A Walking Tour of RDO

Click & Retrieve Source CODE!

BY WILLIAM R. VAUGHN

Forget ODBC's API. Remote Data Objects give VB4 client/server front ends a safer, simpler data interface.

I was just nodding off in my big blue chair when the phone rang. "Hello?" I said in my this-had-better-be-a-good-reason-to-bug-me-on-Saturday voice. I figured MCI was calling me with another pitch but it turned out to be Fred in trouble again. Fred, one of my daughter's soccer team parents, thinks that because I work at Microsoft I must be a computer guru. I should have told him I was an IRS agent.

Fred's office had just installed SQL Server, but a few problems still loomed over a Monday morning deadline to go online. I said, "You know this is Saturday, right?" He knew.

The database had been built and the data loaded, but the front end had become troublesome. Fred had read my book, and I assumed he had taken some of my advice.

"So, your VBSQL applications aren't working?" I queried.

"Actually, we decided to use the new Remote Data Objects instead." I provided a pregnant pause. Finally he broke the silence. "Ah...does RDO have a problem we should know about? Should we have used VBSQL instead? Or maybe Jet DAO? A consultant I know had argued for that."

"No, no." I replied. "I'm just surprised you used RDO. It's fairly new and not many people have tried it yet. VBSQL would have been the conventional choice. However, you would have had a tougher time coding it. As for DAO, certainly a lot of people still use that, especially if they have a background in dBASE. But you chose the right technology for the long run. RDO suits the size of the system you're working with, especially since you depend so heavily on secure stored procedures. On the other hand, this call tells me that being the pioneers caught you a few arrows."

"You got that," Fred said. "We set up the new server using stored procedures all right, but we haven't been able to get them working properly. Could you come over and take a look?"

"I'll be there in twenty minutes." My spouse shook her head in resignation. She knew I'd be gone for most of the weekend. I pushed the stopwatch button on my watch. Fred was now on the clock, and at my double-time weekend rate.

WRESTLING WITH THE RESULT SET

It turned out that Fred had the problems I've come to expect from sites that migrate from ISAM-based systems to true relational client/server. In many cases, these sites must convert older applications that perform many of the tasks now done entirely by the intelligent back end. Examples include maintaining indexes, finding data subsets, and performing bulk operations on those subsets.

In systems based on ISAM (IBM's nonrelational, hardware-dependent Indexed Sequential Access Method which permits both direct and sequential access to file records), apps work directly with the base tables. Such apps often handle system maintenance tasks such as compression or repair routines.

Relational systems rarely permit routine access to the base tables, much less let you add indexes or new tables on the fly. You can create applications that work directly with the indexes and tables, but in most serious systems—especially large enterprise installations—their administrators prevent access to base tables. Instead they create SQL views or stored procedures that extract data in carefully orchestrated ways. Administrators appreciate this, along with the ability to assign permissions to procedures and views that cannot be assigned to the base tables.

Fred's team had done a pretty good job of setting up the database. They used the Access Upsizing Wizard for the initial conversion, and hired a consultant to build a number of intelligent stored procedures to return and update the data. The consultant completed the task and then went off to Australia.

The errant consultant's procedures asked a series of questions that returned a varying number of result sets based on the answers. Fred's group was having trouble with the complexity of the result sets. For example, the front-end application queries the database to find available hotel rooms that match a set of criteria. Since the hotel rooms vary so much in type and suitability, the agent using the application was permitted to walk through a decision tree to permit faster selection of the room best matching the requested features. The result sets provide answers to the questions:

- What hotels in the area chosen have available rooms? Once a hotel is chosen…
- How many rooms does the chosen ho-

William R. Vaughn is the author of A Hitchhiker's Guide to Visual Basic Version 4.0 and SQL Server 6, *due in April from Microsoft Press. He works as a senior writer on the Visual Basic team at Microsoft Corp., specializing in client/server and other data access issues. Bill can be reached at billva@microsoft.com, and by telephone or fax at Beta V, 206-556-9205.*

