Visual FoxPro Client Server Handbook

Third Edition

Author: Stamati Crook
redware research ltd
stamati.crook@redware.com
Date: 18 August 2004
Version: 5.1
Document: vfphandbook60.doc

Shareware Licence

Acknowledgements

Third Edition August 2004

Thanks to the FoxPro team for finally making an improvement to the excellent client server capabilities of Visual FoxPro and to **redware** customers worldwide for employing our consultancy services to convert applications to the client server platform.

Second Edition September 2001

Thank you this time to Victor, Phong and especially James for helping build really big database systems during our roller coaster ride at First Telecom. Thanks also to the job market for letting me take a break and update this book.

First Edition December 1996

Thank you to the technical team at F1 Computing Systems past and present for eight years of implementing FoxPro projects. All of my FoxPro experience has resulted from working with Ian, Phong, James, Danny and David on various projects during my eight years at F1. They still have the best training courses and FoxPro team in the UK.
Thank you especially to James Thornton at F1 Computing who put me straight on a few things regarding SQL Server. Any errors remaining in this book are, of course, down to me and I apologise in advance for them. Please email me with your comments, good and bad.

1. Contents

CONTENTS 3

1. OBJECTIVES 5

2. TUTORIAL 6

Create a Connection 6
Create a Remote View 6
Updating Data 7
Table Buffering 7
Using Parameterised Remote Views 7
Executing a Stored Procedure 8
Creating a client server Form 8
Create a Cursor with the CursorAdapter Object 9
Summary 11

3. CONNECTIONS 13

ODBC OPEN DATABASE CONNECTIVITY 13
ODBC String Connection 14
ODBC Data Source Administrator 14
ODBC Performance Tips 15
CONNECTION DESIGNER 16
Data Source 16
Data Processing Options 16
TimeOut Intervals 17
CREATE CONNECTION 17
PROGRAMMING CONNECTIONS 17
Connection Properties 17
SETTING DEFAULTS 18
ASYNCHRONOUS CONNECTIONS 19

4. REMOTE VIEWS 21

VIEW DESIGNER 21
Creating a Remote View 21
Fields 22
Filter 23
Join 24
Order By 25
Group By 25
Update Criteria 26
PARAMETERISED VIEWS 28
CREATE SQL VIEW 28
VIEW PROPERTIES 29
Shared Connections 30
Update Criteria 30
WORKING WITH REMOTE VIEWS 31
PARAMETERISED VIEWS 31

5. CURSORADAPTER CLASS 33

CURSORADAPTER 33
Using parameters to filter data 34
Updating data 35
CursorAdapter Event Model 36
Attaching an Existing Cursor 38

Using the CursorAdapter Builder *38*
Building a CursorAdapter with cabuilder.prg *38*
USING CURSORADAPTER OBJECTS IN A FORM 38

6. DATA BUFFERING 39

SPECIFYING DATA BUFFERING 39
SAVING CHANGES 40
REVERTING CHANGES 41
DETERMINING UPDATES 41
ERROR HANDLING 42
COMMIT AND ROLLBACK 44

7. FORM AND DATA ENVIRONMENT PROPERTIES 45

8. OPTIMISING VIEWS AND CURSORADAPTERS 46

Advanced Options *46*
Remote View Properties *47*
Recommended Settings *48*

9. SQL PASS-THROUGH 49

Asynchronous Mode *50*
Batch Mode *50*
SQL PASS THROUGH AND DATA BUFFERING 51
PREPARING SQL STATEMENTS 52
ODBC EXTENSIONS 53
STORED PROCEDURES 54

10. ADO AND XML 55

11. OFFLINE VIEWS 56

12. CLIENT-SERVER APPLICATION DESIGN 57

PERFORMANCE BOTTLENECKS 57
PARAMETERISED VIEWS 57
LOCAL VALIDATIONS 58
TRANSACTIONS 58
STORED PROCEDURES 59
SQL PASS THROUGH 59
CONNECTIONS 59
MICROSOFT TRANSACTION SERVER 60

13. DATABASE MAINTENANCE 61

UPSIZING A FOXPRO DATABASE 61
Visual FoxPro Upsizing Wizard *61*
Upsizing Considerations *64*
DATA MANIPULATION LANGUAGE 66
CREATE TABLE *67*
ALTER TABLE *67*
CREATE INDEX *67*
GENDBC.PRG *68*

14. INDEX 69

2. Objectives

This Handbook aims to cover all the features required to implement client-server database applications using Visual FoxPro as the front end. It is designed as a reference text for experienced programmers and should be used in conjunction with the FoxPro programmers' manual.

An experienced programmer with access to a client-server database should a few days to work through the handbook and discover the implementation tricks and traps of client-server development.

After reading this handbook you will have the knowledge to:

- Configure and optimise shared connections with the database server.
- Create and optimise remote parameterised views.
- Execute SQL pass through onto the database server both synchronously and asynchronously.
- Upsize and modify database definitions on the server from FoxPro.
- Design and optimise a client-server application.

This book was originally written for Visual FoxPro 6.0 and has been updated and tested fully with Visual FoxPro 7.0. The CursorAdapter class that is new with Visual FoxPro 8.0 is described in detail and some guidelines provided to help you chose which of the various FoxPro technologies are relevant to your client server project.

All examples except for the CursorAdapter classes will work with Visual FoxPro 6.0 and 7.0.

The second and third edition of the Handbook does not cover aspects specific to server-side database issues. All sections are relevant whatever back end database you are using. Please note that we advise you to load up enough test data to stress test your application during the development phase as the performance aspects of client-server are very different from native FoxPro databases.

REDWARE also publishes the SQL Server Handbook covering all the essential aspects of implementing a database with Microsoft SQL Server.

Please feed back any comments on the text of the book especially if there are any important points or topics that you feel are missing from the book or any errors you have uncovered. You may contact the author by email on **stamati@redware.com**.

Please look at our web site on **www.redware.com** for information on related products and updates to this handbook.

Remember that this book is shareware and register your copy at ***http://www.redware.com/register.html***. You have a choice of making a small payment or registering a link regarding the handbook from your website. Please recognise the effort we have put into creating this resource and register now.

Register your shareware now at **www.redware.com** *with either a small payment or a link from you website.*

3. Tutorial

This tutorial section introduces client-server database programming with Visual FoxPro with a minimum of fuss and bother. We explore the major concepts and get to know our way around the PUBS database that ships with SQL Server.

First let us get to grips with some concepts:

- FoxPro uses the Windows ODBC drivers to communicate with the client-server database. You will need to install the appropriate driver on each workstation.
- Remote Views can be created in a database container to reference a client server database query. The retrieved data is represented as a FoxPro cursor.
- Cursors can now (since Visual FoxPro 8.0) be created with the object oriented CursorAdapter base class.
- Cursors employ table buffering to control the timing of updates sent to the database server.
- Direct communication with the database server is possible using SQL pass through commands.

Create a Connection

A connection can be created with reference to an ODBC datasource defined on the workstation or directly using a connection string. The following example connects to the PUBS database on a locally installed copy of SQL Server using the system administrator username and password.

```
lnHandle = SQLSTRINGCONNECT(;
    'DRIVER=SQL Server;SERVER=(local);UID=sa;PWD=;DATABASE=pubs')
```

The `SQLSTRINGCONNECT` function will return a positive integer referring to the connection handle if successful. Incorrect passwords will usually result in a user prompt while other errors will be trapped by the `AERROR` function.

Alternatively, if you have defined an ODBC driver for the workstation you may access the datasource definition directly with the `SQLCONNECT` command:

```
lnHandle = SQLCONNECT('dsnPubs','sa','')
```

You can test a connection by using `SQLTABLES` to get a list of the defined tables returned from the database server into a cursor. The PUBS database should contains several tables including AUTHORS and PUBLISHERS.

```
? SQLTABLES(lnHandle,'table')
BROWSE
```

We can also execute database commands directly against the database server. The following command will return a read-only copy of the AUTHORS table into a local cursor called fred:

```
? SQLEXEC(lnHandle, 'select * from authors', 'fred' )
```

`SQLEXEC` returns a 1 if completed successfully, 0 if still processing asynchronously, and –1 if there is an error.

Create a Remote View

A remote view can be created programmatically but requires a database container to be open for update. Most valid SQL SELECT clauses can be used to define a remote view.

```
CREATE DATABASE dbctutorial
CREATE CONNECTION conpubs ;
    CONNSTRING 'DRIVER=SQL Server;SERVER=(local);UID=sa;PWD=;DATABASE=pubs'
CREATE SQL VIEW vueauthor REMOTE ;
    CONNECTION conpubs SHARED AS select * from authors
SELECT 0
USE dbctutorial!vueauthor
BROWSE
```

Updating Data

The default definition for a remote view is to create read only data. We must change the view definition by setting the `SENDUPDATES` parameter to true either in code as shown below or by using the database designer:

```
? DBSETPROP( 'vueauthor',"VIEW","SENDUPDATES",.t.)
```

Now we can browse and alter data. Notice the error messages from the server if you attempt to set an illegal value (for example by entering letters into the ZIP field).

```
USE dbctutorial!vueauthor
BROWSE
```

Table Buffering

Table buffering can be useful if you want to update several records in a cursor and control the moment they are updated onto the database. Table buffering is set on an already open cursor using the `CURSORSETPROP` function below:

```
SELECT vueauthor
? CURSORSETPROP("Buffering",5)
```

You can now make any changes you require without sending anything to the server. You can cancel all the changes as shown below:

```
REPLACE ALL au_fname WITH 'fred'
? TABLEREVERT(.t.)
```

Or you use the `TABLEUPDATE` function to update onto the database server:

```
? TABLEUPDATE(.t., .t.)
```

Using Parameterised Remote Views

One of the primary design criteria for a scalable client-server application is that only small amounts of data are sent to the workstation at any one time. Parameterised Views allow FoxPro variables to be defined as part of the selection clause so that only the required records are retrieved.

In the example below, a parameterised view is created on the authors table that retrieves only the authors that live in a single state:

```
MODIFY DATABASE dbctutorial
CREATE SQL VIEW vuestateauthors REMOTE CONNECTION conpubs SHARE as
select * from authors where state = ?lcstate
? DBSETPROP('vuestateauthors','view','sendupdates',.t.)
```

The parameterised view is first opened with the NODATA clause so that no data is retrieved from the server. The required value for the State is specified in the appropriate variable and selecting the empty cursor and issuing a REQUERY() command will interrogate the server and retrieve only the required records.

```
USE vuestateauthors NODATA
lcstate ='CA'
```

```
REQUERY()
lcstate ='TX'
REQUERY()
```

Executing a Stored Procedure

Another great advantage of a client-server database is that programs, known as stored procedures, can be defined to run on the server minimising the traffic passing to and fro between server and workstation.

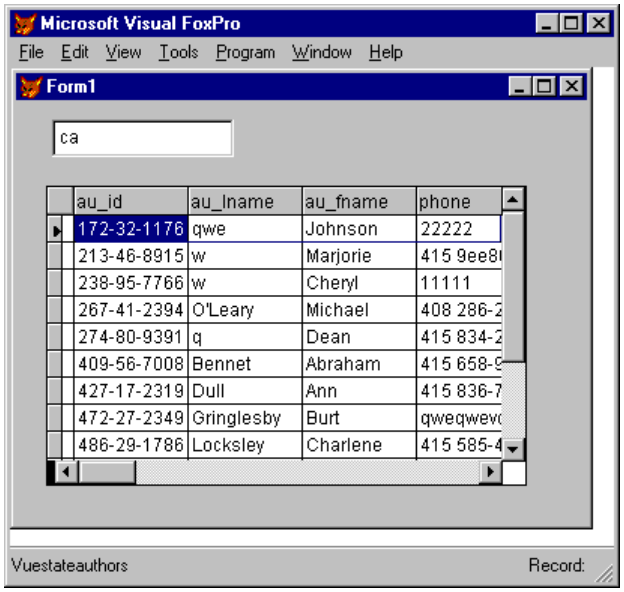
The following example shows how to call the BYROYALTY stored procedure which returns a cursor of author identifiers who receive a particular percentage royalty:

```
OPEN DATABASE dbctutorial
lnhandle = SQLCONNECT( 'conpubs')
? SQLEXEC(lnhandle,'exec byroyalty 40','fred')
```

Creating a client server Form

FoxPro forms behave in a similar fashion with client-server remote views as they do with local views and local tables. Care is required to manage the retrieval of the data and updating using table buffering but, otherwise, the behaviours of the cursor is the same in all three environments.

We shall create a form that prompts the user to enter the state required and then retrieves the corresponding authors into a grid for updating. The VUESTATEAUTHORS parameterised view is employed for this form, which will operate with record level buffering on the cursor.



First create the form and right click to add the VUESTATEAUTHORS remote view into the data environment. Select the properties for the cursor and set the `NODATAONLOAD` property to true so that no data is retrieved from the server when the form is opened. Also specify the `BUFFERMODEOVERRIDE` property to 3 for record level buffering.

Now drag the image of the cursor from the data environment onto the form to create a grid control.

Finally, create a textbox control that allows for a two character string to be entered for the user to specify the state required and add the following code the refresh the parameterised view and the grid:

```
lcstate = THIS.Value
SELECT vuestateauthors
=REQUERY()
THIS.Parent.GRdVuestateauthors.Refresh
```

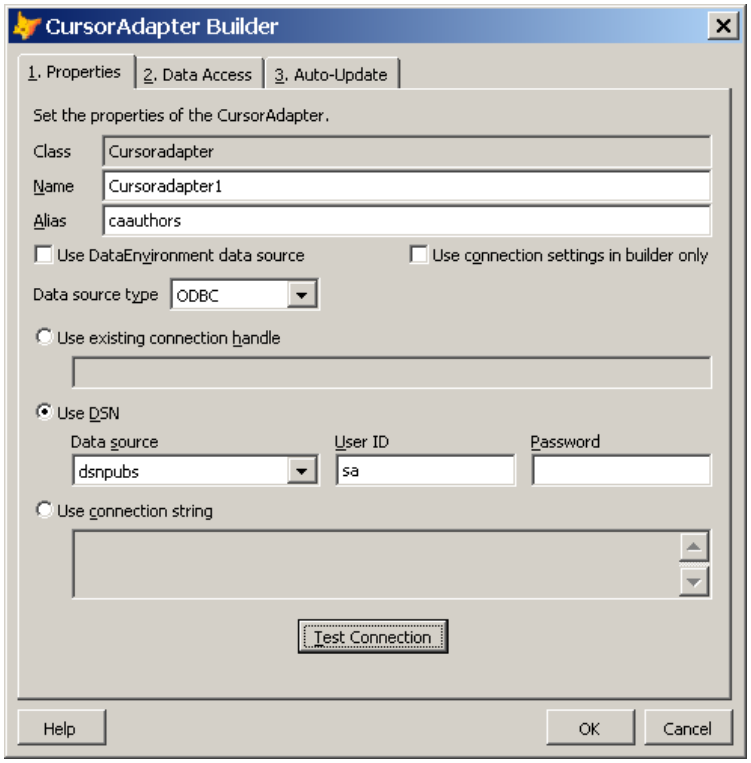
Run the form and enter CA into the textbox. The grid should refresh and allow you to update the author records. Try entering an invalid zip code to check that a response is returned from the server as you move to the next record in the grid.

Create a Cursor with the CursorAdapter Object

CursorAdapter classes are the new object oriented way to access both local and remote client server databases as well as access data using ADO RecordSets and XML.

An easy way to build a CursorAdapter object directly into a form is to create a new form and richt-click to view the data environment. Ignore the initial prompt for a table and right-click to add a CursorAdapter. Right-click on the newly added CursorAdapter and select the builder.

