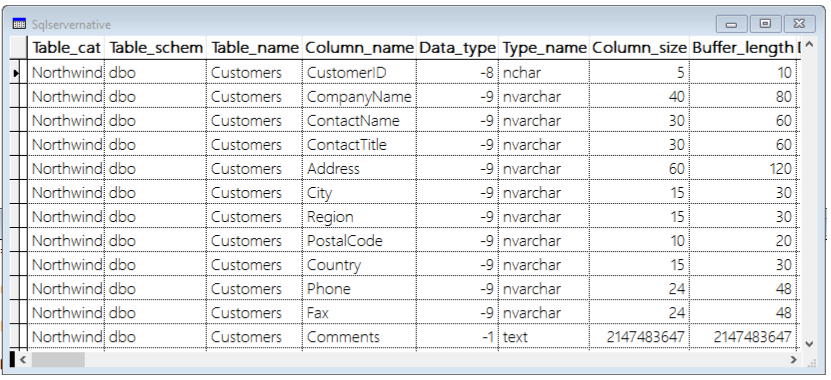
Table_cat	Table_schem	Table_name	Table_type	Remarks
Northwind	dbo	1099categories	TABLE	.NULL.
Northwind	dbo	1151011	TABLE	.NULL.
Northwind	dbo	accounts	TABLE	.NULL.
Northwind	dbo	autorefresh	TABLE	.NULL.
Northwind	dbo	Case	TABLE	.NULL.
Northwind	dbo	Categories	TABLE	.NULL.
Northwind	dbo	CH-HOLD	TABLE	.NULL.
Northwind	dbo	condition	TABLE	.NULL.
Northwind	dbo	CustomerCustomerDemo	TABLE	.NULL.
Northwind	dbo	CustomerDemographics	TABLE	.NULL.
Northwind	dbo	Customers	TABLE	.NULL.
Northwind	dbo	dtproperties	TABLE	.NULL.
Northwind	dbo	Employees	TABLE	.NULL.
Northwind	dbo	Group	TABLE	.NULL.
Northwind	dbo	images	TABLE	.NULL.

Figure 8. SQLTABLES() creates a cursor of tables in the ODBC data source.

- SQLCOLUMNS(nHandle, cTableName [, "FOXPRO" | "NATIVE"] [, cCursorName]) creates a cursor containing information about the specified table. Specify “FOXPRO” if you want data types specified as VFP data types or “NATIVE” to return information about columns using the data source’s format. In **Figure 9**, the left image shows the structure of a table using “FOXPRO” and the right image shows the structure using “NATIVE” (there are more columns for “NATIVE” than shown here).



Field_name	Field_type	Field_len	Field_dec
CUSTOMERID	C	5	0
COMPANYNAME	C	40	0
CONTACTNAME	C	30	0
CONTACTTITLE	C	30	0
ADDRESS	C	60	0
CITY	C	15	0
REGION	C	15	0
POSTALCODE	C	10	0
COUNTRY	C	15	0
PHONE	C	24	0
FAX	C	24	0
COMMENTS	M	4	0



Table_cat	Table_schem	Table_name	Column_name	Data_type	Type_name	Column_size	Buffer_length
Northwind	dbo	Customers	CustomerID	-8	nchar	5	10
Northwind	dbo	Customers	CompanyName	-9	nvarchar	40	80
Northwind	dbo	Customers	ContactName	-9	nvarchar	30	60
Northwind	dbo	Customers	ContactTitle	-9	nvarchar	30	60
Northwind	dbo	Customers	Address	-9	nvarchar	60	120
Northwind	dbo	Customers	City	-9	nvarchar	15	30
Northwind	dbo	Customers	Region	-9	nvarchar	15	30
Northwind	dbo	Customers	PostalCode	-9	nvarchar	10	20
Northwind	dbo	Customers	Country	-9	nvarchar	15	30
Northwind	dbo	Customers	Phone	-9	nvarchar	24	48
Northwind	dbo	Customers	Fax	-9	nvarchar	24	48
Northwind	dbo	Customers	Comments	-1	text	2147483647	2147483647

Figure 9. SQLCOLUMNS() creates a cursor with information about the columns in the specified table.