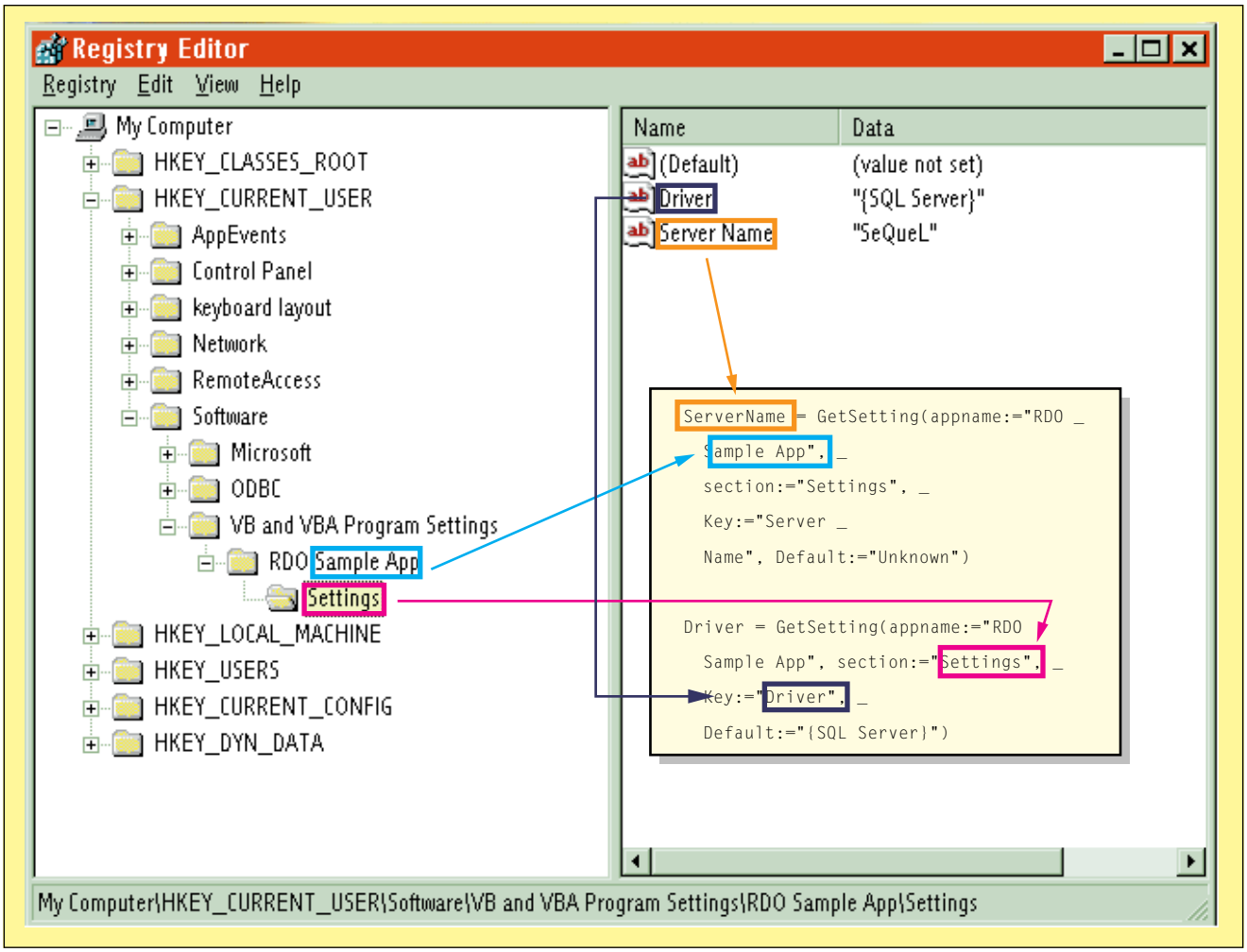

FIGURE 1 ***Keys to the Kingdom.*** *Windows 95's RegEdit program lets you display the registry keys used in the application. These keys were created manually at first and filled in with code from the application. The Get Setting function pulls out the information based on app name, section and key, with a single function for each entry. This process uses less than half the code it would have taken to use the Windows API registry functions, and eliminates the risk of blowing up your app by overlaying memory when the values come back from your API calls.*

tel have for the date given? If more than zero, then…

- How many of these available rooms are the right type, with the right number of beds?
- Does the cost of these rooms fall within the requested price range?

