Make Your Client/Server Programs Scream!

Click & Retrieve Source CODE!

BY MICHAEL MEE AND EMILY KRUGLICK

With a little more coding effort and some new approaches, you can deliver a lot more client/server performance.

Sure, you can build client/server apps with what you already know about ISAM or PC database programming. But if you learn to take advantage of the new features found in VB4 32-bit (and SQL Server 6.0), you could find yourself turning even simple client/server apps into rockets that run two to 22 times faster (see Figure 1) and consume fewer resources to boot. And you don't have to drop VB3 to get there. Many of these optimizations apply equally well to VB3 and VB4.

Some of the techniques involve code changes. Others involve changing your user interface to work with the server a different way. But nothing we talk about requires you to get a brain transplant. Mainly, you need to unlearn some habits that served you well in the past, and learn some new ones that will serve you well in the future.

Moreover, in this article we've focused on coding strategies that have proven successful with people who haven't written much client/server code, and who generally need to get their app done yesterday.

For the sake of example, let's look at such an app: a simple database program called BookSale, programmed the old ISAM/PC database way. Bookstore customers use BookSale to look up books by author or title and buy the books they choose. The app allows you to add more records to the database, enabling you to see how the app functions with different amounts of data.

To revamp BookSale, let's start with linked tables—perhaps the most misunderstood part of Jet/DAO remote database access. Jet provides two ways of accessing data from a remote database. First, you can open the database directly: this method doesn't demand a local Microsoft Jet database. The ODBC connect string (such as "ODBC;dsn=<dsn name>;database=<database server>;uid=<user id>;pwd=<password>") passes directly to the OpenDatabase method, establishing a direct connection to the remote database.

Second, you can install a local Jet database and create linked tables (called "Attached Tables" in Access 2.0) pointing to the remote database tables from within this local database. When using linked tables, the application first opens the local database and then uses the linked tables to retrieve the remote data.

BookSale was programmed with direct ac-

Michael Mee, a program manager in the Data Access and Retrieval Technologies group at Microsoft, designs and coordinates the Data Access Objects used by products such as Access, Visual Basic, and Visual C++. Emily Kruglick, a software test engineer at Microsoft, is currently testing DAO, especially for client/server usage. Michael and Emily were technical contributors to the recent "Microsoft Jet Database Engine Programmers Guide" from MS Press. This article, derived from chapter 9, "Developing Client/Server Applications," is published with the permission of Microsoft Corp.

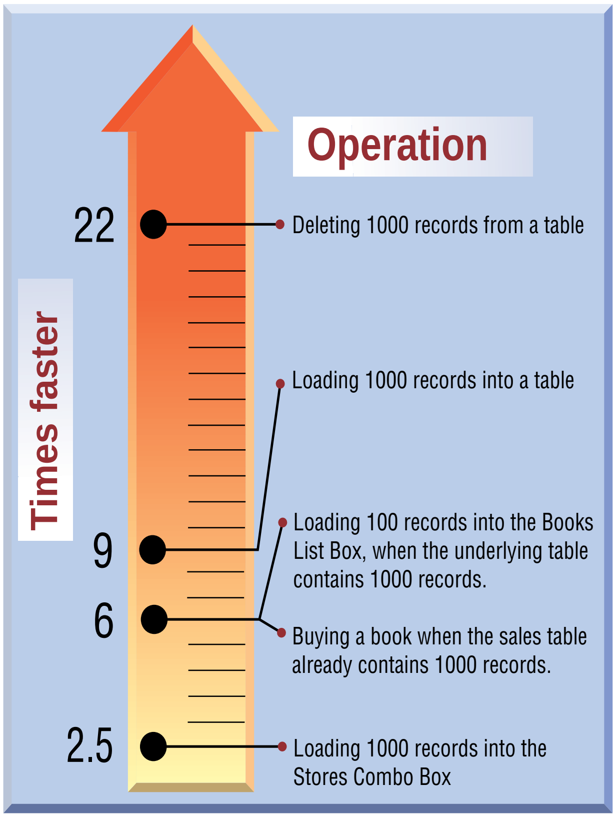

FIGURE 1 ***Client/Server Coding** can reap spectacular performance gains. A simple production app, BookSale, was first coded using conventional ISAM/PC database-era techniques, then optimized for client/server, for two to 22 times the performance.*

cess because it sounded like it would be faster. But it isn't. Every time BookSale accesses a remote table, Microsoft Jet must retrieve information about tables, such as field names and types, before it can retrieve the actual data. But when you link a table, Jet caches the information in the local Jet database, making it quickly accessible each time BookSale opens the table, eliminating several round trips to the server.