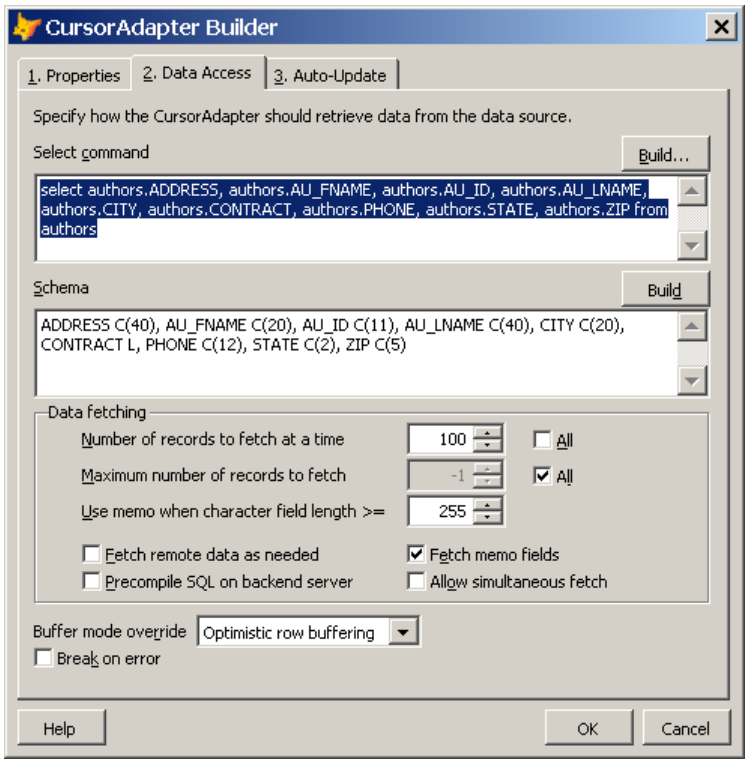
Name the alias and select the ODBC datasource type and define the connection characteristics. An existing workstation datasource is used below and the password and username specified.



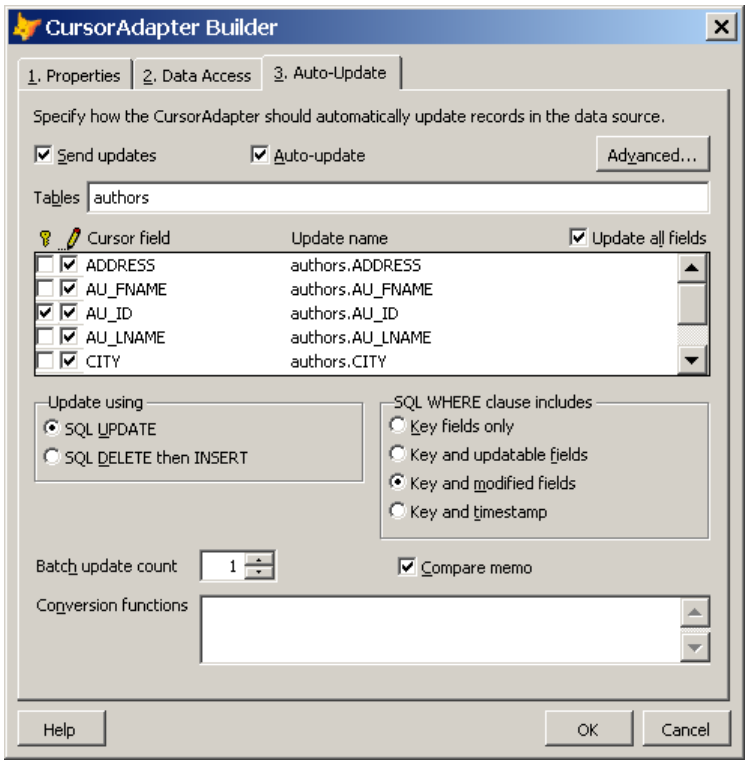
Click the `Data Access` tab and enter the following SELECT command:

```
SELECT * FROM authors
```

Now press the BUILD button just above the select command to select the fields required and automatically update the SCHEMA values. Select all the fields in the AUTHORS table:



Now select the AUTO-UPDATE tab to spacify the SENDUPDATES and AUTOUPDATE check boxes and the updatable fields. Specify the AU_ID field as the primary key.



The CursorAdapter object is now complete and will create a cursor that updates the table on the client server database. Close the builder down and drag the fields onto the form to create a form in the usual way.

Summary

We have covered most of the concepts involved in creating client-server applications with Visual FoxPro in this short tutorial:

- A connection that uses the Windows ODBC/OLEDB middleware to communicate with the client-server database must be defined before FoxPro can communicate with the server.
- Once the connection has been made, commands can be executed directly against the database server to retrieve data or to execute stored procedures.
- A Remote View can be defined in a Database Container to coordinate the retrieval of data from the server into a local cursor. The `SENDUPDATES` property must be set to allow updates back onto the server controlled by the table buffering properties of the cursor.
- CursorAdapter objects offer an object oriented way of accessing data stored locally or remotely or via ADO or XML with a single programmable object. CursorAdapters can be defined with a builder tool and properties stored persistantly in a Visual Class Library.
- Parameterised views are used to control the retrieval of small sets of data as required by the application. The art of client-server application design is to break up access to the data into small queries that will allow the application to scale easily to hundreds of users.

- Forms are created in a similar fashion to standard FoxPro data access with attention required for the specification of parameterised views and to the control of the table buffering.

The remainder of the handbook describes each of these areas in detail and offers instruction in the fine tuning and optimisation of FoxPro access to database server.

4. Connections

Visual FoxPro controls all communication with client-server databases by using a software component called a connection.

All connections require the appropriate ODBC drivers to be installed on the local workstation. A single application may communicate with several different client-server databases by defining a connection for each datasource.

Connections can be defined in a Database Container for use together with FoxPro remote views as a very convenient way to access data. They may also be defined programmatically when used together with SQL statements that are passed directly through to the server.

Many parameters influence the behaviour and performance of connection. Fox example, synchronous connections are relatively easy to handle programmatically and will cause the application to wait until all the data requested from the server has arrived. Asynchronous connections allow access to data and continues receiving data from the server in the background.

Client-server performance can be influenced by the number of connections open at any one time. An application with many users should minimise the number of open connections for each client. Visual FoxPro aids this by allowing remote views to be controlled by a single shared connection.

This section looks all aspects of Connections defined programmatically or in a database container together with properties and settings that can be used to improve performance and control behaviour.

ODBC Open DataBase Connectivity

ODBC (Open DataBase Connectivity) is the Microsoft Windows middleware for connecting to external data sources from Windows applications. It has proved very successful, and many applications have direct support for ODBC including the Microsoft Office product range.

ODBC terminology has changed several times over the last decade with OleDB and ADO drivers providing better functionality and performance particularly with SQL Server. Non-Microsoft database vendors usually support ODBC for their drivers and the latest drivers for SQL Server operate as ODBC drivers even if their underlying functionality is the more recent ADO.

The ODBC drivers need to be installed on each workstation that requires access to the data source and will work with all supporting applications once they are installed. Specific drivers are not required for Visual FoxPro.

> *You may wish to access FoxPro data from other Windows applications. The FoxPro ODBC driver can be download from the Microsoft website. Just search for ' FoxPro ODBC driver'.*

ODBC String Connection

A full ODBC connection string can be used to create a connection to the server. In this case there is no need to define a datasource on the workstation as only the relevant drivers need to be installed.

This string may contain references to the driver, server, username, password, calling application, workstation name, and database name to create a connection to SQL Server for example:

```
DRIVER=SQL Server;SERVER=(local);UID=sa;PWD=;APP=Microsoft(R)
Windows NT(TM) Operating System;WSID=REDNTS001;DATABASE=pubs
```

The connection string may also be used in connection with a datasource that has been defined on the workstation:

```
DSN=odbpubs;uid=stamati;pwd=fred
```

The `SQLSTRINGCONNECT` function can be used to test the connection from FoxPro and should return a positive number greater than zero if sucessfully connected to the database server. Use `AERROR` to resolve any problems.

```
lcSQL = [DRIVER=SQL
Server;SERVER=(local);UID=sa;PWD=;APP=Microsoft(R) Windows NT(TM)
Operating System;WSID=REDNTS001;DATABASE=pubs]
LnHandle = SQLSTRINGCONNECT( lcSQL )
```

ODBC Data Source Administrator

An ODBC datasource may be defined on a workstation using the control panel application to provide consistent details to any application running on the workstation. The datasource can include username and password information as well as the default database to use.

The datasource can be one of three types:

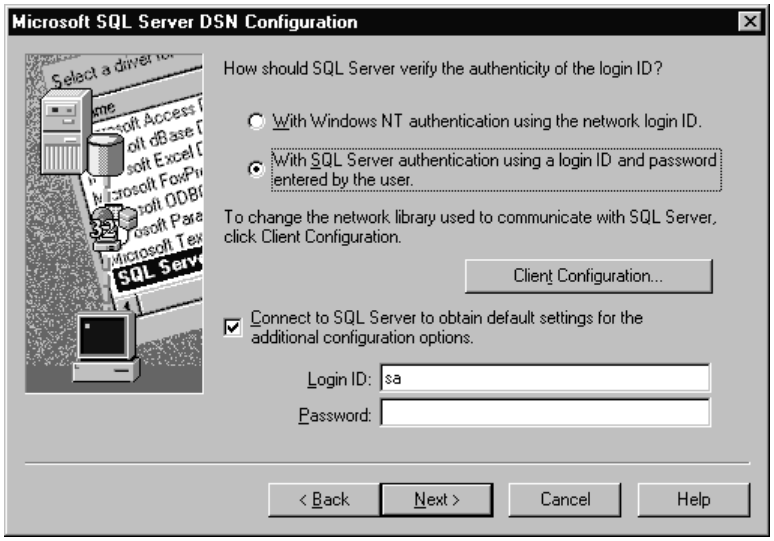
- User DSN is available only to the current workstation user.
- System DSN is the most commonly used option and is available to all applications and users on the workstation.
- File DSN is rarely used but has the advantage that it can be installed easily by copying a file into the relevant folder.

  The author has had trouble configuring File DSNs to work with FoxPro.

Some points to note when installing an ODBC datasource for SQL Server:

- Storing the username and password in the datasource allows any user on the workstation to access the database. Consider omitting the username and password and providing the information within the FoxPro connection.
- Specify the `(local)` server when referring to the locally installed copy of SQL Server
- Authentication is more straightforward from the application programmers point of view when using SQL Server security instead of Microsot Authentication. The username and password can reflect the permissions of the application rather than that of the individual user.
- Take care when using the system administrator (sa) password. Preferably ask the database administrator to give you a development username and password.

- Try to set the default database in the datasource to avoid accidentally creating items in the wrong database.
- Test the connection every time.



Specifying an ODBC Driver for a SQL Database

Test the ODBC connection from FoxPro using the `SQLCONNECT` command. The function should return a positive number greater than zero. Use `AERROR` to determine any problems:

```
lnHandle = SQLCONNECT('dsnPubs','sa','')
```

ODBC Performance Tips

ODBC is the middleware that connects Visual FoxPro to the server. The performance of the ODBC driver will significantly affect client-server performance. Always use the latest drivers as more recent drivers may offer enhanced performance.

The latest SQL Server drivers from Microsoft use ADO technology and offer a high level of performance which more than matches the DB-LIB library that has traditionally been used with SQL Server and Sybase implementations.

It may be worthwhile looking at third party suppliers of ODBC middleware for non-Microsoft databases. Note that performance may be significantly faster when connected to SQL Server in comparison with non-Microsoft databases such as Oracle and Sybase.

In rare instances the use of a specific library or DLLs or ADO ActiveX controls may provide faster access to data. Connecting to a database with ODBC is usually the preferred method for FoxPro, but you may need to look at these options for specific requirements.

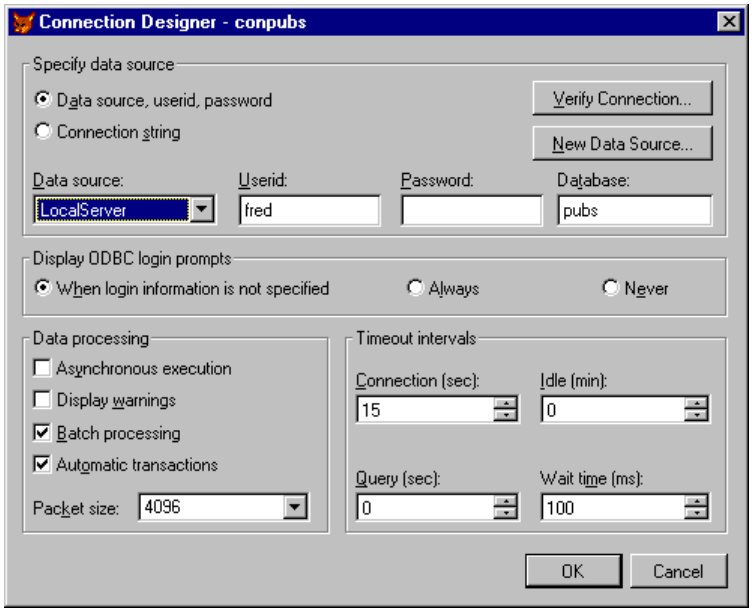
> *Connection Pooling will improve performance in applications, such as a COM server where connections are being rapidly created and destroyed.*

Connection Designer

An easy way to modify connection properties is to use the Project Manager to open the required database container. Once the database container is opened the following command may also be used to modify the connection:

```
OPEN DATABASE dbcNwind
MODIFY CONNECTION conNwind
```

The Connection designer is also available from the **DATABASE** menu of an open database container or by right clicking on the background.



Modifying Connection Properties with the Database Designer

Data Source

The Connection designer allows an existing ODBC datasource to be used or for the full connection string to be entered. The full connection string has the advantage that only the required driver needs to be installed on the workstation.

> *SQL Server can be configured for integrated security and will automatically sense the login permissions on the database from the currently active NT login.*

Data Processing Options

The defaults shown in the connection properties window above are fine for most requirements:

- Synchronous execution is the best option when beginning to work with client-server systems so leave the **ASYNCHRONOUS** checkbox empty. FoxPro will wait for all the required data to be retrieved before continuing with processing.

- Displaying the warnings will display any ODBC authentication issues to the user. The alternative is to trap and display the errors under program control (using the AERROR function).
- Leave batch processing and the automatic transactions set to True to allow FoxPro to automatically handle the commit and rollback activity on the database.

TimeOut Intervals

Various timeout intervals can be defined for a Connection to help prevent problems with runaway queries. These can normally be left on the default values.

In a development environment or with user defined selection criteria you may find that the server takes a long time to return a query (even if the connection is asynchronous). In this context it is advisable to set the Query timeout so that control returns to the workstation for a runaway query (which is eventually cancelled on the server).

CREATE CONNECTION

Connections can be defined programmatically in a Database Container with the CREATE CONNECTION command. In this case a Database Container must be open for modification.

```
CREATE CONNECTION [ConnectionName | ?]
   [DATASOURCE cDataSourceName]
   [USERID cUserID] [PASSWORD cPassWord]
   [DATABASE cDatabaseName]
   | CONNSTRING cConnectionString]
```

The following example creates a connection in the SALES database container using a datasource that has already been defined on the workstation:

```
SET DATABASE TO sales
CREATE CONNECTION conNwind DATASOURCE odbNwind
```

Alternatively the full connection string can be used:

```
CREATE CONNECTION conpubs ;
   CONNSTRING 'DRIVER=SQL Server;SERVER=(local);UID=sa;PWD=;DATABASE=pubs'
```

All the properties for a Connection object can also be set programmatically into the Database Container with the DBSETPROP command. For example, the username could be changed with the following command:

```
? DBSETPROP('conpubs','connection','userid','fred')
```

Programming Connections

Connections can be created programmatically with the SQLCONNECT or the SQLSTRINGCONNECT command depending on whether an ODBC connection is created on the workstation or a full connection string is used.

Both function returns a connection handle as a positive integer greater than zero if a connection is made sucessfully. Problems can be resolvved with the AERROR comand.

```
lnHandle = SQLCONNECT( 'dsnpubs', 'sa', '')
```

Connection Properties

Some of the advanced properties can be set programmatically or with the Connection Designer and are described below:

- The **BATCHMODE** is used for multiple SQL pass through queries with a single command. Each query can be returned simultaneously in batch mode or one by one.
- **DISPLOGIN** allows the system to prompt for the user password each time the connection is used.
- The **CONNECTBUSY** property is read-only but is useful for determining whether an existing query is active.
- The **PACKETSIZE** defaults at 4096 (4K) and may be changed if the multiples of the record size are significantly smaller or larger than the default.
- **TRANSACTIONS** may be set to manual and **SQLCOMMIT** and **SQLROLLBACK** used to update the server. A simpler approach might be to set the buffering on the cursor to table level and control server updates programmatically.