If the stored procedure discovers a "no" answer to any of these questions, it doesn't go to the next step. This design separated each question into a separate procedure. This way, given appropriate parameters, the base procedure can call subsequent procedures down the line. In other situations, lower-level questions can be asked by calling the subordinate queries directly from the app. And to Fred's consternation, he discovered that each SQL operation returns a result set, so one procedure might return two or more result sets.

In both VBSQL and RDO you can easily capture the returned arguments sent back by each step. In DAO, on the other hand, you must execute a make-table query to build a permanent Jet table that must be dealt with once the operation is done with the result set. DAO users must also deal with name collisions and other multiuser concerns that RDO handles automatically.

Note that if the procedure performed action queries, and you need to capture the rows affected by the Update statements, ODBC does not support this functionality for multiple result set queries.

Don't be surprised if your own stored procedure implementation rivals the convolution of Fred's hotel query plan. Consider this technique to examine result set return values. Basically, a query—especially one using stored procedures—can return:

- A "return status" set by passing back an INT value from the procedure.
- Zero or more sets of data rows—one for each select statement that returns rows.
- Zero or more output parameters.
- A "rows affected" value for non-select queries like update, insert, or delete statements.

All but the rows-affected value can be retrieved from procedures that return one or several sets of results (see Listing 1). The code in Listing 1 comprises a test procedure that makes use of most of these return arguments. The procedure accepts the name of a city, and finds any hotels with available rooms. It also returns the range of prices for those rooms and the total number of vacant rooms in the city's hotels. Try to ignore the way Fred wrote the select statement and focus on the mechanism used to pass arguments to and from the procedure. The procedure accepts an input parameter and returns two output parameters as

well as a row set.

Next you need techniques to call stored procedures. RDO lets you call stored procedures in a variety of ways, but if you want to have RDO help you pass parameters back and forth, you really should use the rdoParameters collection associated with the rdoPreparedStatement object. This also permits you to set a number of important options like MaxRows and RowsetSize, and access the RowsAffected property after it executes. But before exploring these nuances, look at your options. In each case, a connection (*Cn*) has been established to the back-end server:

```
Dim Cn as rdoConnection
Set Cn = rdoEnvironments(0)._
   OpenConnection(dsname:=" ", _
   prompt:=rdDriverNoPrompt, _
   connect:="UID=;PWD=;_
   Database=Workdb;"_
   Driver={SQL Server};_
   Server=SeQueL;" )
```

This hard-codes the driver and server name into the connect string, so you don't have to install and manage a data source name (DSN) on every workstation that runs the app. As an alternative, I also showed Fred how to capture registered DSNs from the registry and choose one from the list.

A better approach, however, saves the server name in the registry. Because the VB4 help example for registry functions is dysfunctional, I had to give him a nudge in the right direction. In his case we set up the registry when the application was installed and used this setting thereafter. Establishing a registry setting was straightforward:

```
' Place the settings in the registry.
SaveSetting "RDO _
   Sample App", "Settings", "Server Name", "SeQueL"
SaveSetting "RDO _
   Sample App", "Settings", "Driver", "{SQL Server}"
```

I run this code when I'm first setting up the program. Later in the initialization routine, we simply read in the registry settings. Note these are all stored in a special registry location (see Figure 1). Fred wanted to know if he could use another registry path. He also wanted to know why he had to use the registry instead of an INI file. I told him that his path must use \HKEY_CURRENT_USER\Software\VBasic and VBA program settings. "Of course," I added, "you could use the Windows API settings." Fred looked like my four-year-old when she finds brussels sprouts heaped on her plate. He didn't exactly enjoy using the APIs directly.

The registry is the new de facto-standard location for storing program settings such as these. By doing so, Windows 95 and Windows NT can help manage the settings as part of the registry security mechanism. Other applications can also locate these settings more easily; they don't get confused about what INI file is used or where it's saved. And users have a harder time accidentally erasing or munging the registry. This happens all the time when you use INI files.