These functions have several things in common:

- The return value is 0 if the function is still executing, 1 if it's successful, or a negative number if it failed (use AERROR() to determine why).
- If cCursorName isn't passed, the cursor is named SQLResult.
- They can be used asynchronously; call the function repeatedly until it returns a non-zero value.

Using these functions is more complicated than it seems because different database engines and/or ODBC drivers work differently. The biggest issue is that the structure of the cursor varies. For example, with SQLTABLES(), SQL Server gives a column named TABLE_SCHEM for the schema while other drivers name it TABLE_OWNER or TABLE_SCHE. In addition, each driver has its quirks. For example, you have to pass an empty string to SQLTABLES() for the table type when using the Excel driver or no results are returned. This makes writing generic code that works with different database engines (such as Stonefield Query) more difficult.

Some database engines support defining relationships between tables. VFP doesn't have ODBC functions that retrieve information about relationships so you have to use the OpenSchema method of an ADODB.Connection object, opened with the OLE DB provider for the database engine if there is one, for that.

Specific ODBC Drivers

Microsoft SQL Server

According to https://docs.microsoft.com/en-us/sql/connect/connect-history, there are three generations of SQL Server drivers (the DLLs are located in C:\Windows\SysWOW64 for the 32-bit driver and C:\Windows\System32 for the 64-bit version):

- SQL Server (SQLSrv32.dll): the original SQL Server driver has been included with Windows since 98 and NT 4.0 and is still part of Microsoft Data Access Components (MDAC; https://en.wikipedia.org/wiki/Microsoft_Data_Access_Components). It's no longer being updated, but if you're not using features added to SQL Server 2005 or later, such as DATE columns, it has the advantage that it's already installed on your users' systems.
- SQL Server Native Client (SQLNCLI*nn*.dll, where *nn* is the version number): this was included with SQL Server 2005 through 2012. It's no longer being updated so there probably isn't a good reason to use this driver. The version number is part of the driver name; the latest one is "SQL Server Native Client 11.0."
- ODBC Driver for SQL Server (MSODBCSQL*nn*.dll, where *nn* is the version number): this is included with SQL Server versions after 2012. You can also download it from https://docs.microsoft.com/en-us/sql/connect/odbc/download-odbc-driver-for-sql-server; be sure to download the 32-bit version since that's what VFP needs. It's the driver that supports new SQL Server features going forward so this is the

preferred driver, especially if you're using features added to SQL Server 2005 or later. The version number is part of the driver name; the latest one is "ODBC Driver 17 for SQL Server."

There is a difference in the connection string for the drivers besides the driver name. The SQL Server driver uses SQL Server authentication if you specify a username and password or Windows authentication if not. The newer drivers, however, require "trusted_connection=yes" in the connection string if the username and password are omitted or the connection will fail.

There are also differences in the behaviors of the drivers as shown in **Table 3**.

Table 3. Difference in behavior between SQL Server drivers.

Data Type	SQL Server	Native Client	ODBC Driver
DATE	Comes in as Character (e.g. "2021-01-01")	Comes in as Date	Comes in as Date
VARCHAR(MAX)	Comes in as Memo	Comes in as Character(0) in VFP, Memo in VFPA	Comes in as Character(0) in VFP, Memo in VFPA
IMAGE	Comes in as General	Comes in as General	Comes in as General
VARBINARY(MAX) containing images	Comes in as General (corrupted)	Come in as Memo containing image bytes as BMP in VFP, General (corrupted) in VFPA when MapBinary is .F. Come in as VarBinary(0) in VFP, Blob in VFPA when MapBinary is .T.	Come in as Memo containing image bytes as BMP in VFP, General (corrupted) in VFPA when MapBinary is .F. Come in as VarBinary (0) in VFP, Blob in VFPA when MapBinary is .T.