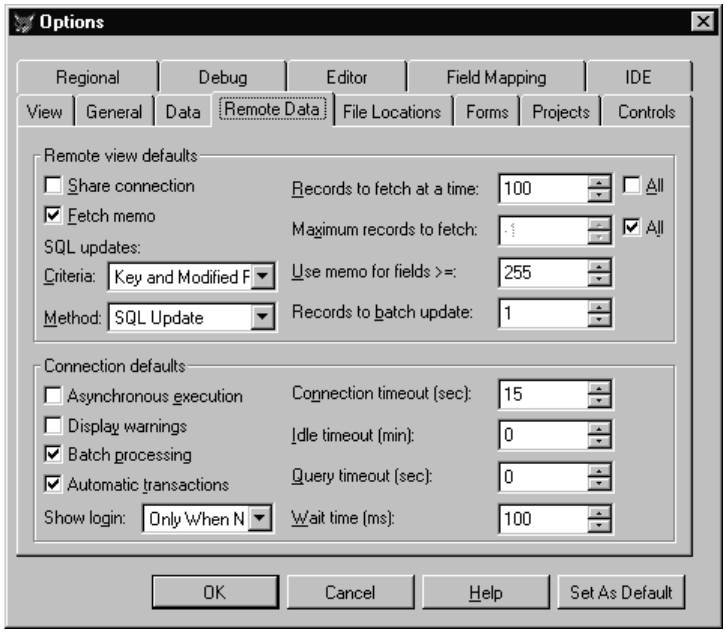
For very sophisticated access, the ODBC handle may be determined with a connection property and the ODBC driver interrogated directly with low level operating system calls.

The connection handle used for the current cursor can be determined with the CURSORGETPROP() function by looking at the CONNECTHANDLE property. The SQLSETPROP() function can then be used to change connection properties before a subsequent requery.

Setting Defaults

Default values for some of the options discussed above may be set for the FoxPro environment in the Remote Data tab of the Options window which is available from the FoxPro Tools menu.

Remote Data Options Window



Default options for Connections may be specified programmatically using the **SQLSETPROP** function with a zero Connection handle. To default all Connections to asynchronous processing the following code is required:

```
SQLSETPROP( 0, 'asynchronous', .T.)
```

Similarly the default cursor properties may be set by using the **CURSORSETPROP** command with a zero workarea. To set the default buffering to table level buffering for all cursors use the following syntax:

```
CURSORSETPROP( 'buffering', 5, 0 )
```

Asynchronous Connections

Asynchronous processing is an important property of a Connection in that control is returned to the application while a query is still running. The number of records to fetch from the server at any one time is specified for each remote view and the application will continue running once the first set of records has been returned. The remaining records are retrieved in the background.

The default number of records to retrieve is 100 and is determined by the **FETCHSIZE** property of the view. Using a view that has 10,000 records to be retrieved from the server may take some minutes with a synchronous connection. An asynchronous connection will retrieve the first 100 records and then return control to the application.

The following command will set up a Connection for **ASYNCHRONOUS** queries so that program control will return whilst the cursor is being populated with data returned from the server. You may need to execute the second **SQLEXEC** several times before the data begins to be retrieved. A zero indicates that data is being retrieved or that FoxPro is waiting for the first results. A negative value is an error and a 1 indicates that all the data has been retrieved:

```
lnHandle = SQLCONNECT('dsnDataSource','sa','')
SQLSETPROP(lnHandle,"Asynchronous",.T.)
```

```
? SQLEXEC(lnHandle, [SELECT * FROM largeTable])
? SQLEXEC(lnHandle, '')
```

Retrieving more records is performed automatically in the background but is forced to operate immediately by issuing a GO command, for example GO 1000, to force FoxPro to retrieve the first 1,000 records or a GO BOTTOM to force FoxPro to retrieve all the records synchronously. Be careful when issuing this type of command when a large amount of data is being retrieved.

> *`ASYNCHRONOUS` queries can cause problems when a Connection is being used for accessing several tables. Use the `CONNECTBUSY` property and the `SQLCANCEL` functions to control the connection. In general `SYNCHRONOUS` connections are easier to handle and you should filter the query to an acceptable number of records.*

5. Remote Views

Remote Views are stored in a Database Container and allow client server data to be retrieved and updated as if the table was a local FoxPro table. Views can also be defined to refer to local FoxPro tables to provide code compatability for systems that access local or remote data according to the installation (a separate Database Container is defined for each configuration).

Remote Views allow for transparent access to external ODBC data and also maintain some degree of logical data independence from the physical definition of the underlying data. Various components are required to create remote views successfully:

- An ODBC driver must be installed on the workstation to act as the middleware linking the operating system to the database server.
- A Connection is defined in the Database Container or programmatically to refer to the ODBC datasource.
- A Remote View is defined in a Database Container using the View Designer.

Views are created in a Database Container using the View Designer. This allows the required tables to be selected and any join conditions specified between them. The required fields, selection condition, order sequence, and update criteria are also specified. Local Views can access data from local tables whilst Remote Views use ODBC to connect to a database server.

Triggers cannot be defined for a View.

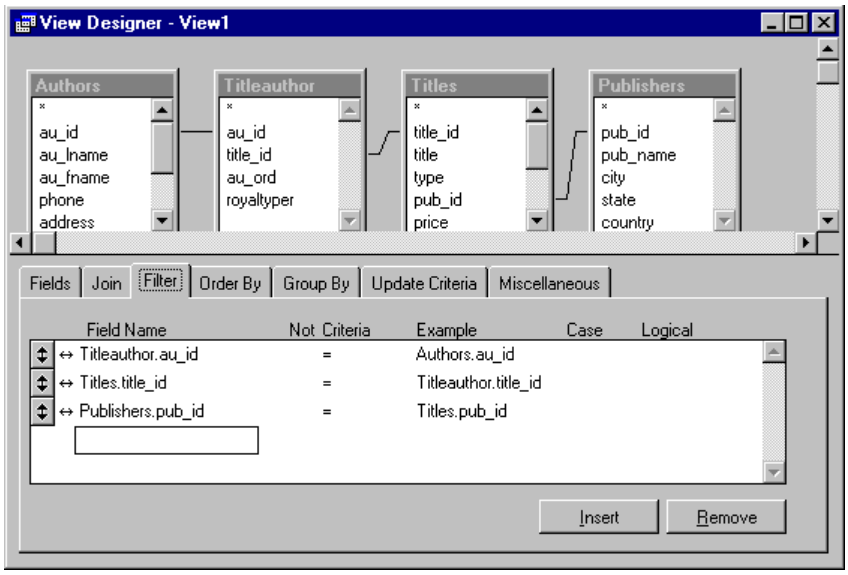
View Designer

The View Designer allows for the visual definition of the SQL statement that makes up the View. The tables that make up the View, the available fields, and the selection criteria are all defined visually along with the ability to group the table and calculate summary information.

Various field and view properties can be set with the view designer or by using the `DBSETPROP` command. For example, the `RULEEXPRESSION` property of the View provides table level validation for the view regardless of the data source but cannot be set using the View Designer.

Creating a Remote View

Tables are added to the View by Rightclicking in the View Designer and selecting the Add table option. If more than one table is added, the system will prompt for the fields that define the relationship to be specified and add them into the FILTER expression as shown below.



Adding tables with the View Designer

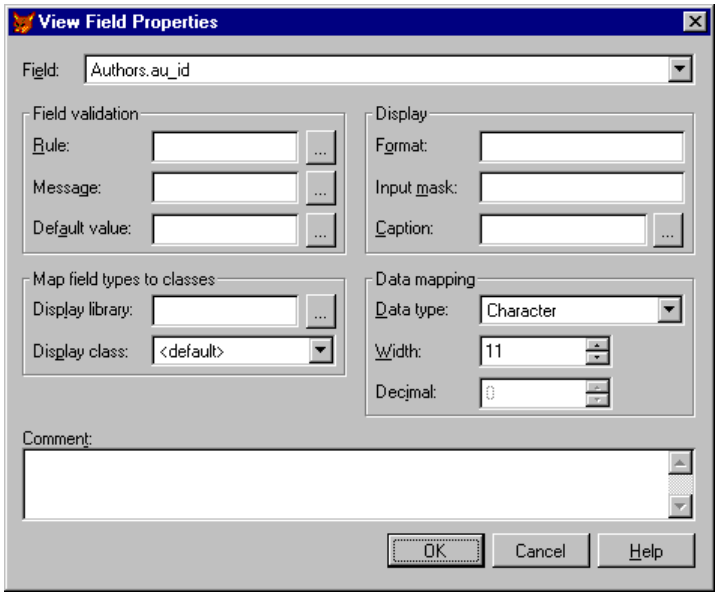
> *FoxPro uses a filter to create the join between tables. The JOIN Page must be used when inner and outer joins need to be specified.*

Fields

The fields must be selected using the FIELDS tab of the designer. It is advisable to limit the fields selected so as to reduce the traffic of data passing across the network. The primary keys will need to be included for the view to be updateable.

The `FIELDS` Page has a `PROPERTIES` button which allows field properties to be set on the view and stored in the FoxPro database container. Most of these properties are better set in the server database management system and are enforced for all client applications.

Field Properties Window

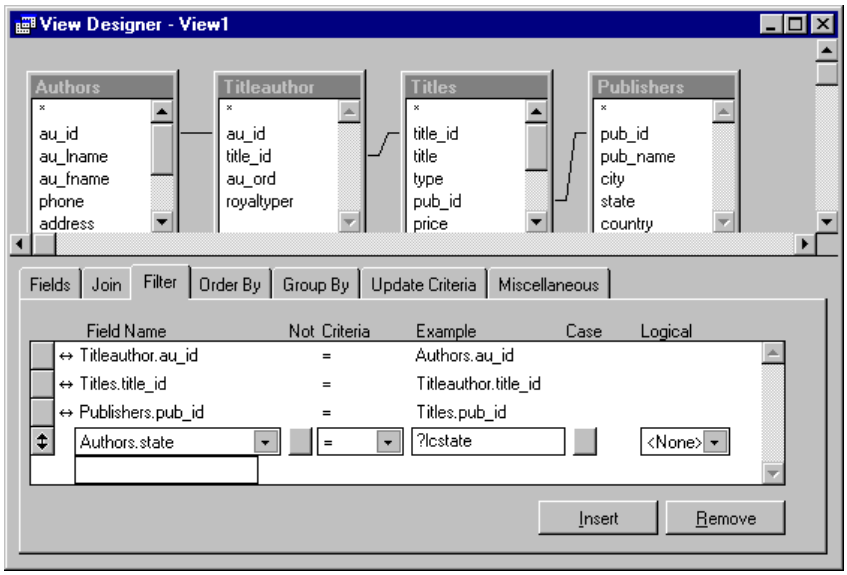


The Data Mapping will usually default to the field type most appropriate for FoxPro to receive values from the ODBC data source. Occasionally this may need to be modified and the View Designer allows the FoxPro data type to be modified to match particular processing requirements.

> *SQL Server does not have a date field as all dates are stored as date and time values. In this case it may be appropriate to change the mapping to a date datatype.*

Filter

The Filter window may be used to create a selection of records from the table. A parameterised view can be created by specifying a local FoxPro variable as part of the selection filter. In this case prefix the variable name with a question mark as shown below.



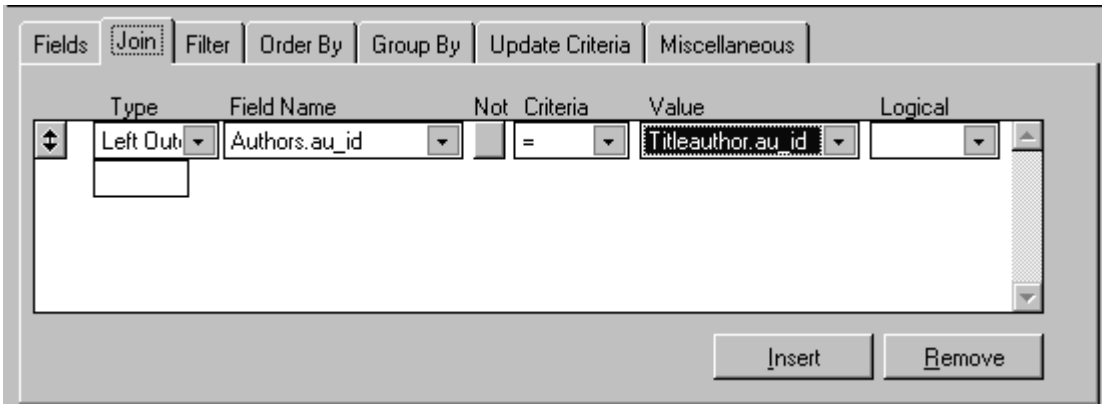
Specifying a Filter

Join

FoxPro uses the filter expressions to define the join condition whenever a new table is added to the view. This creates a Natural or Inner Join where only records that appear in both tables are selected.

Occasionally a more sophisticated Join is required where all the records of a table are displayed even if there are no records in the related table. This is called an Outer Join.

Authors may not have any records in the titleauthor table for example. A view that shows data from both tables and requires all Authors records even if there are no entries in the titleauthor table requires a Left Outer Join between the two tables. This is achieved by removing the expression from the Fields Page and including it on the Join Page specifying that a Left Outer Join is required.



Specifying a Left Outer Join

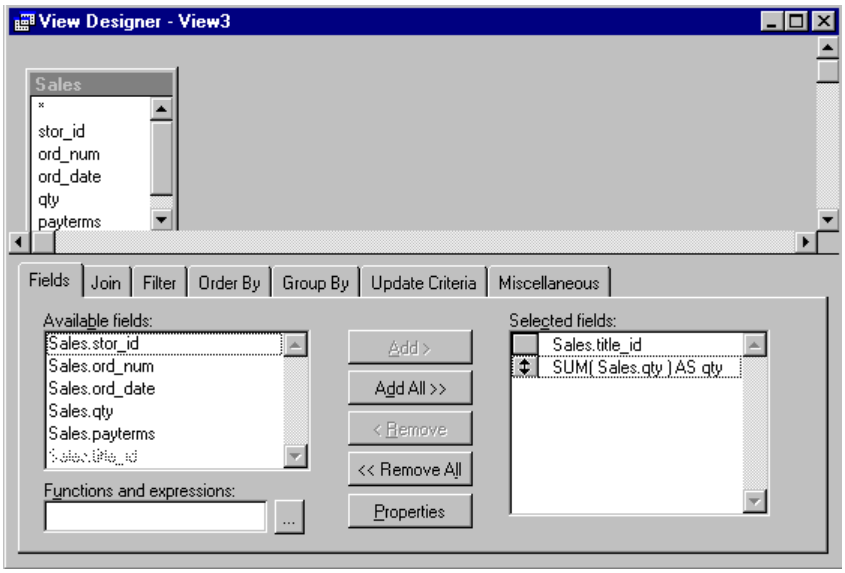
Order By

The Order By Page passes the sequence required onto the Server which returns the results set in the required order.

> *Take care when requesting large results sets that the order by is optimised on the server. In some cases it is more efficient to index the remote view locally after the view is opened.*

Group By

The Group By window is used to make the grouping selection operate on aggregate fields. These are specified using the Functions and Expressions item on the Fields Page using the appropriate SQL aggregate clause.



Specifying an Aggregate Clause

> *The field name assigned to the aggregate field normally defaults to EXP. This can be changed by modifying the fieldname with the AS keyword following the expression in the View Designer.*

The SQL-92 standard requires that all fields that are not modified with an aggregation function must be included in the Group By. In the above example, which displays the Author ID, Title ID, and the sum of the Sales Quantity, both the Author ID and the Title ID must be in the Group By. Visual FoxPro is not as strict as the SQL Server ODBC in this respect.