To get the settings out of the registry, you pull up the server name and ODBC driver to use from their corresponding registry slots:

```
Dim ServerName As Variant
Dim Driver As Variant

ServerName = GetSetting(appname:="RDO _
   Sample App", section:="Settings", Key:="Server _
   Name", Default:="Unknown")
```

VB4

```
CREATE PROCEDURE FindHotelInCity

@CityWanted  Varchar(128),
@MaxPrice Money OUTPUT,
@MinPrice Money OUTPUT

AS

declare @cnt int
Select Hotel, Count(*) "Available Rooms"
from Hotels H, Rooms R
Where H.ID = R.HotelID and
City = @CityWanted and
R.Vacant = 0
group by Name

Select @cnt = Count(*)
from Rooms R
where HotelID in(select ID from hotels _
  where City = @CityWanted)c
and R.Vacant = 0

Select @MaxPrice = Max(Price) , @MinPrice = Min(Price)
from Hotels H, Rooms R
Where H.ID = R.HotelID and
City = @CityWanted and
R.Vacant = 0

Return @cnt
```

LISTING 1 ***Get the Best Rate.*** *A Transact SQL stored procedure, FindHotelInCity, returns the highest- and lowest-priced room at any hotel in an indicated city. The procedure accepts a single input parameter and returns two output parameters. It also returns a set of rows for each hotel in the desired city indicating the hotel and a count of its rooms.*

```
Driver = GetSetting(appname:="RDO _
   Sample App", section:="Settings", Key:="Driver", _
   Default:="{SQL Server}")
```

Then you stick the settings into the RDO OpenConnection method to gain access to the specific server and driver:

```
Dim Cn as rdoConnection
Set Cn = rdoEnvironments(0).OpenConnection _
   (dsname:="", prompt:=rdDriverNoPrompt, _
   connect:="Server=" & ServerName & ";Driver=" & _
   Driver & ";" & "UID=;PWD=;Database=Workdb;" )
```

Both methods assume that the SQL Server has domain-managed (integrated) security enabled, and that as a result users don't need to provide a password or user name. As long as they get access to the workstation and can log into the network domain controller, they can gain access to SQL Server. The Windows logon interface captures both the UID and PWD values and passes the UID along to the SQL Server security manager, as long as you code them according to the method I showed to Fred.

Fred wanted to know if he had to use the new coding syntax that kept appearing in my examples. I told him that I like the new named-argument syntax because I don't have to put in commas as placeholders for missing arguments. But you can still use the old comma-delimited syntax if you are into commas.

Once Fred's application was connected, he needed to call the procedure we had set up. Fred had been using ISQL to manage these procedures, but found it a pain to have to remember to drop the procedure each time.

"Aren't you using SQL Enterprise Manager?" I asked.

"Sure, on the server. But that system is across town in the

central IS facility."

I asked, "Why don't you install it here on your Windows 95 system?"

"You can?"

We spent the next few minutes running Win95's SQL Server setup utility. Before long we had registered into his remote server and were doing maintenance on his stored procedures. By keeping one window open for SQL Enterprise Manager we could keep an eye on the connections being used and tune the procedures while working on the new Visual Basic app simultaneously.

Then we created an rdoPrepared-Statement to execute the procedure and manage the various input and output parameters.

"Where are the input arguments for the procedure coming from?" I asked. "The stuff going into the procedure—are you going to get them from a drop-down list box or from a text-box control?

"They're coming from a text box for now," Fred said. "We might use a list box later. The old application filled a list box from data we read in from a separate initialization file, but we don't plan to use that file anymore. Now we put the data in a new database table."

I suggested we use RDO to set up a new database table, and bind one of the new DBList or DBCombo controls to the database table. But Fred declined the offer. "I think I can handle that later," he said. "For now, let's just use the text boxes."

CAREFUL SYNTAX

I showed Fred how to code the CreatePreparedStatement function. He tried to do it earlier, but he received object reference errors due to faulty SQL syntax. When RDO asked the ODBC drivers to execute SQLPrepare, he'd get an rdoPreparedStatement, but no rdoParameters collection. I showed him the right syntax for his procedure. As we worked through the coding for CreatePreparedStatement we failed not on the create statement itself, but on the first reference to the rdoParameters collection. If the process of checking the syntax of the SQL argument fails, few outward signs will tip you off.