The solution to handling Date columns using the SQL Server driver is to cast the column to DateTime, then convert it back to Date in VFP. For example:

```
text to lcSQL noshow
SELECT CAST(OrderDate AS DATETIME) AS OrderDate
   FROM DriverTest
endtext
sqlexec(lnHandle, lcSQL, 'DriverTest')
SELECT CAST(OrderDate AS DATE) AS OrderDate ;
   FROM DriverTest ;
   INTO CURSOR DriverTest
```

The solution to handling VARCHAR(MAX) columns with the newer drivers is to cast them to TEXT:

```
text to lcSQL noshow
SELECT CAST(Comments AS TEXT) AS Comments
   FROM DriverTest
```

```
endtext
sqlexec(lnHandle, lcSQL, 'DriverTest')
```

This isn't necessary with VFP Advanced because it handles VARCHAR(MAX) columns properly (http://baiyujia.com/vfpdocuments/f_vfp9fix46.asp).

Images in IMAGE columns may or may not be valid General fields, depending on the format of the data in the column. For example, the recommended technique to save an image into a record is something like this:

```
INSERT INTO DriverTest
       (Photo)
   VALUES
       ((SELECT BulkColumn FROM Openrowset(Bulk 'Image.jpg', Single_Blob) as Image))
```

That gives an invalid General field when queried into VFP. The solution in this case is to use CURSORSETPROP('MapBinary', .T., 0) before issuing a query to force them to come into VFP as Blob.

For images in VARBINARY(MAX) columns, which is the preferred data type for images because IMAGE is deprecated, it's more complicated: using MapBinary makes them come into VFP as VarBinary(0) with the newer drivers, so you have to use both MapBinary and cast the column to IMAGE:

```
cursorsetprop('MapBinary', .T., 0)
text to lcSQL noshow
SELECT CAST(Picture AS IMAGE) AS Picture
   FROM DriverTest
endtext
sqlexec(lnHandle, lcSQL, 'DriverTest')
```

You don't have to use the cast with VFP Advanced because it handles VARBINARY(MAX) columns properly (the page linked above mentions VARCHAR(MAX) but it also fixes the issue for VARBINARY(MAX)).

One complication with having to cast columns is that you can't use SELECT * (although DBAs will tell you not to do that anyway). A couple of alternate solutions that handle all these issues are to use CursorAdapters or remote views.

With a CursorAdapter, specify a schema that contains the desired data types in the CursorSchema property and pass .T. to CursorFill to use that schema. For example:

```
loCA = createobject('CursorAdapter')
loCA.SelectCmd      = 'select * from DriverTest'
loCA.DataSourceType = 'ODBC'
loCA.DataSource     = lnHandle
loCA.CursorSchema   = 'ID C(36), OrderDate D, Entered T, EnteredBy C(10), ' + ;
   'Comments M, Notes M, Name C(50), Photo W, Picture W, IsCustomer L'
loCA.CursorFill(.T.)
```

With a remote view, use DBSETPROP(FieldName, 'DataType') to set the desired data type of the columns that are improperly handled:

```
dbsetprop('rv_DriverTest.OrderDate', 'DataType', 'D')
dbsetprop('rv_DriverTest.Comments',  'DataType', 'M')
dbsetprop('rv_DriverTest.Photo',     'DataType', 'W')
dbsetprop('rv_DriverTest.Picture',   'DataType', 'W')
```

However, this won't work with DATE columns using the SQL Server driver because it can't convert the Character value into a Date value; it gives a "Type conversion required by the DataType property for field 'Orderdate' is invalid" error on the USE statement for the remote view.

Microsoft Excel

As you likely know, Excel is the most popular database on the planet. I frequently use the Excel ODBC driver to read data from Excel documents and combine them with VFP or other data for reports.