> *The `HAVING` clause does not seem to be implemented in the View Designer but can be implemented using `DBSETPROP` to specify the SQL property.*

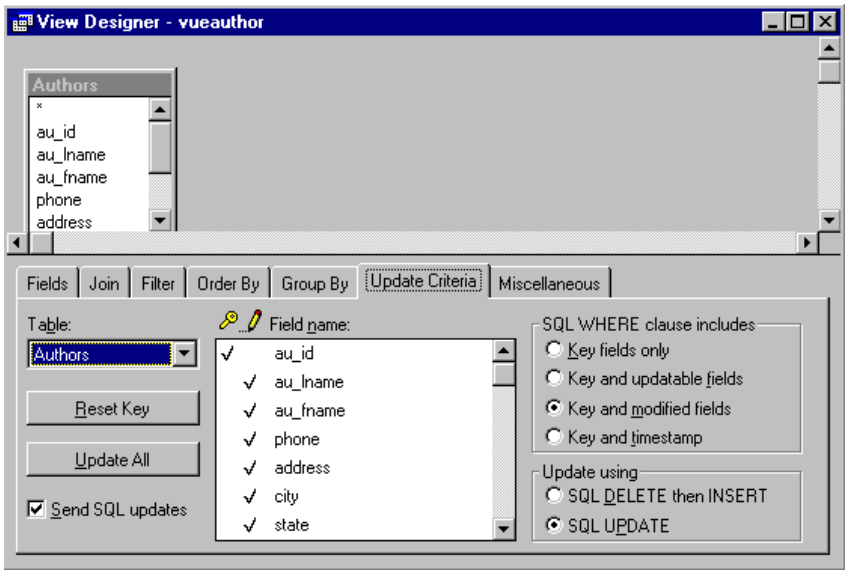
Update Criteria

The Update criteria specify how the data in a view is updated back onto the remote data source.

The Update Criteria can substantially affect the performance of the Remote View, and consideration of the nature and performance of the back end database is required to fully optimise the Remote View together with the Connection definition.

A primary key must be specified for FoxPro to determine which record to update in the external data source. Visual FoxPro may be able to determine the key field automatically but it can also be specified manually by clicking beside the appropriate field in the column indicated with a Key icon.

The fields that are modifiable by the program or end-user can also be specified for security and performance purposes by checking the column with a pencil icon.



Specifying the Update Criteria with the View Designer

A remote view can be specified on a join of several tables. The Key fields for each table should be indicated with the view designer to make all the tables updateable.

There are several optimistic record locking strategies that can be employed with a remote view.

- One scenario is that FoxPro can check the values of the key fields and all updateable fields to see if any changes have been made to any of the updateable fields by another user whilst the user was editing the record on the workstation. If any changes have been made by another user, the update transaction fails.
- Alternatively, the View can be configured to update the table by matching the key fields only. The update still occurs overwriting their changes if another user has made changes to the record.
- A timestamp field can be defined in the external data source (for SQL Server tables) which is maintained automatically each time the record is updated. Visual FoxPro can then check the timestamp field to see if other users have updated any of the fields in the record.
- The Key and Modified Fields option may be specified so that the remote view checks only the Primary Key and any fields that have been changed by the user. This allows several users to change the same record simultaneously as long as they do not update the same field.

The Key and Modified Fields option works well and is a good initial choice for an optimistic record locking strategy.

The `TABLEUPDAT`*) command has a* `FORCE` *parameter which will perform the update regardless of any changes made by another user.*

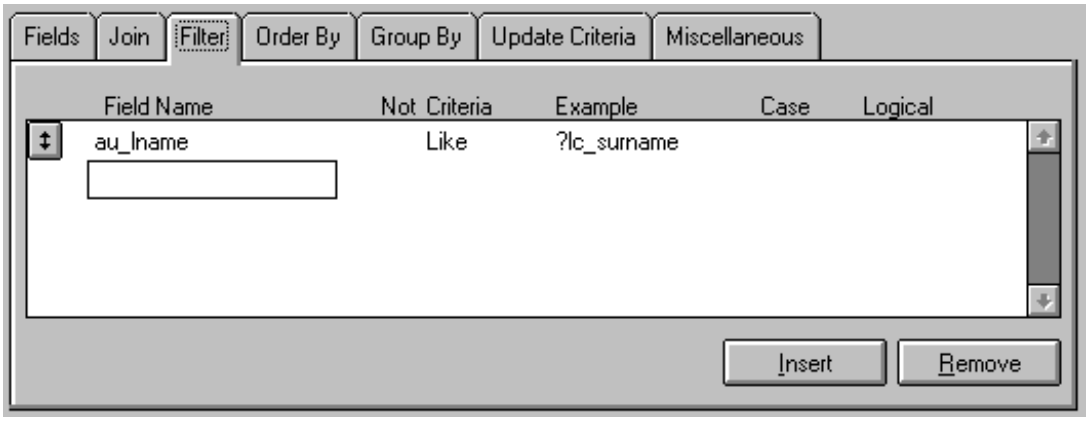
Some database engines do not allow SQL UPDATE commands on records and the record must be deleted and reinserted. This is allowed by checking the appropriate option.

Parameterised Views

Parameters may be defined for a View to limit the data selected in the SQL SELECT statement. The parameter can be employed programmatically and interactively.

The parameter is specified in the View Designer by placing a question mark before the parameter name in the EXAMPLE column of the selection criteria.

Specifying a Parameterised View



The end-user is automatically prompted to enter the parameterised value unless a memory variable with the same name as the parameter is defined before opening the View.

> *Take care to include a numeric string in quotes when prompted for a value.*

```
m.lc_surname = 'B%'
USE authorsurnameview IN 0
BROWSE
```

The parameter can easily be changed and the remote view updated with the **REQUERY**) syntax:

```
m.lc_surname = 'D%'
REQUERY( 'authorsurnameview' )
```

> *Parameter names can be defined for interactive use by enclosing the name in Quotation marks. For example: ?'Please enter the Customer Identifier'*

CREATE SQL VIEW

Views can be created programmatically in the Database Container by defining the SQL SELECT statement with the following syntax:

```
CREATE SQL VIEW [ViewName ] [REMOTE]
   [CONNECTION ConnectionName [SHARE]
   | CONNECTION DataSourceName]
   [AS SQLSELECTStatement]
```

A remote view must have the REMOTE keyword added and the connection details specified:

```
SET DATABASE TO dbcpubs
CREATE SQL VIEW authorsurnameview ;
   REMOTE CONNECTION odbpubs SHARED AS ;
   SELECT * FROM authors WHERE au_lname LIKE ?lc_surname
```

More complex views are also possible. A local view that contains the total value of each Product ordered by a single Customer might be defined as follows:

```
SET DATABASE TO sales
CREATE VIEW ProductTotal AS ;
   SELECT Products.product_id, Products.prod_name,
   SUM(Orditems.quantity),;
   SUM(orditems.unit_price*Orditems.quantity);
   FROM testdata!orditems, testdata!products;
   WHERE Products.product_id = Orditems.product_id;
   GROUP BY Products.product_id;
   ORDER BY Products.prod_name, Products.product_id
```

*The View may not be updateable if certain properties are not set. The **SENDUPDATES** property, for example defaults to False resulting in a read-only cursor.*

View Properties

Properties can be set for all of the objects in a Database Container including Tables, Views, Remote Views, and Connections. For many of the objects in the database container, the properties can be set both interactively using the appropriate designer or programmatically.

DBSETPROP is used to programmatically set database container properties. The function returns a True value if the property is successfully set in the Database Container. The following syntax can be used to set a Caption on a field in a Remote View specified in the Orders Database Container:

```
SET DATABASE TO orders
? DBSETPROP( 'customertotal.cnt_id', 'field', ;
   'caption', 'Orders' )
```

The following example sets the validation rule property for the view to call a validation function. This property cannot be seen or be set with the view designer:

```
SET DATABASE TO orders
? DBSETPROP( 'customertotal', 'view', ;
   'ruleexpression', 'custval()' )
```

See the help for DBGETPROP() for a list of the property names that can be set in the database container. Take care because Visual FoxPro does not check the validity of Database Container properties when validation rules are set programmatically.

Shared Connections

Database servers are often licensed per active connection or may have a performance bottleneck if too many connections are used. Reductions in the number of connections per user, preferably to one, will reduce server bottlenecks and/or cost.

A FoxPro connection can be shared for several queries. This is straightforward for synchronous queries but may cause some problems with busy connections for asynchronous queries.

FoxPro allows for Connections to be shared by specifying the `SHARED` keyword when defining a Remote View in program code or by selecting the Advanced Options when defining a View. Alternatively `DBSETPROP` can be used to set the `SHARECONNECTION` property to `True`.

> *The server may soon run out of connections with a very small number of users if this option is not used as the default for remote views.*

Update Criteria

The update criteria specified for a remote view can affect performance.

FoxPro will retrieve the record from the server to check if another user has changed any of the data on the updated record. The amount of data returned from the server depends on whether the key only, modified fields only, or all modified fields is checked. In a high transaction situation, the fastest option is the one that retrieves the least data from the server. In this case, checking only the primary key value will reduce network traffic at the cost of potentially overwriting another users changes. Note SQL Server allows the creation of a Timestamp field to control the resolution of these update conflicts with the minimum of traffic.

The update criteria correspond to the following properties which may be set in the view designer or by using `DBSETPROP`:

- `SENDUPDATES` – must be true for any updates to occur.
- `UPDATETYPE` – update in place or delete and insert.
- `WHERETYPE` – key and modified fields etc.

Working with Remote Views

```
USE [[DatabaseName.]Table | SQLViewName | ?]
   [IN nWorkArea | cTableAlias]
   [ONLINE]
   [ADMIN]
   [AGAIN]
   [NOREQUERY [nDataSessionNumber]]
   [NODATA]
   [INDEX IndexFileList | ?
   [ORDER [nIndexNumber | IDXFileName
   | [TAG] TagName [OF CDXFileName]
      [ASCENDING | DESCENDING]]]]
   [ALIAS cTableAlias]
   [EXCLUSIVE]
   [SHARED]
   [NOUPDATE]
```

A View may be defined in the Database Container that operates on a external data source. The View might be opened using the **USE…NODATA** command for implementation of a data entry function that only requires the addition of new records. This saves on server performance because no records are retrieved from the server.

In some instances the View may have already been opened by another program function. The open view can be used in the current form without executing the query again with the **NOREQUERY** clause of the use command:

```
USE remotecustomer ALIAS remotecustomer02 AGAIN NOREQUERY 5
```

The datasession where the remote view was originally opened can be specified in the NOREQUERY clause.

Alternatively the **NODATA** clause might be used to open a Remote Cursor without bringing down any records. This would be appropriate for a data window that is used only to add new records for example:

```
USE remotecustomer NODATA
```

Parameterised Views

A parameterised View will bring only the selected records down from the server. Minimising the number of records will reduce network traffic and improve performance. This is probably the single most important consideration when designing client-server applications.

A parameterised view can be defined in the View Designer or in program code. The following example creates a connection and a remote view in the database container:

```
OPEN DATABASE dbcPubs EXCLUSIVE
CREATE CONNECTION conPubs DATASOURCE odbPubs
CREATE SQL VIEW authorscontractview ;
   REMOTE CONNECTION conPubs SHARE AS ;
   SELECT * FROM authors WHERE contract = ?m.ll_Contract
```

This View is set up with all the current defaults. As the View is created, FoxPro will interrogate the server and determine the names of all the fields and of the primary keys. All fields apart from the primary key are set updateable by default and the update criteria are set according to the View defaults.

If the View properties are not setup correctly the view will not be updateable. For example the **SENDUPDATES** property must be True for the view to be updateable.

The parameterised View might initially be opened with the **NODATA** clause and then the value set for the parameterised query and the **REQUERY** command used to bring down the recordset from the server:

```
USE authorscontractview NODATA
….
m.ll_Contract = .T.
= REQUERY( 'authorscontractview' )
```

In this particular example, the Contract field is stored as a bit field on the server with values 0 and 1 instead of .T. and .F. FoxPro will automatically translate to the required value although the server values are also valid and 0 and 1 may be used as a value for the parameter.

The PREPARED property of a parameterised view can improve performance when requerying complex select statements by preparing (compiling) the statement on the server.

Each Cursor in the data environment of a form has a NODATAONLOAD option which can be used to open the table without any data. This is useful for forms which only add new records or where a parameterised query is required but the user has not yet specified the selection criteria.

6. CursorAdapter Class

The CursorAdapter class is new for Visual FoxPro 8.0 and provides an object-oriented base class for creating cursors that control access to data.

CursorAdapters work well with various different data sources:

- Data stored in native FoxPro format on your local disk drive or local area network servers.
- Client-server data stored in a central database management system accessed using ODBC drivers.
- ADO recordsets created on the workstation or by a middle tier business component server.
- XML documents.

CursorAdapters can be created programmatically or using the CursorAdapter builder that forms part of the visual design tools for the DataEnvironment of a Form. Third party tools such as the cabuilder from ***www.mctweedle.com*** build CursorAdapter class libraries directly from existing local or remote databases.

There are several advantages of using CursorAdapter classes:

- Object oriented inheritance allows a base class to be defined with application specific properties and methods.
- CursorAdapters are suitable for implementing systems that need to operate with either local or remote data according to the installation.
- CursorAdapter definitions can be created and stored in a visual class library and can be added into a DataEnvironment for a Form in a similar manner to local tables or views defined in a database container.

Some disadvantages include:

- CursorAdapters are objects and the object variable needs to remain in scope for the data to be available.
- Visual class libraries have a limit fo 255 characters for properties and these properties often need to be defined in a method of the class.
- Views in the database container have additional properties for the individual fields defined for the cursor.
- A cursor created by a CursorAdapter class is instatiated in program code instead of with the `USE` command.

CursorAdapters combine many of the best qualities of views defined in a database container with the flexibility of programmatic control and object oriented inheritance. Storing CursorAdapter definitions within a Visual Class Library in combinations with the builder tool allows visual design and persistance of the definitions (although this needs some improvement). The main benefit however is a single object oriented technique for accessing data from a variety of data sources including local tables, client-server tables, ADO recordset objects and XML documents.

CursorAdapter

A CursorAdapter uses a connection to communicate with a client server database when accessing data through with ODBC. A `SQLCONNECT` or `SQLSTRINGCONNECT` command

is issued to get a connection handle and the DATASOURCETYPE and DATASOURCE properties defined as shown below.

The SELECTCMD property is given a command that is executed on the server and a local cursor is created from the server data when the CURSORFILL method is called.

```
lnHandle = SQLCONNECT('dsnpubs','sa','')
loPubs = CREATEOBJECT('cursoradapter')
loPubs.DATASOURCETYPE ='ODBC'
loPubs.DATASOURCE = lnHandle
lopubs.alias = 'caAuthors'
loPubs.SELECTCMD=[select * from authors]
IF loPubs.CURSORFILL()
   BROWSE
   ELSE
   ? 'Error'
ENDIF
```

This cursor will be closed as the program ends and the variable holding a reference to the CursorAdapter goes out of scope.

An identical procedure is followed for local FoxPro data where the DATASOURCE is blank and the DATASOURCE type is set to 'NATIVE'.

Examples of CursorAdapter use with ADO and XML are shown later in this document.

Using parameters to filter data

Accessing client server data without specifying a filter is expensive as all the data must be retrieved from the server into a cursor on the local machine. A paramerised query may be specified in the SELECTCMD property to specify a selection of data.

The following example shows a CursorAdapter object retrieving an empty cursor by specifying the NODATA property when filling the cursor with the CURSORFILL command. This adds a WHERE 1=2 clause onto the SELECT command and returns an empty cursor with zero records.

```
lnHandle = SQLCONNECT('dsnpubs','sa','')
loPubs = CREATEOBJECT('cursoradapter')
loPubs.DATASOURCETYPE ='ODBC'
loPubs.DATASOURCE = lnHandle
loPubs.ALIAS = 'caAuthors'
loPubs.SELECTCMD=[select * from authors where state=?lcState]
IF loPubs.CURSORFILL(.F.,.T.)
   BROWSE TITLE 'NODATA'
ELSE
   ? 'Error'
ENDIF
```

An empty cursor might typically be created when initially running a form until the user can be prompted for some selection criteria. At this stage the parameters can be set and the CURSORFILL command reissued to populate the cursor as shown below. This process can be repeated as often as required.

```
lcState = 'TX'
```

```
IF loPubs.CURSORFILL(.F.,.F.)
   BROWSE TITLE [state=TX]
ELSE
   ? 'Error'
ENDIF
```

The SELECTCMD property of the CursorAdapter can be updated as required. This removes a major limitation of database container views in that a complex where clause can be specified for a cursor at run time.

```
lcWhere = [WHERE state='CA' AND contract=1]
lcSQL = [SELECT * FROM authors ] & lcWhere
loPubs.SELECTCMD = lcSQL
IF loPubs.CURSORFILL(.F.,.F.)
   BROWSE TITLE lcWhere
ELSE
   ? 'Error'
ENDIF
```

It seems that the SELECTCMD property for a CursorAdapter can contain any command that can be executed against the server. The following example executes a paramerised stored procedure on the server and returns the results as a cursor:

```
lnHandle = SQLCONNECT('dsnpubs','sa','')
loPubs = CREATEOBJECT('cursoradapter')
loPubs.DATASOURCETYPE ='ODBC'
loPubs.DATASOURCE = lnHandle
lopubs.alias = 'caRoyalty'
lnPerCent = 40
loPubs.SELECTCMD=[exec byroyalty ?lnPerCent]
IF loPubs.CURSORFILL()
   BROWSE
   ELSE
   ? 'Error'
ENDIF
```

Explain cursorschema here….