"Shouldn't the rdoErrors collection have an entry to show that the syntax was wrong?" Fred wondered.

"You'd think so, but no, bad syntax is not caught at the Visual Basic level, so there are no entries we can test for." I answered. "You just have to be careful. Look here. We left out the 'call' keyword. That's the problem."

To show Fred I had not made up this answer, I added several lines of code to examine the rdoErrors collection. We did find the informational messages returned by the driver to indicate that SQL Server had changed databases and set the default language:

```
Sub ShowErrors()
Dim er As rdoError
For Each er In rdoErrors
   Debug.Print er.Description
Next
End Sub
```

This code dumped three errors that were really just informational messages.

We made sure to clear the rdoErrors collection after having opened the connection, to see if errors popped up. We refocused on CreatePreparedStatement again.

""Why did you add the database name to the SQL statement?" Fred wanted to know.

"Without it, the ODBC driver can't always find the procedure."

"Doesn't it just send down the procedure to be executed?"

"Not really. The ODBC driver and RDO must determine the data type of each parameter sent and received. This way it can bind to the variables and properly format subsequent calls that reference the returned arguments. To do this, the ODBC driver queries the SQL Server for specific information about each parameter specified. Without explicit addressing, especially for those procedures in the master database, the driver can't locate the procedure. To be safe, I always add the full path to the procedure. If you want to see details about how this is done, check out the ODBC logs."

But he wanted to get the form-load procedure written, so we did it (see Listing 2).

Fred didn't notice that I had selected the ODBC cursor driver. This simplifies the process of handling multiple result-set stored procedures. While you can execute multiple result-set procedures (procedures with more than one select statement) using the server-side cursor library, you must create a single-row, forward-only, read-only cursor.

We can use rdoPreparedStatement later in the application as well. The next step involves coding the command procedure that runs the query and returns the rows. We just have to set the parameter and use either the Refresh method to rebuild the existing rdoResultset or the OpenResultset method to create a new rdoResultset.

"Can't we just create a new rdoResultset each time?" Fred asked?

"Sure. But RDO, unlike DAO, keeps the old rdoResultset objects when you don't explicitly close them. To avoid filling up memory, you must remember to close them down when you're done. But your design doesn't let us know if we're done until we need to create another. So we need to tell VB to close the old rdoResultset before we create another:

```
While rdoResultsets.Count
   rdoResultsets(0).Close
Wend
```

This routine closes down all rdoResultset objects, so you might not want to use it as is. Since the rdoResultset `rs` is a global variable, you could just say:

```
rs.Close
```

REFRESH SPEEDS UP SQL SERVER

However, one of the faster strategies that leverages the existing rdoResultset uses the Refresh method: you simply change the rdoPreparedStatement parameter and use the Refresh method to re-run the query, which not only makes programming easier, but it improves performance. This way SQL Server does not have to rebuild and recompile the procedure each time it runs. That's the whole idea behind an rdoPrepared-Statement. It builds a custom stored procedure for you, designed to accept parameters.

"OK, let's use the Refresh method." Fred was caught up in the process now.

"Fine. But if we do that, we need to add a line or two to create the first rdoResultset based on the query."

We needed to add code that either created the first rdoResultset or executed the Refresh method. The rdoResultset is created using rdOpenStatic (like a DAO snapshot) and the rdConcurReadOnly (which makes the result set read-only) to improve performance.

Generally, don't ask for features such as updatability if you don't need them. As a matter of fact, you could probably use rdOpenForwardOnly because you don't expect to move around in the cur-