There are two Excel drivers available:

- Microsoft Excel Driver (*.xls) (ODBCJt32.dll): this driver can only read from XLS files, not the newer XLSX ones, so its use is limited. However, it has the advantage that it's installed on your users' systems by default.
- Microsoft Excel Driver (*.xls, *.xlsx, *.xlsm, *.xlsb) (ACEODBC.dll): this driver can read both XLS and XLSX files so it's the preferred driver. It's installed as part of the Access Connectivity Engine (ACE), which also includes Access, dBase, and other drivers; you can download ACE from https://www.microsoft.com/en-ca/download/details.aspx?id=13255. Be sure to download the 32-bit version since that's what VFP needs.

The settings for the connection string are documented at https://docs.microsoft.com/en-us/sql/odbc/microsoft/odbc-jet-sqlconfigdatasource-excel-driver.

There are numerous quirks using the Excel driver:

- Excel must be installed. If you need to read from or write to an XLSX file without having Excel installed, use the XLSXWorkbook VFP project (https://github.com/ggreen86/XLXS-Workbook-Class).
- Only one user can open a document at a time, even if you specify "ReadOnly=1" in the connection string (the second and subsequent users get an "External table is not in the expected format" error).
- If the Excel workbook is protected by a password, you cannot connect to it, even by supplying the correct password with your connection string. If you try, you'll receive an "External table is not in the expected format" error.

- Each worksheet in the document is a separate table, named as the sheet name with a "$" suffix. Surround table names with square brackets, such as "[Sheet1$]."
- The Excel driver assumes that the first non-blank row in the worksheet contains column headers and uses those as the field names in the resulting cursor. According to the driver documentation, the optional FirstRowHasNames connection setting can be used to change this default behavior by using 0 for False and 1 for True. However, due to a known bug, the driver disregards this setting and the first non-blank row of data is always treated as column headings and is not included in the resulting cursor. There are two solutions to this:
    - Add a row 1 that contains column headings.
    - Hack the Windows Registry: change the FirstRowHasNames entry to 00 at HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Office\ClickToRun\REGISTRY\MACHINE\Software\Wow6432Node\Microsoft\Office\16.0\Access Connectivity Engine\Engines\Excel. In that case, fields are named F1, F2, and so on.
- Text columns come into VFP as Memo.
- By default, the driver scans the first eight rows of each column to determine the data type of the resulting field. According to the same driver documentation page mentioned in the previous point, you can adjust that by setting the MaxScanRows connection setting to a value between 1 and 16, or 0 to scan all rows. However, as with FirstRowHasNames, this setting is ignored. Instead, set the TypeGuessRows entry at the same Registry key mentioned above.
- In addition to using a WHERE clause, you can query a range of data in the FROM clause. For example:

  ```
  sqlexec(lnHandle, 'select * from [Sheet1$A1:C4]')
  ```

  Note that this may impact the data types of the columns if fewer than eight rows are read.
- The Excel driver doesn't support the CAST keyword. Instead, use VB functions like CStr or CInt (see https://support.microsoft.com/en-us/office/type-conversion-functions-8ebb0e94-2d43-4975-bb13-87ac8d1a2202 for details).

SFRegistryODBC

SFRegistry (defined in SFRegistry.vcx) is a class designed to read from and write to the Windows Registry. SFRegistryODBC is a subclass of SFRegistry that's specific for ODBC functionality. It has the following methods:

- CreateUserDSN(cDriverName, cDSN, cSettingString): creates a 32-bit user DSN using the specified string, DSN name, and connection string.
- DSNExists(cDSN): returns .T. if the specified DSN exists as a user or system DSN.

- GetDataSources(aArray): fills the specified array with the user and system DSNs available on this machine.
- GetDrivers(aArray): fills the specified array with the ODBC drivers installed on this machine.
- IsDriverInstalled(cDriverName): returns .T. if the specified driver is installed.
- RemoveUserDSN(cDSN): removes the specified DSN.

SFDataManager

Rick Strahl has a SQL helper class named wwSQL that helps with parameterized SQL statements. I used ideas from that class and other ideas shown to me by Bob Pierce to create SFDataManager, a class that takes a lot of the grunt work out of working with ODBC data sources. Here are some of the features of this class.