Updating data

The process for making a cursor updatable involves specifying properties to allow the system to automatically generate commands to execute on the server to update the changed records in the cursor.

The required properties are the same properties used to make a cursor created with SQL passthrough updatable and are the same as those set against a cursor created by a remote view:

- TABLES contains a list of the tables on the server to be updated.
- KEYFIELDLIST is a list of the fields that form the primary key of the tables.
- UPDATEABLEFIELDLIST is a list of the cursor fields that are updatable. Changes to the remaining fields are ignored.
- UPDATENAMELIST is a translation from local cursor field names to the field names on the server with the correct table prefix.

The primary key fields must be specified in the UPDATENAMELIST to allow the update commands to be automatically created by the system. They do not need to be updatable to update records or if the key values are automatically generated on the server when adding a new record.

The following example allows updates to occurs only on the **AU_LNAME** and **AU_FNAME** fields of the **AUTHORS** table:

Example here…

The **ALLOWUPDATE**, **ALLOWINSERT**, **ALLOWDELETE** and **SENDUPDATES** properties are true as the default settings for each CursorAdapter object. These values can be changed to restrict the apporpriate data modification operation.

The **WHERETYPE** property can also be set to specify the type of update command that is applied when records are changed. A good default value to chose is to set Key and Modified fields so that the system checks any modifed fielda for changes by other users but allows two users to change the same record provided they change different fields.

UPDATETYPE by default notifies the system to automatically generate a single update command to update records on the server. Change this setting to generate separate **DELETE** and **UPDATE** commands if required by your database server.

CONVERSIONFUNC is an iteresting property allowing local field values to be converted with a FoxPro function immediately prior to updating the data on the server. This is particularly useful for data stored as variable length character strings on the server but recognised as fixed length character strings in the local cursor. Specifiyng the **RTRIM** conversion function for each required field ensures that spaces are added onto the end of the character values in the server.

Example here…

Adding spaces onto variable length character fields may cause unpredicatable results when selecting records from the server which may now expect the correct number of trailing spaces when comparing values.

FoxPro automatically generates the update commands for any changes in the local cursor and controls their execution against the server system. This functionality works perfectly for both local and remote client server tables using ODBC. Properties such as **UPDATECOMMAND**, **UPDATEDATASOURCE**, and **UPDATEDATASOURCETYPE** allow detailed specification of update, insert, or delete commands if required.

CursorAdapter Event Model

CursorAdapter cursors have an event model that allows methods to be defined before and after major events such as filling a cursor or modifying or inserting data.

These events can be particularly useful for definging the equivalent of database validation and triggers in an object oriented fashion. Although is it perhaps better to exncapsulate this functionality on the database server, this approach might be useful if defining data validation rules on the client where a variety of database servers are used some of which may not allow data constraints or triggers to be specified.

The following code specifies a an updatable CursorAdapter class programmetically and then saves the object into a Visual Class Library.

```
lnHandle = SQLCONNECT('dsnpubs','sa','')
loPubs = CREATEOBJECT('cursoradapter')
loPubs.DATASOURCETYPE ='ODBC'
loPubs.DATASOURCE = lnHandle
loPubs.ALIAS = 'caAuthors'
loPubs.SELECTCMD=[select * from authors where state=?lcState]
lcState = 'TX'
IF NOT loPubs.CURSORFILL()
   ? 'Error'
ENDIF

loPubs.TABLES='authors'
loPubs.KEYFIELDLIST='au_id'
loPubs.UPDATABLEFIELDLIST='au_lname,au_fname'
* Primary key must be defined in updatenamelist
loPubs.UPDATENAMELIST = 'au_id authors.au_id, au_lname
authors.au_lname, au_fname authors.au_fname'

loPubs.SAVEASCLASS('ca07','cauthors','authorsw - only fname and
lname updatable')
```

The CursorAdapter object can subsequently be opened using the following code. Note the use of the `AUTOPEN` method instead of `CURSORFILL`. The method simply fills the cursor but is the method used by a Form DataEnvironment to open a CursorAdapter.

```
SET CLASSLIB TO ca07
lcState = 'TX'
loAuthors = CREATEOBJECT('cauthors')
loAuthors.AUTOOPEN
loAuthors.BUFFERMODEOVERRIDE= 3
BROWSE
? TABLEUPDATE(.t.,.t.)
```

A validation rule can easily be defined by modifying the `BEFOREUPDATE` event method. The `DODEFAULT` command is issued to continue with the update or not if the current record transgresses the validation rule. The following example prevents BILL being entered as an author's first name:

```
* BEFOREUPDATE
LPARAMETERS cFldState, lForce, nUpdateType, cUpdateInsertCmd,
cDeleteCmd

* Do not allow update if firstname is bill
LOCAL llFailUpdateRule
STORE .F. TO llFailUpdateRule
LOCAL lcFirstName
STORE SPACE(0) TO lcFirstName
LOCAL lcField
STORE SPACE(0) TO lcField

lcFirstName = EVALUATE(THIS.ALIAS+[.au_fname])
IF TYPE('lcFirstName')=='C'
   IF ALLTRIM(UPPER(lcFirstName))= 'BILL'
      llFailUpdateRule = .T.
      WAIT WINDOW 'Sorry. You cannot have BILL as the first name.'
   ENDIF
```

```
ENDIF

IF llFailUpdateRule
   RETURN .F.
ELSE
   RETURN DODEFAULT(cFldState, lForce, nUpdateType,
cUpdateInsertCmd, cDeleteCmd)
ENDIF
```

*The **BREAKONERROR** property defaults to allowing a CursorAdapter class to trap its own errors within an **ERROR** method. Set this property to **FALSE** to allow standard errors to be generated if there is a problem with your code.*

Attaching an Existing Cursor

An existing cursor can be attached to a CursorAdapter object and the properties of the object manipulated as required. The following example shows a read-only cursor created with SQL pass-through attached to a CursorAdapter with the **CURSORATTACH** method and then specified as an updatable cursor.

```
lnHandle = SQLCONNECT('dsnpubs','sa','')
lnExec = SQLEXEC( lnHandle, ;
  [SELECT * FROM authors],'sptAuthors')

loPubs = CREATEOBJECT('cursoradapter')
loPubs.CURSORATTACH('sptAuthors')

loPubs.DATASOURCETYPE ='ODBC'
loPubs.DATASOURCE = lnHandle
lopubs.Tables='authors'
lopubs.KeyFieldList='au_id'
lopubs.UpdatableFieldList='au_lname,au_fname'
* Primary key must be defined in updatenamelist
lopubs.UpdateNameList = 'au_id authors.au_id, au_lname
authors.au_lname, au_fname authors.au_fname'
```

The **CURSORDETACH** method is used to detach a cursor from the CursorAdapter object. The object reference can then go out of scope but the cursor remains in the environment as a standard local cursor. The CursorAdapter properties will need to be specified again if the cursor is subsequently attached to a CursorAdapter object.

Using the CursorAdapter Builder

Building a CursorAdapter with cabuilder.prg

Using CursorAdapter Objects in a Form

7. Data Buffering

Visual FoxPro creates a local cursor when creating a Local or Remote View. These cursors can be defined to automatically update the original data when the record pointer is changed or the cursor is closed. This process is called Data Buffering and may be controlled in various ways.

This section describes data buffering and how it can be used to control server updates and how error messages returned from the server may be processed.

Buffering may be specified at Record or Table level, or not at all. The locking of the source table may be specified as Pessimistic or Optimistic:

- Record level buffering will automatically commit any changes to the record each time the record pointer is moved.
- Table buffering will commit all changes and additions to the table when the table or form is closed.
- Pessimistic locking will lock the record being edited on the “real” table.
- Optimistic locking does not lock the record until changes are committed but double checks at this time to see if another user has changed the record.

Application design aims to minimise the time that records in the tables are locked so as to maximise the throughput of data for a larger number of users. Optimistic record locking is used wherever possible throughout these notes.

Record buffering can be enabled in a variety of ways:

- The `BUFFERMODE` property of the form can be set to specify Optimistic or Pessimistic buffering.
- The `BUFFERMODEOVERRIDE` property can be set on an individual table in the Data Environment to override the Form setting to any combination of Optimistic and Pessimistic locking or Table and Record level buffering.
- The `CURSORSETPROP` function can be used to set the buffer mode on an individual cursor.

Multilocks must be set on to enable buffering. Be careful to set this inside a form if you are using private data sessions.

Specifying Data Buffering

The CURSORSETPROP() function may be used to enable buffering for the current table or view cursor. To set record level buffering on the AuthorsView remote view for example:

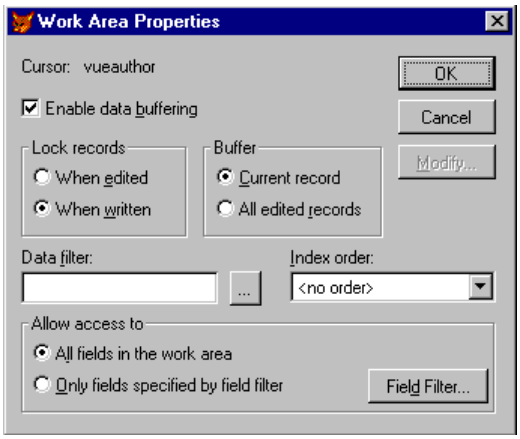
```
USE pubs!authorsview
llOK = CURSORSETPROP( 'buffering', 3 )
```

Record level buffering allows changes to be made to the current record which are sent to the server only when the record pointer is moved or the TABLEUPATE() command is issued.

Table level buffering allows changes to be made to the workstation copy of the data and only sent to the server when the TABLEUPDATE() command is issued.

```
USE pubs!authorsview
llOK = CURSORSETPROP( 'buffering', 5 )
```

Buffering may also be manually set up on a cursor once it has been opened with the View window by pressing the PROPERTIES button after selecting the view cursor workarea.



Setting Optimistic Record Level Buffering in the View Window

Some validation rules prevent APPEND BLANK from functioning correctly and being used on a form button for example. This might occur where empty values were not permitted for a field in the validation criteria but the program was not able to progress to the user input stage until a value has been entered. Table buffering overcomes this limitation.

Both Field and Record Level database rules are checked each time the record pointer is moved. FoxPro Triggers are only run when the data is committed into the database and are not available on Local and Remote Views.

Saving Changes

The TABLEUPDATE function is used to update the table under program control. The following command will write changes for the current record into the database with buffering enabled:

```
TABLEUPDATE()
```

The command can accept parameters to update just the current record (the default), or the whole table where table level buffering is specified. The cursor to be updated can also be specified. All changes on all records of the TITLESVIEW cursor can be sent to the server with the following command:

```
TABLEUPDATE( .T., .F., 'titlesview' )
```

The function returns a True value if the data is committed and a False value if there is a problem with the record level validation rule or any of the triggers. Field level validations are also fired even if the fields have not been entered during an append procedure.

Record level buffering will display any update errors to the user attempting to move the record pointer from a record that cannot be modified because of a server validation problem. This message can be suppressed by using the TABLEUPDATE command to update the record instead of relying on FoxPro.

If an error occurs and any records are not updated on the server, the **TABLEPDATE** function returns a **False** value. The errors returned from the server can be determined with the **AERROR** command.

```
DIMENSION la_Error[ 1 ]
AERROR( la_Error )
DISPLAY MEMORY LIKE la_Error
```

See the section on error trapping later in this chapter for more details on processing errors.

Reverting Changes

The **TABLEREVERT** function is used to abandon changes made to a buffered record or table. The function will return the number of records in the workstation cursor that have been reverted.

```
TABLEREVERT()
```

Normally **TABLEREVERT** will revert the current record if changes have been made. With Table level buffering set and a parameter passed to the function, all the changes made on the required cursor can be reverted:

```
TABLEREVERT( .T., 'titlesview' )
```

*Changed records cannot be reverted after a successful **TABLEUPDATE**.*

Determining Updates

The update status of each field can be determined with the GETFLDSTATE(fieldname) function that returns the status of each field in the current record of a buffered cursor:

- 1 - No Changes.
- 2 - Field Updated.
- 3 - Field Added in a new record but not modified.
- 4 - Field Modified in a new record.

The **GETFLDSTATE(-1)** function is used to return the update status of the current record where buffering has been enabled. The first character of the returned value indicates the delete status of the current record and the remaining characters indicate individual field status.

If the returned string contains only the character 1 then no changes have been made to the current record. 2,3, or 4 indicates that a change has been made or the record added or deleted. The following expression will check for changes to the current record in the current buffered cursor:

```
LOCAL m.lc_getfld
m.lc_getfld = GETFLDSTATE(-1)
IF '2' $ m.lc_getfld ;
   OR '3' $ m.lc_getfld ;
   OR '4' $ m.lc_getfld
   * Changes have been made
   ...
```

```
ENDIF
```

The SETFLDSTATE() function can be used to alter these settings.

Only one record at a time is updated with Record Level Buffering. Several records are changed on the local workstation cursor with table level buffering and in some cases it is useful to determine which records have been changed before updating.

The `GETNEXTMODIFIED` function returns the next record number of the cursor that has been modified on the workstation and not updated onto the server. The function requires a parameter to indicate the record number to search from:

`? GETNEXTMODIFIED(0)` will return the first record that has been modified. If record 5 is returned, the record number of the next modified record is obtained with the following command:

```
? GETNEXTMODIFIED( 5 )
```

Records appended with table buffering have negative record numbers.

Additional functions exist to help resolve update difficulties. The record pointer must be positioned on the required record for these functions to work.

The current value in the local workstation cursor is determined by using the alias and field name in the normal way. For the STORE_ID field in the SALESVIEW cursor for example:

```
m.lc_Local = salesview.store_id
```

The original value determined when the snapshot of the original data was copied onto the workstation is determined by a function called OLDVAL().

```
m.lc_Snapshot = OLDVAL( 'store_id' )
```

The current value on the server will be the same as the OLDVAL() unless the server data has been changed by another user since the snapshot of the cursor data was taken. It can be determined with the CURVAL() function:

```
m.lc_Server = CURVAL('store_id')
```

These three functions may be used in combination to check which fields have been changed by the current user or another user and then use application logic to set values that are acceptable to the server.

Error Handling

Many data validation and business rules can be implemented on the server. The server will return an error to the application program if it is unable to process a transaction. The ODBC standard requires that the front end programming language accepts multiple errors from the server.

The `AERROR` function will capture these errors from the server and place them into the specified array. This is usually required when a TABLEUPDATE() function returns a False value. The following example traps for an error when the current record of the Salesview cursor is updated onto the server:

```
m.lc_cursor = 'salesview'
IF NOT USED(m.lc_cursor)
```

```
   RETURN .F.
ENDIF

m.lc_getfld = GETFLDSTATE(-1, m.lc_cursor)
IF '2' $ m.lc_getfld ;
   OR '3' $ m.lc_getfld ;
   OR '4' $ m.lc_getfld
   * Changes have been made to current record
   IF NOT TABLEUPDATE( .F., .F., m.lc_cursor )
      IF AERROR( la_error ) > 0
         * Errors in updating server
         …
      ENDIF
   ENDIF
ENDIF
```

If an error occurs, the corresponding records will not have been written to the database. You may use the **GETFLDSTATE**, **GETNEXTMODIFIED**, and other options to change the local data and resubmit using a **TABLEUPDATE**. Alternatively, use **TABLEREVERT** to cancel the change for a single record or for all changed records.