VB4

```
Private Sub Form_Load()

Set en = rdoEnvironments(0)
en.CursorDriver = rdUseOdbc

ServerName = GetSetting(appname:="RDO Sample App", _
  section:="Settings", _
  Key:="Server Name", Default:="Unknown")
'
Driver = GetSetting(appname:="RDO Sample App", _
  section:="Settings", _
  Key:="Driver", Default:="{SQL Server}")
'
' Next, we stick them into the RDO OpenConnection method
' to gain access to the specific server and driver:
Connect = "Server=" & ServerName & ";Driver=" _
  & Driver & ";" & "UID=;PWD=;Database=Workdb;"
Debug.Print Connect
Set Cn = rdoEnvironments(0).OpenConnection(dsname:="", _
  Prompt:=rdDriverNoPrompt, Connect:=Connect)

rdoErrors.Clear_
SQL = "{ ? =  call Workdb..FindHotelInCity (?,?,?)};"
Set ps = Cn.CreatePreparedStatement("", SQL)
If rdoErrors.Count > 0 Then ShowErrors

If ps.rdoParameters.Count > 0 Then
  ps(0).Direction = rdParamReturnValue
' ps(1) does not have to be set.
' The default direction is rdParamInput
  ps(2).Direction = rdParamOutput
  ps(3).Direction = rdParamOutput

Else
  MsgBox "Could not create _
    prepared statement query. Call Fred at home"
End If


End Sub
```

LISTING 2 ***Prepare a Statement.*** *Sub Form_Load gathers the current server name and driver from the registry. Using these parameters, the procedure then opens a connection and tries to build an rdoPreparedStatement to be used later.*

sor once it is built: we move forward as we fill a list-box control with the row. But I'll just leave it as static for now. We then extracted the contents of the first and second output parameters and the return value, which were placed in text-box controls on the form (see Listing 3).

The ShowRows subroutine simply dumps the rows we found to a list box. RDO is coded so much like DAO, this was easy:

```
Sub ShowRows()
List1.Clear

While Not rs.EOF
   List1.AddItem rs(0)
   rs.MoveNext
Wend

End Sub
```

"But doesn't the procedure return more than one set of results?" Fred asked.

"Sure, but output parameters handled all the results of the internal procedures," I answered. "Look at every select statement in the stored procedure. They assign values to stored procedures variables like @MaxPrice and @cnt; they don't

VB4

```
Private Sub ShowHotels_Click()
ps(1) = Text1.Text
'While rdoResultsets.Count
'rdoResultsets(0).Close
'Wend
If rs Is Nothing Then
  Set rs = ps.OpenResultset(rdOpenStatic, _
rdConcurReadOnly)
Else
  rs.Requery
End If
Text2.Text = ps(2)       ' Output parameter 1
Text3.Text = ps(3)       ' Output parameter 2
Text4.Text = ps(0)       ' Return Value
ShowRows
```

LISTING 3 ***Reservation, Please?*** *The Sub ShowHotels procedure opens an rdoResultset based on the chosen hotel. Once the result set has been created, subsequent executions simply rebuild it using the rdoPreparedStatement and the Refresh method.*

return rows. If you run a select statement like this from ISQL or SQL Enterprise Manager, you don't get the results of the query at all. You just get the error message, 'number of rows affected,' because the results are stored internally and not returned as a row.

"If you add a select statement that returns a value and makes sure there is a column name, you'd be able to capture the results as the second or "nth" result set. Suppose you add a select statement to the procedure:

```
Select ID,Name,Zip, Substring(City,1,2) CityCode _
   From Hotels Where City = @CityWanted
```

Note that when you issue a select statement that includes a function (like Min, Max, or Substring), SQL Server doesn't assign a column name, making it hard for the interface (like the ODBC driver) to recognize and manage the column. To help, I always label the columns using the TSQL aliasing syntax.

"Now, we can adapt the code to deal with this extra result set if you want to," I said. I'll set up a new stored procedure that returns another result set using our little select statement, with a bit extra:

```
r = rs.MoreResults
' Get the next set of results.
If r Then
   List2.Clear
   Do Until rs.EOF
      List2.AddItem rs(0) & ":" & rs(1) & " - " & rs(2)
   rs.MoveNext
   Loop
End If
```

The first line asks the RDO interface if there are more result sets to deal with. If MoreResults returns True, the code pulls down the result rows and puts them into a list box. When you move to the next result set, you lose access to the previous result set. It does not remain in the rdoResultsets collection for later examination.

By the time I got home, Fred had most of his app put back together and one of our cats had burrowed into a throw cushion on my blue chair. I lifted her, pillow and all, and put her on the floor at my feet. I had just settled back into the blue chair when the phone rang. I didn't answer it.

My daughter called down from upstairs. "Dad! It's some guy from MCI…." ×