Automatically handles parameters

SFDataManager automatically handles parameterized statements. For example:

```
ldDate1 = {^1996-07-01}
ldDate2 = {^1996-07-30}
text to lcSQL noshow
select *
   from Orders
   where OrderDate between ?ldDate1 and ?ldDate2
endtext
loData.ExecuteSQL(lcSQL, 'Orders')
```

Note that since the expressions are evaluated within SFDataManager, anything specified in the expressions must be in scope in the SFDataManager object. That means:

- Variables must be declared as PRIVATE or PUBLIC rather than LOCAL.
- If you want to use something like “?(Thisform.txtName.Value),” you have to drop an instance of SFDataManager on the form so “Thisform” is valid.
- If you want to use something like “?(Cursor.FieldName)” from within a form with a private data session, you have to drop an instance of SFDataManager on the form or pass SET('DATASESSION') as the third parameter to ExecuteSQL so the cursor is in scope (more about data sessions later).

Alternatively, you can use parameter expressions surrounded with curly braces in the SQL statement. The benefit of using this syntax is that the cSQLWithValues property contains the SQL statement with evaluated values inserted, which isn’t passed to the data source but is provided for debugging and logging purposes. For example:

```
ldDate1 = {^1996-07-01}
ldDate2 = {^1996-07-30}
text to lcSQL noshow
select *
```

```
   from Orders
   where OrderDate between {ldDate1} and {ldDate2}
endtext
loData.ExecuteSQL(lcSQL, 'Orders')
```

Internally, the ExecuteSQL method converts the SQL statement to:

```
select * from Orders where OrderDate between ?p1 and ?p2
```

and sets p1 to the value of ldDate1 and p2 to the value of ldDate2. The value of cSQLWithValues is:

```
select * from Orders where OrderDate between 07/01/1996 and 07/30/1996
```

As an alternative to both of these, you can manually call AddParameter to register parameters and their values for expressions that won't be in scope for SFDataManager. For example:

```
loData.AddParameter('ldDate1', {^1996-07-01})
loData.AddParameter('ldDate2', {^1996-07-30})
text to lcSQL noshow
select *
   from Orders
   where OrderDate between ?ldDate1 and ?ldDate2
endtext
loData.ExecuteSQL(lcSQL, 'Orders')
```

Collaborates with SFConnectionManager

SFDataManager expects that either its oConn property is set to an instance of an SFConnectionManager object or there's a global variable named goConn that does. It calls the Connect method of that object and uses its nHandle property.

Handles private data sessions

If you have forms with private data sessions that use SFDataManager, you can either have a single global SFDataManager object for your application and pass SET('DATASESSION') as the third parameter to ExecuteSQL so cursors are created in the forms' data sessions, or you can drop an instance of SFDataManager on each form so it lives in the form's data session. I prefer the latter, so I can use calls such as Thisform.oData.ExecuteSQL().

Converts VFP to SQL Server syntax

I've used SFDataManager to convert VFP applications from using local tables to SQL Server. One thing that helped reduce the work was being able to pass a VFP SQL statement with minimal changes to ExecuteSQL because that method handles some (but not all) syntax differences between the two platforms. For example, suppose this is the original code:

```
select * ;
   from Orders ;
   where between(OrderDate, ldDate1, ldDate2) ;
   into cursor ResultSet
```

Set the lConvertVFPSyntax property of the SFDataManager object to .T. so it automatically converts VFP syntax. Then for each SQL statement, surround it with TEXT and ENDTEXT lines, add "?" in front of ldDate1 and ldDate2, and call ExecuteSQL:

```
text to lcSQL noshow
select * ;
   from Orders ;
   where between(OrderDate, ?ldDate1, ?ldDate2) ;
   into cursor ResultSet
endtext
loData.ExecuteSQL(lcSQL)
```

Notice I didn't pass a cursor name to ExecuteSQL; it knows from the SQL statement to create a cursor named ResultSet. Internally, this SQL statement is converted to:

```
select * from Orders where OrderDate between ?ldDate1 and ?ldDate2
```

Notice it automatically removed the semi-colons and INTO CURSOR clause and converted the BETWEEN() function to a BETWEEN clause.