The array created by **AERROR** contains the FoxPro error number as well as the server generated error numbers and messages. The server error may need to be parsed in order to be processed correctly.

One common error with optimistic data buffering is that another user will have changed the same record from another workstation. The **TABLEUPDATE** will fail and will return error 1585 into the error array.

```
   * Changes have been made to current record
   IF NOT TABLEUPDATE( .F., .F., m.lc_cursor )
      * Error(s) returned from server
      IF AERROR( la_error ) > 0
         FOR m.ln_error = 1 TO ALEN( la_error,1)
            IF la_error( m.ln_error,1) = 1585
               * Another user has edited the current record
               …
            ENDIF
         ENDFOR
      ENDIF
   ENDIF
```

The values that have been changed by the other user may be checked with the **GETFLDSTATE**, **CURVAL**, and **OLDVAL** functions and the update forced on the server with the **FORCE** parameter of the **TABLEUPDATE** command after setting any acceptable changes from the server record into the current record:

```
IF TABLEUPDATE( .F., .T., m.lc_cursor )
   * Other users changes have been overwritten
ELSE
   * There are still some errors in the data
   …
ENDIF
```

*The Update Criteria of remote views may be set up as **KEY AND MODIFIED FIELDS** so the optimistic locking will fail only if users have attempted to update the same field on the same record.*

*A series of dependent transactions should employ manual transactions on the cursor and use **SQLCOMMIT** and **SQLROLLBACK** to ensure that all the transactions in the sequence are rolled back on the server if there is an error.*

Commit and Rollback

Data Buffering is a very effective way of controlling updates onto the server. Some sequences of transactions however are very sensitive and full commit and rollback functionality may be required.

Visual FoxPro implements server side transaction processing at the level of the connection. The Transactions property of the Connection is set to Manual and the **SQLCOMMIT**and **SQLROLLBACK** functions become operational.

Commit and Rollback work on top of data buffering so both may be used concurrently with the commit and rollback wrapping around the data buffering.

The Connection handle of the current cursor representing a remote view can be determined easily and the transactions set to manual. Commit and rollback is now in operation for the cursor:

```
m.ln_Connect = CURSORGETPROP( 'connecthandle' )
m.ln_Transactions = SQLSETPROP( m.ln_Connect, ;
   'transactions', 2)
IF NOT m.ln_Transactions == 1
   * Error
   …
ENDIF
```

Similar code is required to set commit and rollback on any connection for use with pass through queries.

Data processing takes place in the normal way with **TABLEUPDATE** and **TABLEREVERT**. The changes may be committed into the server with the following command:

```
IF SQLCOMMIT( m.ln_connect ) == 1
   * Committed OK
ELSE
   * Error
ENDIF
```

Rollback is achieved with the **SQLROLLBACK** command and should be used in conjunction with **TABLEREVERT** to refresh the table although these may become out of synchronisation with the table and a **REQUERY** may be required.

```
IF SQLROLLBACX( m.ln_connect ) == 1
   * Rollback OK
   = REQUERY( 'authorscontractview' )
ELSE
   * Error
ENDIF
```

8. Form and Data Environment Properties

A form is created for a remote view in exactly the same manner as a standard form except that the Views radio button must be selected in order to select the view when adding a table into the Data Environment.

Adding a View into the Data Environment

The table buffering on the view is automatically set to optimistic row buffering so that the data is saved onto the server automatically each time the record pointer is moved.

- The **BUFFERMODE** Form property may be used to specify OPTIMISTIC buffering which will default to Table Level buffering for each table.
- The **BUFFERMODEOVERRIDE** property may be set in each cursor in the data environment to specify specific buffering mode for each type.

  ***SET MULTILOCKS ON** is required for table level data buffering.*
  This should be set into forms with Private Data sessions.

There are no differences between using a remote view and using a local view or indeed a native FoxPro table. The table buffering commands should be used to control the updates onto the server however and account needs to be taken of errors being returned from the server.

Add and delete functionality can be implemented once record buffering is employed on a form. Appending records, in particular, can be problematical when using **APPEND BLANK** combined with unique indexes defined in the database container unless record buffering is employed.

9. Optimising Views and CursorAdapters

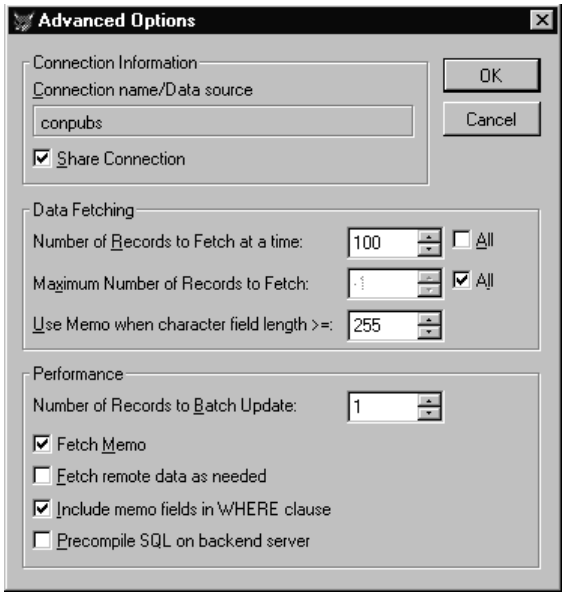
There are many considerations when optimising Visual FoxPro for client-server operation. The art to Client-Server optimisation is to ensure that the minimum amount of information is sent to the workstation with the minimum number of transactions. FoxPro has many features designed to help optimise the transactions.

Some of the optimisations result from common sense or good database design whereas others involve detailed setup of the FoxPro components used in client-server computing or use of server functionality to perform processing on the server.

Advanced Options

Remote Views are usually defined with the View Designer or may be defined with the `CREATE VIEW` command. Remote Views are essentially SQL SELECT statements and care is required to ensure that the indexes and other performance features are set up correctly for maximum performance on the server. If the server is taking too much time to return the query the workstation cannot speed up.

The Advanced Options for a View can be used to specify options that affect performance at the Workstation but do not alter any server side characteristics.



Remote View Advanced Options

The number or records to fetch at a time (`FETCHSIZE`) allows FoxPro to perform a progressive fetch for an asynchronous query. Control returns to the application after the specified number of records have been received from the server and the remainder are received in the background. This value might be set lower when running an application over a modem line.

> *The Connection should be asynchronous for the number of records to fetch at one time to apply. A synchronous query will retrieve all the records at once before returning control to the application.*

The maximum number of records (MAXRECORDS) will prevent the Workstation from receiving more records than specified. Note here that some servers will carry on processing regardless but the workstation process will stop after receiving the number of records specified.

The Fetch Remote Data As Needed option (FETCHASNEEDED) modifies the behaviour of the progressive fetch so that processing does not automatically continue in the background and records are only fetched from the server when requested by the application.

Too many memo fields can clog up network traffic and the Fetch Memo option (FETCHMEMO) allows memo fields to be brought over from the server only when the required field is active in the View. The character length for the memo fetching to operate can also be determined.

Some remote views, especially parameterised views, may be executed repeatedly in an application. The server will interpret the SQL statement passed by the remote view each time the view is requeried. The Precompile SQL on Backend Server option (PREPARED) will allow the SQL statement to be compiled on the server and run faster on subsequent requeries. Not all Servers will support precompilation however.

Remote View Properties

The properties defined for a Remote View may also be defined with the DBSETPROP command as an alternative to the Advanced Options window in the Database Designer.

The Advanced Options correspond to the following database container properties for a view:

- BatchUpdateCount
- CompareMemo
- FetchAsNeeded
- FetchMemo
- FetchSize
- MaxRecords
- Prepared
- ShareConnection
- UseMemoSize

The PREPARED property is important especially for parameterised views that are called repeatedly. SQL Server allows SQL statements to be compiled on the server so that performance is faster on subsequent re-use. Performance on subsequent requeries of a parameterised view will increase if a parameterised view is prepared on the server the first time it is executed.

```
DBSETPROP( 'authorscontractview', 'view', 'prepared', .T. )
```

In addition, the BATCHUPDATECOUNT can be used to automatically batch a number of transactions before sending them to the server. In a high transaction environment, this allows several updates to be processed with a single transaction over the network transaction and can help reduce network traffic.

The transactions sent to the server can be controlled programmatically by setting table level buffering and using TABLEUPDATE() to update the server as required.

There are other properties of a view and of fields in a view which can be useful in determining the structure of the remote view. These are documented in the help under the **DBSETPROP** topic.

***CURSORSETPROP** can be used to alter some of these properties after the view has been opened.*

Recommended Settings

The following recommendations for initial remote view settings should cover most situations and ensure adequate performance:

- SendUpdates should be true.
- Shared Connection.
- Update key and modified fields.
- Asynchronous Queries under careful program control (remember to **SQLCANCEL** before running a subsequent query) or synchronous with careful use of the timeouts to protect against runaway queries.
- 100 Records fetched at a time (10 for dial up access).
- FetchMemo to retrieve memos only when required.
- FetchAsNeeded.
- Do not CompareMemo on update.

10. SQL pass-through

Pass though queries bypass FoxPro functionality and execute a query or command directly on the server. The results are returned from the server into a read-only local workstation cursor.

SQL pass through queries can exploit functionality built into the ODBC driver to add features such as outer joins or date manipulations to the query. These ODBC extensions are interpreted by the driver so that the same syntax can be applied across multiple database servers.

SQL pass though queries can also bypass the ODBC driver to perform operations on the server in the native server syntax. This can be used to optimise queries by using specific server syntax and also to run housekeeping and other functions that need not necessarily return a results set.

> *Cursors created with a SQL pass through command do not automatically update on the server*

Pass through queries often use a combination of commands:

- `SQLCONNECT` is used to create a connection to the server using either an ODBC datasource or more often a connection from the current database container.
- `SQLSTRINGCONNECT` is used with a complete connection string to avoid having to define a datasource on the workstation.
- `SQLEXEC` will execute a query directly on the server.
- `SQLCANCEL` will cancel a query on the server.
- `SQLMORESULTS` is used if more than one results set is returned from the server in non-batch mode.
- `SQLPREPARE` can be used to prepare (compile) statements on the server for subsequent re-execution.
- `SQLDISCONNECT` is used to close the connection.

The `SQLCONNECT` function is passed the name of an ODBC datasource or connection in the current database container and will return a positive number representing the connection handle if the connection with the server is made successfully.

```
m.ln_connect =  SQLCONNECT( 'conPubs' )
IF m.ln_connect < 1
   * Error
   ….
ENDIF
```

The connection handle returned from SQLCONNECT is now used to execute the pass through query onto the server. This example executes a housekeeping function on the server that does not return a results set:

```
m.ln_exec = SQLEXEC( 2, 'UPDATE STATISTICS authors' )
```

A negative value returned from SQLEXEC indicates an error which requires the AERROR() function to provide further information. A value of one indicates that the query has completed and a value of zero indicates that the query is still processing.

The following pass through query will return a read only cursor called CURSALES:

```
m.ln_exec = SQLEXEC( 2, 'SELECT * FROM sales' , 'cursales' )
```

The SQLEXEC can be used to create more than one results set simultaneously by separating the queries with a semi-colon. The following example will return two results sets from the server, one containing a read only view of the sales table, and the second a read only view of the authors table:

```
? SQLEXEC( 1, 'SELECT * FROM sales;SELECT * FROM authors' ,
'cursales' )
```

In this case the SQLEXEC returns a 2 indicating that two results sets have been created on the workstation (CURSALES and CURSALES1).

Asynchronous Mode

SQL Pass Through queries may use an asynchronous connection to pass results onto the application before the query has finished processing.

The Asynchronous property of the connection can be set using the connection designer or programmatically with the SQLSETPROP() command:

```
? SQLSETPROP( 1, 'asynchronous', .T. )
```

The SQLEXEC command may now be issued and will return zero while still executing.

```
m.ln_exec = SQLEXEC( 1, 'SELECT * FROM sales', 'cursales' )
IF m.ln_exec < 0
   * Error
   …
ELSE
   IF USED('cursales')
      BROWSE
   ENDIF
ENDIF
```

The command is issued again periodically to check if processing has finished when one is returned.

```
IF SQLEXEC( 1, '' ) = 1
   * Finished
   …
ENDIF
```

The query can be cancelled when enough records have been retrieved with the `SQLCANCEL` command:

```
? SQLCANCEL( 1 )
```

Batch Mode

Batch mode can be set on the Connection so that multiple queries are not returned simultaneously but one after another. This allows the first table to be processed before receiving the second.

The Batch property of the connection can be set using the connection designer or programmatically with the SQLSETPROP() command:

```
? SQLSETPROP( 2, 'batch', .F. )
```

The non-batch pass through query is now executed and only the first results set is returned.

```
? SQLEXEC( 1, 'SELECT * FROM sales;SELECT * FROM authors' ,
'cursales' )
```

The second results set is waiting to be sent to the server and must be retrieved with the SQLMORERESULTS function which used the connection handle as a parameter and will return 2 because the second results set is created. This function is called until the final set is retrieved.

```
? SQLMORERESULTS( 1 )
```

The SQLCANCEL command is used to cancel a query whenever required:

```
? SQLCANCEL( 1 )
```

SQL Pass Through and Data Buffering

SQL Pass Through queries execute a statement directly on the server and often return a cursort. This cursor might be the product of a system procedure or a standard SQL SELECT statement. Whatever the nature of the cursor it is read-only.

Sometimes it is useful to make a results set from a pass through query updateable. A remote view may not be possible if some functionality specific to the server is included in the SELECT statement or if a more flexible approach than parameterised queries is required.

A pass through query is used to create a results set which is represented as a cursor in FoxPro. The cursor has many properties that may be set with the CURSORSETPROP command. A remote view automatically sets these properties to allow the cursor to be updateable. With a results cursor from a pass through query these properties must be set programmatically.

The simple example where a remote view was created to show authors with contracts can be implemented as an updateable pass through query by following a fairly complex procedure.

First make a connection handle to the required datasource or connection. Various connection properties can be set to create asynchronous queries and so forth. These examples use the default settings.

```
m.ln_Connect = SQLCONNECT( 'odbpubs' )
IF m.ln_Connect < 0
   * Error
   …
ENDIF
```

Now create the results set as required using the SQL pass though commands. In this instance the value for the Contract field will not be a logical because it is stored as a Bit field on the server. The SELECT statement is presented directly to the server and will not automatically translate datatypes (unless a parameter is specified):

```
m.ln_exec = SQLEXEC( m.ln_connect, ;
   'SELECT * FROM authors WHERE contract = 1', ;
   'curauthorscontract' )
IF m.ln_exec < 0
   * Error
   …
ENDIF
```

The cursor has not been created but is read-only. Several cursor properties need to be set in order for the cursor to be updateable:

- SENDUPDATES must be set True for updates to be passed back to the server.
- TABLES must specify the server tables used in the view.

- **KEYFIELDLIST** must contain a comma separated list of key fields.
- **UPDATENAMES** should contain a comma separated list with the FoxPro field name then a space and the full serve alias table and field name. This must include the primary key fields.
- **UPDATABLEFIELDLIST** must contain a comma separated list of updateable fields.

A useful technique to determine these settings is to use the CREATE SQL VIEW command to create a similar remote view on the database and then use DBGETPROP() to determine the required property settings.

The following code, executed at the command line, will set all the required properties on the cursor to allow updates to be made to the forename and surname fields:

```
? SQLEXEC(1,'select * from authors','autest')
SELECT autest

? CURSORSETPROP( 'sendupdates', .T. )
? CURSORSETPROP('tables','authors')
? CURSORSETPROP('updatename', 'au_id authors.au_id,au_lname
authors.au_lname,au_fname authors.au_fname' )
? CURSORSETPROP( 'updatablefieldlist','au_lname,au_fname')
? CURSORSETPROP( 'keyfieldlist', 'au_id' )
```

*There are many other properties that can be set against both the cursor and the connection using **CURSORSETPROP** and **SQLSETPROP** respectively. Refer to on-line help for more details.*

Preparing SQL Statements

SQLPREPARE can be used to prepare SQL statements on the server. The statement is prepared on the server and will execute faster when subsequently executed and is therefore of most benefit for complex select statements with parameters that are run repetitively:

Complex select statements may be better implemented as a stored procedure on the server as the server may be able to store and monitor an optimised plan for executing the SQL efficiently.