The FinalizeSQLStatement method, called from ExecuteSQL, does the work. It currently handles the VFP syntax shown in **Table 4**, because those are the conversions I've needed so far, but of course this could be expanded to cover more cases.

Table 4. SFDataManager converts VFP syntax in SQL statements to SQL Server syntax.

VFP	SQL Server
ALLTRIM(expr)	RTRIM(LTRIM(expr))
BETWEEN(expr, a, b)	expr BETWEEN a AND b
CDOW(expr)	DATENAME(dw, expr)
CHR(value)	CHAR(value)
CNT(expr)	COUNT(expr)
DATE()	GETDATE()
EMPTY(expr) (also handles NOT)	expr = ''
INLIST(expr, a, b, …)	expr IN (a, b, …)
INT(expr)	CAST(expr AS INT)
NVL(expr, value)	ISNULL(expr, value)
TRANSFORM(expr)	STR(expr)
TRANSFORM(expr, format)	FORMAT(expr, format)

VFP	SQL Server
TTOC(expr, 2)	CONVERT(CHAR(5), expr, 108)
TTOD(expr)	CAST(expr AS DATE)
.F.	0
.T.	1
==	=
{^YYYY-MM-DD}	'YYYY-MM-DD'

Getting a record by ID

You can, of course, create a cursor by passing a SQL statement to ExecuteSQL. However, if you just want the content of a single record and you know its primary key value, you can call GetRecordByID:

```
uResult = Object.GetRecordByID(cTable, uID [, cCursorName] [, nDataSession]])
```

If you pass the name of the cursor to create, GetRecordByID returns .T. if it succeeded and .F. if not. If you omit that parameter, the method returns a data object (that is, an object created with SCATTER MEMO NAME) if it succeeded or NULL if not. A data object is handy for scenarios where a cursor cannot be used, such as in different data sessions or passing to a COM or .NET object.

GetRecordByID automatically converts VARCHAR(max) to memo fields in the resulting cursor.

“Updatable” cursors

Two methods, SaveData and DeleteRecord, provide the ability to add, update, and delete records.

To add a record, create a cursor with the desired columns (it doesn’t have to contain all the columns in the table; only those that exist are updated in the table) and add a record to it, then call SaveData. You can create the cursor manually (CREATE CURSOR) but the easiest way is to call CreateStructureCursor. Here’s an example:

```
loData.CreateStructureCursor('Customers', 'CustRecord')
insert into CustRecord ;
   (CustomerID, CompanyName, ContactName, Country) ;
   values ('SSI', 'Stonefield Software', 'Doug Hennig', 'Canada')
loData.SaveData('Customers', , .T.)
```

Here’s the syntax for CreateStructureCursor:

```
lResult = Object.CreateStructureCursor(cTable, cCursorName [, nDataSession])
```

Here's the syntax for SaveData:

```
lResult = Object.SaveData(cTable [, cCursorName | oData [, lNew  [, nDataSession]]])
```

The second parameter is the name of the cursor (if it isn't specified, the current work area is used), the current record of which is used as the source of the values. It can also be a data object. If the table has a primary key column, you don't have to pass the third parameter: if it doesn't exist in the source or is empty, the record is added; otherwise, the record with the specified key value is updated.

SaveData handles some columns specially:

- If the record is being added and a column named Entered exists, it's set to the current date and time. If EnteredBy exists, it's set to the value of the cUserName property.
- Whether the record is added or updated, if a column named Updated exists, it's set to the current date and time. If UpdatedBy exists, it's set to the value of the cUserName property.
- Blank Date and DateTime values in the source are written to the table as NULL if the column allows nulls.
- Strings are trimmed so they don't have trailing spaces when written to a varchar column.

To delete a record, call DeleteRecord:

```
lResult = Object.DeleteRecord(cTable [, cCursorName | oData [, nDataSession]])
```

The parameters are similar to those for SaveData.

You can use these methods for data maintenance. For example, in a data entry form, bind the controls to fields in a cursor. To edit a specific record, use GetRecordByID or ExecuteSQL to populate the cursor. To add a record, call CreateStructureCursor to create an empty cursor. When the user clicks a Save button for either a new or edited record, call SaveData. To delete a record, call DeleteRecord.

Logging

SFDataManager can log what it does if desired. Set lLogSQL to .T. and set cSQLLogFile to the name of a table to log to. SFDataManager creates this table if it doesn't exist, using the structure shown in **Table 5**.

Table 5. The structure of the log table used by SFDataManager.

Name	Type	Contents/Purpose
Date	T	The date and time

Name	Type	Contents/Purpose
User	C(20)	The name of the user; set to the value of cUserName
Module	C(120)	The name of the function, procedure, or method that called ExecuteSQL
Line	I	The line number of the function, procedure, or method that called ExecuteSQL (may not be available if running in an EXE with "debug info" turned off)
Elapsed	N(8, 4)	How long the SQL statement took to execute
Records	I	How many records were affected by the query
SQL	M	The SQL statement
Error	M	If an error occurred, the text of the error message
Hash	N(20)	A hash of the records in the result set, 0 if there are no records, or -1 if there are more than 1000 records (which would take too long to hash). This can be used to compare runs that should give the same results; if the hash is different, something about the query is different.

Summary

While accessing remote databases from VFP can be as simple as using SQLSTRINGCONNECT(), SQLEXEC(), and SQLDISCONNECT(), there can be a lot more to it than that. I hope this document has helped you understand the detail involved in using SPT commands in VFP.

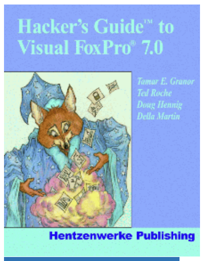
Biography

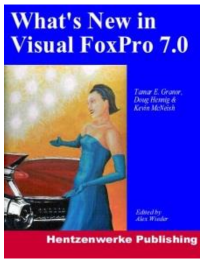
Doug Hennig is a partner with Stonefield Software Inc. He is the author of the award-winning Stonefield Query; the award-winning Stonefield Database Toolkit (SDT; now open source); the MemberData Editor, Anchor Editor, and CursorAdapter and DataEnvironment builders that come with Microsoft Visual FoxPro; and the My namespace and updated Upsizing Wizard in Sedna.

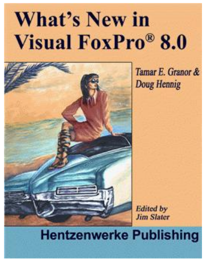
Doug is co-author of VFPX: Open Source Treasure for the VFP Developer, Making Sense of Sedna and SP2, the What's New in Visual FoxPro series, Visual FoxPro Best Practices For The Next Ten Years, and The Hacker's Guide to Visual FoxPro 7.0. He was the technical editor of The Hacker's Guide to Visual FoxPro 6.0 and The Fundamentals. He wrote over 100 articles in 10 years for FoxRockX and FoxTalk and has written numerous articles in FoxPro Advisor, Advisor Guide to Visual FoxPro, and CoDe.

Doug spoke at every Microsoft FoxPro Developers Conference (DevCon) starting in 1997 and at user groups and developer conferences all over the world. He is one of the organizers of the Southwest Fox (http://www.swfox.net) and Virtual Fox Fest (https://virtualfoxfest.com) conferences. He is one of the administrators for the VFPX VFP community extensions Web site (http://vfpx.org). He was a Microsoft Most Valuable

Professional (MVP) from 1996 through 2011. Doug was awarded the 2006 FoxPro Community Lifetime Achievement Award (http://tinyurl.com/ygnk73h).




What's New in Visual FoxPro 7.0
Tamar E. Granor, Doug Hennig & Kevin McNeish
Hentzenwerke Publishing


What's New in Visual FoxPro® 8.0
Tamar E. Granor & Doug Hennig
Edited by Jim Slater
Hentzenwerke Publishing