A connection is used for the prepared statement and the statement must be prepared again if another pass through is executed on the connection. The SQLPREPARE() command is then used to pass the statement through to the server. Any parameters must have been previously defined.

```
m.ln_Connect = SQLCONNECT( 'conpubs' )
m.lc_surname = SPACE(0)
m.ln_prepare = SQLPREPARE( m.ln_connect, 'SELECT * FROM authors
WHERE au_lname LIKE ?lc_surname','cursurname' )
```

The statement is now prepared and the SQLEXEC can be used to run the query. The SQLEXEC behaves normally and may need to be run several times until a 1 is returned depending on the size of the query:

```
m.lc_surname = '[a-d]%'
? SQLEXEC( m.ln_connect )
```

Subsequent changes to the parameter and execution will use the prepared statement.

> *The statement needs to be prepared again if the connection is used for another query.*

ODBC Extensions

ODBC Drivers have various extensions to the SQL Syntax that allow functions to be implemented transparently without relying on specific functionality of the back end server.

There are various scalar functions, for example, that allow string and date conversion and other formatting to be implemented without relying on the native syntax of the database server. SQL pass through statements that use these extensions will function with any ODBC driver that supports them.

The {} brackets indicate to the ODBC driver that ODBC extensions are in operation and the driver will interpret the function and pass it onto the server.

Date functions are useful when passing date values to the database server:

```
? SQLEXEC(1,[select * from sales where ord_date = {d '1992-06-
15'}])
```

The following example will convert the surname field to upper case and provide a SOUNDEX value for the surname without using server resources:

```
m.ln_sql = SQLCONNECT( 'odbpubs' )
? SQLEXEC(m.ln_sql, 'select au_id, { fn ucase(au_fname)},
au_lname, {fn soundex(au_fname)} as soundex from authors',
'curauth01')
```

Note that in the above example, SOUNDEX is also a SQL Server function and would work without the ODBC {} brackets. The UCASE function however is an ODBC extension as the SQL Server syntax is UPPER.

Another example uses date functions to determine the year and week of employment using the HIRE_DATE datetime field in the EMPLOYEE table:

```
? SQLEXEC( 1, 'SELECT *, {fn year(ord_date)} year, {fn
week(ord_date)} week, {fn dayofweek(ord_date)} dow FROM sales' )
```

Date conversion functions are amongst the most useful of the ODBC scalar functions as each server seems to have its own date format. The following pass through query uses the ODBC CONVERT function to query all employees hired on or since Christmas 1992:

```
? SQLEXEC( m.ln_sql, [select * from employee where hire_date >= {
fn CONVERT ('1992-12-25', SQL_TIMESTAMP)}] )
```

Outer Joins may also be specified with ODBC extensions as follows:

```
? SQLEXEC(1, [SELECT authors.*, titleauthor.* FROM {oj authors
LEFT OUTER JOIN titleauthor ON authors.au_id = titleauthor.au_id}]
)
```

Stored Procedures

One of the best ways to improve application performance is to implement server side functionality using stored procedures. These can be called easily using the **SQLEXEC** command and often return results sets.

The PUBS database has a stored procedure which can be called with a **SQLEXEC** to return a results set of authors with a particular royalty rate:

```
m.ln_handle SQLCONNECT('conpubs')
m.ln_exec = SQLEXEC(m.ln_handle, [exec byroyalty 40])
```

The stored procedure may also be called with a parameter:

```
m.ln_royalty = 40= SQLEXEC(1, [exec byroyalty ?m.ln_royalty])
```

Some stored procedures may return values by reference. The following example requires three integer parameters. The first parameter is passed by reference and receives the result of adding the remaining two parameter values together.

The parameter may be passed by reference, as an OUTPUT parameter, using the ODBC extensions for calling a stored procedure as shown below:

```
m.ln_result = 0
m.ln_exec = SQLExec(1, "{CALL stpmathtutor (?@m.ln_result, 2, 4
)}" )
```

Be careful when using output parameters with asynchronous queries as the parameter may not be updated until the final records are retrieved.

Stored procedures often return an integer value to indicate the success status. I have yet to find a method to obtain this value using ODBC. You need to use ADO to get the result back from a stored procedure

11. ADO and XML

12. Offline Views

Offline Views allow a remote view to be created as a local cursor in the usual way and the server connection to be severed and reconnected at a later time.

Offline views are useful for creating local copies of non-volatile data which can record changes for updating onto the server at a later date. Offline views are particularly useful for distributing data onto remote computers. A salesman can take data away on a laptop and update any changes back onto the server on their return.

The **CREATEOFFLINE** syntax is used to create the Offline view. The command simply defines a view in the currently open database container and a filename for the local cursor:

```
OPEN DATABASE dbcpubs
? CREATEOFFLINE( 'vueauthor','c:\temp\authors')
```

Once created the Offline view can be used with the ADMIN keyword of the use command which will refer to the local copy of the data and not require access to the server:

```
SET MULTILOCKS ON
USE c:\temp\authors ADMIN
```

The connection to the server can be re-established using the ONLINE keyword of the **USE** command. The changes can now be updated onto the server using the standard data buffering techniques:

```
USE c:\temp\authors ADMIN
IF NOT TABLEUPDATE( .T. )
   * Errors updating offline view
   …
ENDIF
```

The Offline View may be dropped without updating the server with the **DROPOFFLINE** command:

DROPOFFLINE('vueauthor').

13. Client-Server Application Design

Various bottlenecks need to be addressed when designing an application that is optimised for client-server operation. This section discusses the broad issues and indicates how the application should be designed for maximum performance.

Performance Bottlenecks

Client-server designs using front-end languages designed for a graphical user interface suffer from the user expectations of flexible and on-demand access to data. User expectations of an interface similar to a local spreadsheet together with system requirements to run up to hundreds of users simultaneously over the network require that considerable care must be taken over performance considerations.

The final user interface needs to be a trade off between the performance issues and the usability that is built into the application. It is possible to build mainframe style applications that support hundreds of users at the cost of usability or very usable database applications that support only ten users.

Some of the performance issues are detailed below:

- Network traffic is the greatest bottleneck requiring that a minimum amount of data is transferred between client and server with a minimum number of transactions in high volume transaction processing systems.
- Many validations can be performed locally within the application to prevent the loop of updates being attempted on the server, error messages passed back, changes made locally, and then reissued onto the server.
- Server resources can quickly be consumed with certain types of application design and front end programming tools. For example, each user requires at least one connection on the server which requires some server memory. Some applications may use several server connections for each user thus overloading the server.
- Some database servers can perform a considerable amount of processing on the server if the appropriate triggers and stored procedures are set up. This functionality is server specific and will not allow the application to work with all servers but will significantly reduce traffic for many types of transaction.
- Pass through queries can exploit high performance features of a particular server to take the load off the client and onto the server.

Parameterised Views

Parameterised Views are of great importance in reducing the amount of data retrieved from the server at any one time. Care should be taken that appropriate indexes are set up so that the server can optimise the remote view and a small amount of data is always specified.

Only the required fields (and primary keys) need be specified in a remote view. Be careful with memo and text fields and use the view parameters to reduce the network traffic for these large fields.

Some servers, including SQL Server, place read locks onto records when they are read as part of a transaction that may be followed by updates. The number of records could be

kept to a minimum here or the transaction terminated to release the locks and reduce server overheads in a high volume situation.

Preparing the view on the server will improve performance for a view that is repeatedly queried many times for a single user.

Local Validations

Performing some validations programmatically at the workstation instead of relying on the server validations will reduce the number of transactions passing over the network due to incorrect user entry.

Field and Record level validation can be implemented using the local data dictionary or application logic on the workstation. These validations should not require any checks with data from the server so no network traffic is generated.

> *Field level validation is specified in the View Designer whilst Record level validation rules may be specified for a remote view using the DBSETPROP() command to set the RuleExpression property for the View.*

Validations that require checks on server data are better performed on the server. This applies particularly to referential integrity checks and also to complex record level validations that can be performed with a Trigger.

In some cases a local copy of the server data should be made to reduce network traffic. For example, if a product list is relatively static, a local copy of the table could be made at the beginning of processing so that all validation is performed locally.

> *An Offline View can be used to create a hybrid system where near static data is updateable by the user who will only see local changes to the tables until the next time the view is created.*

Validations that are checked locally will also need to be implemented on the server if other applications are accessing server data. The cost of this approach is that changes need to be made to the server and client data dictionaries each time a change is made to the business rules affecting the application. The workstation application will also need to be upgraded each time a change is made.

Transactions

The frequency and size of transactions between the client and server can be controlled by the design of the application. Try to retrieve a small amount of data to the user and update in one transaction wherever possible to reduce network traffic.

Encourage the user to select a small number of records by providing a usable selection window that allows pinpointing of a small number of records before retrieving the data from the server.

Many data entry forms may search for information on a parent table and only show child related information if required. For example, an accounts system might allow the user to search for an invoice and then display the line items. Two views could be used in this case so that the line items are retrieved only if required for display to the user.

Some applications have several users constantly entering data. This data may not be required immediately on the server and the number of transactions may be reduced by updating the server after ten records have been entered instead of each time one record is entered.

Similarly changes to multiple records that form part of a transaction could be stored on the workstation and issued to the server in one transaction using table buffering rather that issuing several transactions followed by a commit or rollback.

Stored Procedures

Stored Procedures are passed directly to the server and can use whatever performance enhancement functions are supported by the server. SQL Server applications might use a stored procedure to return required information for example.

One example that creates huge gains in performance involves sequences of transactions that are common in accounting applications where a transaction is made in one account and several other corresponding transactions must be made in other ledgers for double entry book keeping.

Many applications would issue a begin transaction on the server, issue the first transactions followed by a transaction for the double entry in the other tables, and then commit. All of these transactions could be performed in a single stored procedure by passing the amount and the ledger names to the procedure. Only one transaction is passed to the server and the server performs all the subsequent transactions locally. In a high volume scenario this improves performance considerably.

Other examples might involve the creation of a temporary table on the server and running various procedures before returning the final results set to the workstation.

SQL Server can also call external stored procedures to compiled DLL programs on the server to integrate with mail, Internet, or other infrastructure systems in the organisation. This requires only one connection to the resource for all database users and may reduce overall network load.

SQL Pass Through

The server may have extensions to standard SQL that allow better server performance through the use of proprietary language features. If performance is critical, these may improve performance whilst tying the application to the particular database platform.

There are some ODBC extensions that may help in creating pass through queries that operate on several servers. These are particularly useful for date arithmetic and left outer joins.

One clever trick to improve performance might be to create long strings of commands that can be sent to the database server as a single command. A series of one hundred INSERT statements separated with semi-colons and sent to the server as a SQL pass through statement might be faster than appending records with a remote view.

Connections

Some applications will use more than one connection to retrieve data from the server. A grid might retrieve data using one connection, a form using another, and a combo control on a form yet another.

Care should be taken to reduce the connections per user (by using shared connections) as these consume valuable resources on the server. Remember that SQL pass through statements can use the same connection handle as remote views.

Microsoft Transaction Server

The 3-tier model of programming promoted by Microsoft can work very well in many situations. In this case an application server runs business components as small objects which can easily be instantiated on separate client workstations. All communication with the database server passes through this middle layer of software components.

Microsoft Transaction Server handles all the software configuration issues and the software behaves as if it is running on the local workstation. This allows Visual Basic or Microsoft Office applications access to complex database functions written in FoxPro without requiring FoxPro to be installed locally. It is also the recommended method for implementing web pages with ASP.

However, it is likely that a native FoxPro application will perform better by accessing the ODBC driver directly from the workstation than it would by using a three-tier software architecture to access a remote automation server.

14. Database Maintenance

Upsizing a FoxPro Database

FoxPro has an upsizing wizard which does a good job of converting a FoxPro database onto SQL Server. The wizard attempts to move as much of the database information as possible onto the server including indexes, defaults, rules and relationships as well as the data itself. A series of reports are generated along with the SQL so that the process can be repeated and tweaked until successful.

It is left to the programmer to create the (parameterised) remote views in a new database container to begin the process of converting the underlying application. In theory the application will work by replacing the tables in the database container with remote views of the same name and using a shared synchronous connection with table buffering.

In practice however, as a minimum you will need to look at the following:

- Use shared connections for the remote view.
- Add parameters to the remote views to minimise the records returned. You may need to create several views on the same table for different access paths. Keep the fields to a minimum and take care with memo fields.
- Review the indexes on the SQL tables and remember that SQL tends to use a single index for selection optimisation. Try to create 'covered' indexes that contain all the fields used to select data.
- Be very careful with clustered indexes. The should represent the most frequently used ORDER BY criterion rather than the primary key and try to avoid using a sequence that is the same as the order of insertion (e.g. Invoice Number) for high transaction volume tables.
- Use table buffering for greater control over the timing of updates to the server.
- Define stored procedures on the database server for regularly called routines that update several records.

Visual FoxPro Upsizing Wizard

The Upsizing wizard is located in the Tools-Wizards-Upsizing menu option and is used to automatically upsize the tables defined in an existing database container.

An ODBC Driver should be defined on the workstation to access the SQL Database before the upsizing wizard is used. It is good practice to plan and create the SQL Database using the SQL Server utilities although the upsizing wizard will let you create a new database. A SQL Database will usually be 1.5 times the size of the FoxPro database files.

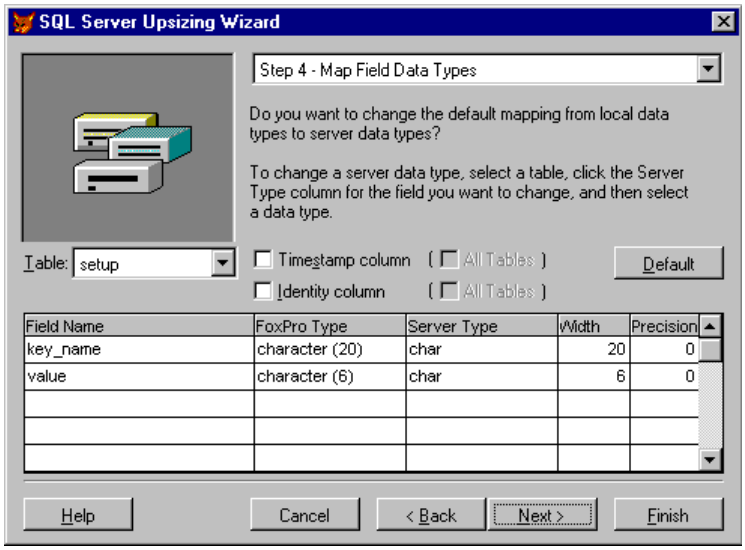
The first few pages of the Upsizing wizard prompt for the following criteria:

- Database Container to be upsized.
- An ODBC DataSource that has already been setup on the workstation to point to the SQL Server. A valid login password may be required at this point.
- The selection of Tables to be upsized.

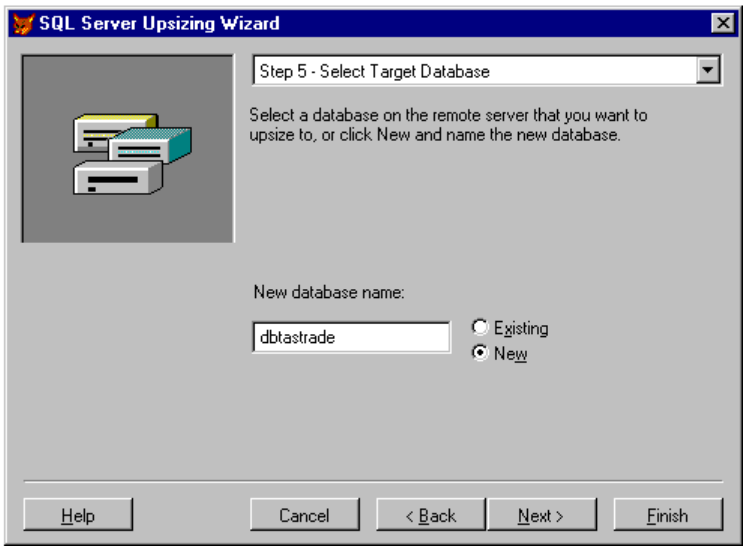
It is then possible to view all the fields for each table and alter the default field type specified for SQL Server. This is useful for using SQL Server User Defined Datatypes or to conform with standards that existing front-end application software may require.

Adding a Timestamp field will enable the Key and Timestamp updating option to be set for the remote view which improves performance when checking to see if other users have made changes to the record being edited..

The Identity column allows for automatic incrementing primary keys which will create new primary key values without the need for any application level logic.



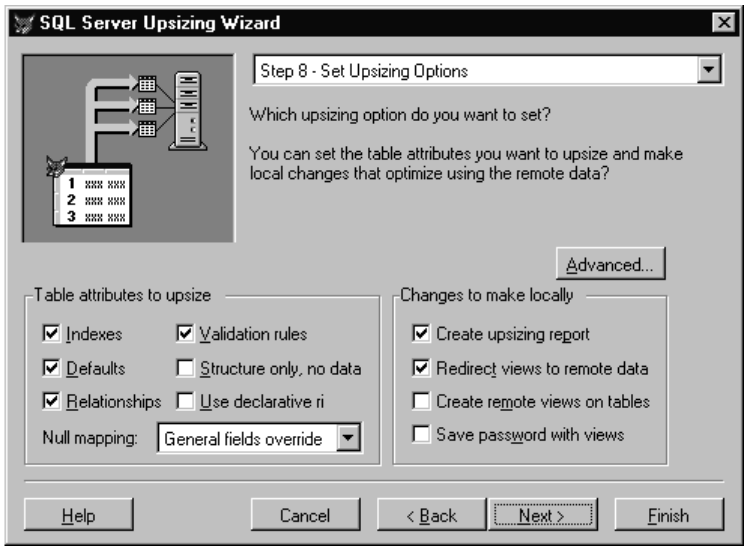
Mapping Field Types with the Upsizing Wizard



Creating a Database using the Upsizing Wizard

The upsizing options allow selection of the required components to update. The advanced options allow specification of clustered indexes but should be used with caution.

> *It is often easier to upsize several times without any data until the process is successful and then append the data into the remote views.*



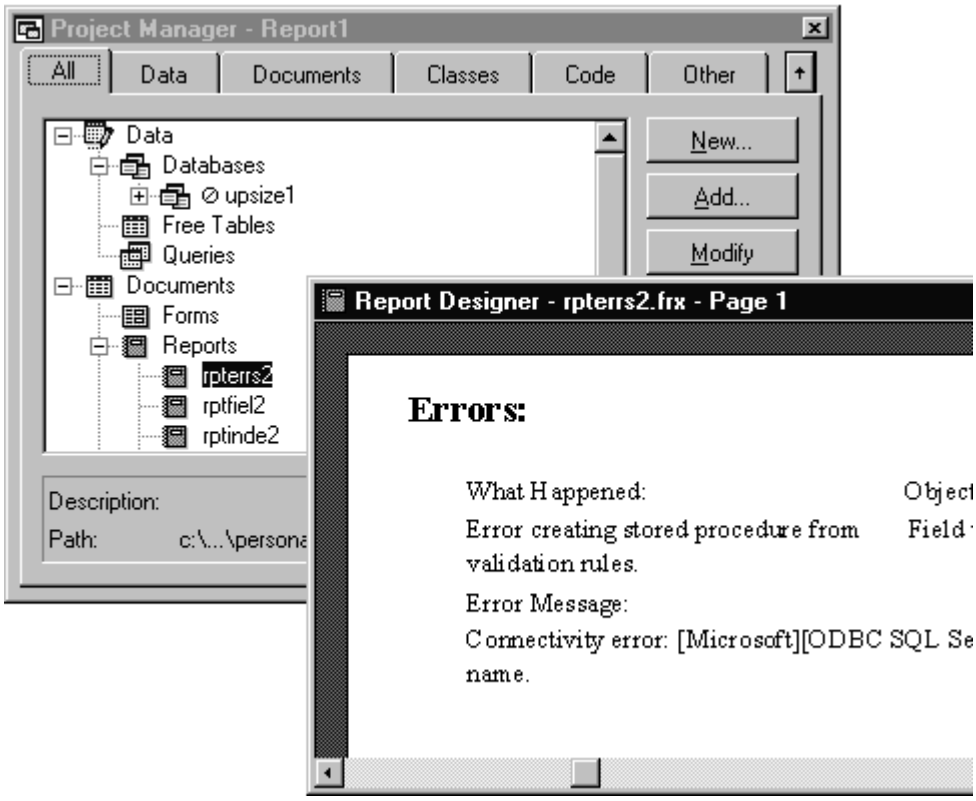
Upsizing Options

The final window allows for the generated SQL to be saved and the upsizing process to begin. The process may take considerable time if there is a lot of data.

A folder called UPSIZE is created and contains a project containing various tables and reports run to view any errors occurring during the upload. The project tables contain details of the upsizing process. Notice the one record table called SQL_UW that contains the SQL used to generate the schema on the database server.

> *A useful approach is to upsize the FoxPro database without any data and fix any errors before creating views and importing the data by opening the view and using the APPEND FROM command.*

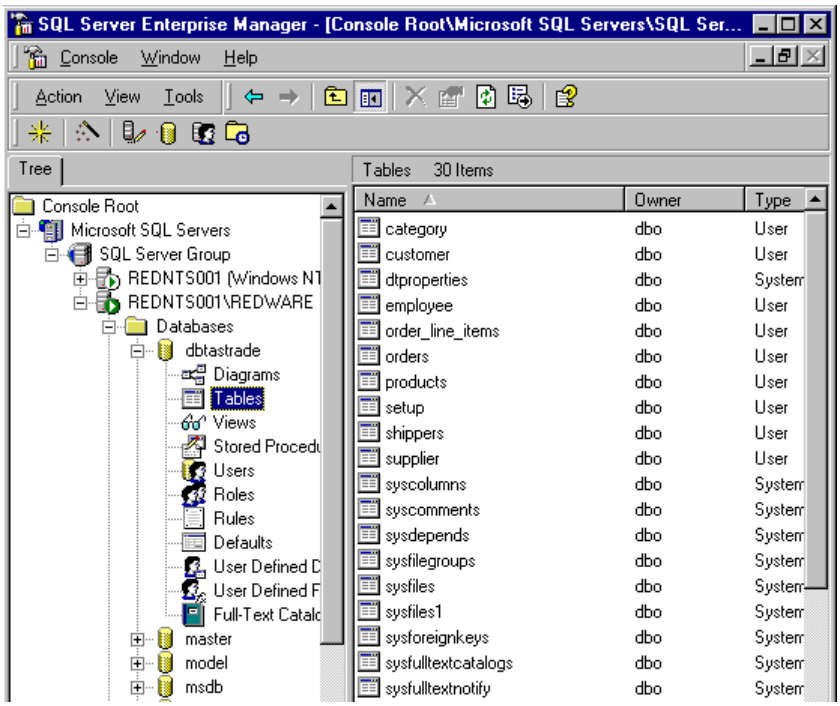
Some errors occur because features defined in the FoxPro Database Container are not available in SQL Server. A default value for a field may be determined by a User Defined Function in the local database container. This code may not be upsized onto SQL Server and an error occurs. Another common error is that a fieldname clashes with a reserved word in SQL Server.



Upsizing Error Report

Upsizing Considerations

The defaults used by the Upsizing wizard may not be the best to use in practice. Some properties of the FoxPro database container may not move successfully into the server database because of differing functionality, and index design criteria may be different for FoxPro and SQL Server optimisation.



Upsized Table viewed in the SQL Enterprise Manager

Some considerations which may be useful when upsizing a FoxPro database are detailed below:

- Select the correct format for numeric fields to save on disk space.
- Wide text fields with values of different widths may be better stored as VARCHAR fields to save on disk space.
- Take care with memo fields which will automatically assign 2K of disk space for each record even if the memo is empty. Make sure Nulls are allowed for memos if a large proportion are empty.
- Some field defaults and validations may pass successfully over to SQL Server. Anything remotely complex will have to be rewritten as a Transact-SQL Trigger.
- Index design with SQL Server should favour more composite indexes designed around the most common queries rather than the FoxPro design of many independent indexes for each part of the query expression.
- Take care when assigning clustered indexes.
- Remember to update statistics after the data has been loaded, or the query optimiser will not function correctly.
- Referential integrity may be implemented easily in the database server.
- FoxPro stored procedures are not upsized and must be rewritten on the server or implemented as View record level validation.
- SQL Server cannot index on Bit (logical) fields only.

Data Manipulation Language

ODBC Supports a variety of data manipulation commands that conform to basic SQL standards. This allows for the initial definition of a table and the setting of indexes and primary key features.

ODBC 2.0 drivers have extended these definitions and the Microsoft ODBC Driver for SQL Server 2.5 supports a full set of Database Manipulation commands.

Not all ODBC Drivers will support the full set of data manipulation commands.

The Data Manipulation commands may be passed directly to the ODBC Driver using the **SQLEXEC** command. A connection handle is first defined with the **SQLCONNECT** function.

The **SQLCONNECT** command may be entered at the Command Window as follows. The result is printed on the screen and a -1 indicates there is a problem and the connection has not been made. You are prompted to log into SQL Server when the connection is made.

```
? SQLCONNECT( 'odbSales' )
```

The error message returned from the server if a SQL command fails may be viewed with the AERROR() command.

Several Connections may be made from one workstation to the server and the correct handle must be used in the SQLEXEC command. The SQLEXEC Command will also return a -1 in the event of failure and passes the command directly to the ODBC Driver.

An index may be created on a remote data source by sending the CREATE INDEX DML command to the driver with the SQL **EXEC** Command:

```
? SQLEXEC( 2, ;
   'CREATE INDEX quantity ON sorditem (quantity,unitprice)')
```

ODBC Supported Data Manipulation commands include the following SQL Statements with a syntax identical or extremely similar to a standard Visual FoxPro statement:

- **CREATE TABLE**
- **CREATE INDEX**
- **ALTER TABLE**

An easy way to determine the required SQL DML command is to create the required structure in Visual FoxPro and then use the Upsizing Wizard to generate the SQL Script only.

*ODBC does not like the COLUMN keyword in the **ALTER TABLE** Command:*

SQL Pass Through command can be used to pass statements directly to SQL Server to allow SQL Server specific functionality to be implemented.

This could be used to create a view for example:

```
? SQLEXEC( 2, [CREATE VIEW CustView AS SELECT * FROM customer
WHERE company LIKE 'C%'] )
```

CREATE TABLE

```
CREATE TABLE [database.[owner].]table_name
(
   {col_name column_properties [constraint [constraint
[...constraint]]]
   | [[,] constraint]}
      [[,] {next_col_name | next_constraint}...]
)
[ON segment_name]
```

The CREATE TABLE command can be used to create tables using the standard SQL functionality for defining fields but may also be extended to define constraints for the default and simple checks as well as primary and foreign keys and full referential integrity checking.

The following example is used to create the Titles table in the PUBS database and can be found in the INSTPUBS.SQL script. Note that scripts can easily be generated from existing databases to provide a view of the CREATE TABLE syntax.

```
CREATE TABLE titles
(
   title_id        tid
      CONSTRAINT UPKCL_titleidind PRIMARY KEY CLUSTERED,
   title   varchar(80)     NOT NULL,
   type    char(12)        NOT NULL
      DEFAULT ('UNDECIDED'),
   pub_id  char(4) NULL
      REFERENCES publishers(pub_id),
   price   money   NULL,
   advance money   NULL,
   royalty int     NULL,
   ytd_sales       int     NULL,
   notes   varchar(200)    NULL,
   pubdate datetime        NOT NULL
      DEFAULT (GETDATE())
```

ALTER TABLE

```
ALTER TABLE [database.[owner].]table_name
[WITH NOCHECK]
[ADD
   {col_name column_properties [column_constraints]
   | [[,] table_constraint]}
      [, {next_col_name | next_table_constraint}]...]
```

ALTER TABLE is very useful for adding or modifying the definition of existing Tables. It is particularly useful when writing upsizing programs as it performs many of the tasks that can be carried out using the SQL Enterprise Manager in a program.

CREATE INDEX

```
CREATE [UNIQUE] [CLUSTERED | NONCLUSTERED] INDEX index_name
   ON [[database.]owner.]table_name (column_name [,
column_name]...)
[WITH
```

```
   [FILLFACTOR = x]
   [[,] IGNORE_DUP_KEY]
   [[,] {SORTED_DATA | SORTED_DATA_REORG}]
   [[,] {IGNORE_DUP_ROW | ALLOW_DUP_ROW}]]
[ON segment_name]
```

CREATE INDEX can also be used programmatically to create indexes without using the Enterprise Manager.

GENDBC.PRG

FoxPro ships with an excellent utility called GENDBC which creates SQL DML code from the current open database container and places the results in the specified program file. Much of the syntax is consistent with syntax that can be run on the server with a script or by using SQL pass through.

```
OPEN DATA tastrade
DO HOME() + 'tools\gendbc\gendbc' WITH 'c:\temp\fred'
MODIFY COMMAND c:\temp\fred
```

15. Index

·SQLDISCONNECT....46
AERROR....37, 39, 63
ALTER TABLE....64
APPEND BLANK....37, 42
asynchronous....43
Asynchronous Connection....18, 47
BATCHMODE....16
BATCHUPDATECOUNT....44
buffering....7
BUFFERMODE....36, 42
BUFFERMODEOVERRIDE....8, 36, 42
Commit....41
CONNECTBUSY....16
Connection....6, 56
  Batch Mode....47
  Shared....27
  Timeout property....15
Connection Designer....15
CONVERT....50
CREATE CONNECTION....16
CREATE INDEX....63, 64
CREATE SQL VIEW....7, 26
CREATE TABLE....63
CREATEOFFLINE....53
CURSORGETPROP....41
CURSORSETPROP....45, 48
  Buffering....36
  setting defaults....17
CURVAL....39
Data Buffering....36, 48
  Optimistic Locking....36, 40
  Pessimistic Locking....36
Data Manipulation Language....62
DBSETPROP....7, 26, 27, 45, 55
DISPLOGIN....16
DROPOFFLINE....53
Error Handling....39
FETCHASNEEDED....44
FETCHMEMO....44
FETCHSIZE....18, 43
Field
  Upsizing....58
form....8, 42
GENDBC.PRG....65
GETFLDSTATE....38
GETNEXTMODIFIED....39
Join
  Outer....22
KEYFIELDLIST....48
MAXRECORDS....43
MODIFY CONNECTION....14
NODATA....7
NODATAONLOAD....8
ODBC....12, 58
  Connection Pooling....14
  connection string....12
  Data Source Administrator....13
  Performance Tips....14
  Scalar Functions....50
Offline View....55
OLDVAL....39
Outer Joins....50
PACKETSIZE....16
Parameterised View....21, 25, 28
Pass Through SQL....46, 48, 56, 63
Performance....54
PREPARED....29, 44
remote view....6, 26
  update criteria....27
Remote View....43
  prepared property....44
REQUERY....7, 25, 29, 41
Rollback....41
RULEEXPRESSION....19
SENDUPDATES....48
SENDUPDATES....7, 26, 27, 29
SET MULTILOCKS ON....42
SETFLDSTATE....39
SOUNDEX....50
SQLCANCEL....46
SQLCOMMIT....17, 41
SQLCONNECT....46, 48, 63
SQLEXEC....6, 46, 63
SQLMORESULTS....46
SQLPREPARE....46, 49
SQLROLLBACK....17, 41
SQLSETPROP....41, 47
  setting defaults....17
SQLSTRINGCONNECT....6
SQLTABLES....6
Stored Procedure....8, 51, 56
  Make Updateable....48
  parameter....51
  parameter by reference....51
TABLEREVERT....40, 41
TABLES....48
TABLEUPDATE....7, 37, 39, 41, 44
  Force Parameter....40
Transactions....55
TRANSACTIONS....17
UCASE....50
UPDATABLEFIELDLIST....49
UPDATENAMES....48
UPDATETYPE....27
Upsizing Wizard....58
USE....28
  ADMIN....53
  NODATA....28
  NOREQUERY....28
  ONLINE....53
view designer....19
  fields....20
  filter....21
  Group By....23
  join....22
  Order By....22
  Update Criteria....23
WHERETYPE....27