BookSale needs the speed linked tables provide. We do this by having most of its access to the remote database go through the PUBS.MDB, which contains linked tables to the remote-publications database. For an example of this change in code, look at the LoadStores subroutine in the BOOKS.BAS module (see Listing 1). The original line:

```
Set dbPubs = OpenDatabase("", False, _
   False, gCONNECT$)
```

is replaced with

```
Set dbPubs = OpenDatabase(App.Path & _
   "\pubs.mdb", False, True)
```

Performance is boosted dramatically when only a few records are being accessed. Because well-designed client/server apps should access only a few records at a time, this modification can lead to significant gains.

The issue of where to put the PUBS.MDB file depends a lot on your environment. Putting it on a read-only network share, and opening the database as read only and exclusive, enables you to avoid most maintenance problems and provides a single point of update. However, if you put it on the user's local machine, your app can create local queries or download data locally. You may want to use a combination of the two.

"DIRECT" ISN'T ALWAYS "FASTER"

VB3 users often access remote databases directly because Jet supports SQL passthrough, which tells the Jet Engine to pass SQL queries directly to the remote database engine for evaluation, instead of Jet evaluating it and passing its own calls to the server. If you use VB3 with a Jet database, it ignores the SQL passthrough option because it make sense only for server data.

VB4 now lets you use SQL passthrough with an MDB database (most likely containing linked tables). The Database object's Connect property, normally unused with MDBs, can be set to an ODBC connection string and be used for subsequent passthrough commands.

Another peek into BookSale's LoadStores subroutine reveals how this extension of SQL passthrough works:

```
Set dbPubs = OpenDatabase(App.Path & _
   "\pubs.mdb", False, True)
dbPubs.Connect = gCONNECT$ _
   '"ODBC;dsn=<dsn name>;
   'database=<database server>;
   'uid=<user id>;pwd=<password>"
```

We can find more ways to speed database access. Every operation on an ODBC database requires a mechanism for conversing with the database server. *Connections* provide this mechanism, as well as the key to improved performance.

To simplify things, Jet hides the concept of connections. But to make things faster for us, we need to see what Jet is doing with connections on your behalf.

The first time you issue a command that accesses the server, such as opening a table, Jet opens a connection to the server. Connections are relatively expensive, so when you finish the operation Jet will keep the connection open in anticipation of the next operation. Similarly, if you have a connection open for a long time but aren't doing anything with it, Jet will silently close it, then reopen it when you resume working.

In some situations, you may want to control the timing of opening connections yourself, rather than when the first operation occurs. If your program has a startup or initialization routine already in place, you may want to establish connections in that routine. Doing so front-loads the execution time of your application, making it easier to avoid delays later on. Also, connections to your particular database might take a long time to establish if the database server resides across the country, or if you're relying on a modem connection to the server.

To minimize delays in opening forms or populating tables, try connecting the database server when your application starts. This technique, called "preconnecting," lets you set up the connection to the database before users request data. Preconnecting effectively speeds the first data retrieval operation in your app by moving the delay to application startup, where it will be less noticeable to users.

In optimizing the sample application, we'll program the initial connect to the database to occur during the load of the first form. No other database activity is occurring at this time. The Form_Load routine of the frmStart Form performs the initial connect:

```
Set dbPubs = OpenDatabase("", False, _
   False, gCONNECT$)
dbPubs.Close
```

This routine appears to do nothing, so it might look like it's wasting time. But it actually establishes a connection that

VB4

```
Sub BuyBook(lstboxidx%)
   Dim dbPubs As Database
   Dim strStoredProcCall$
   Dim qdfInsertSale As QueryDef

   If gintQuant% > 0 Then

      Set dbPubs = OpenDatabase(App.Path & _
         "\pubs.mdb")
      dbPubs.Connect = gCONNECT$

      strStoredProcCall$ = "InsertSale '" & _
         gstrStoreId$ &  "', '" & strOrderNumber$ _
         & "', '"
      strStoredProcCall$ = strStoredProcCall$ & _
         Now & "', "  & gintQuant% & ", '" & _
         gstrPayTerms$
      strStoredProcCall$ = strStoredProcCall$ & _
         "', '"  & gstrTitleArray(lstboxidx%) & "'"

      dbPubs.Execute strStoredProcCall$, _
         dbSQLPassThrough

      dbPubs.Close
   End If

End Sub
```

LISTING 1 ***Record Set Insertion.*** *This subroutine shows how to call a stored procedure to insert a record into the database. First it opens the database and sets the connect string to point to the appropriate SQL Server database. Then it sets up the stored-procedure call. Finally, it executes it using SQL passthrough, to have the back end do the processing.*

hangs around waiting for the next database call. When that call comes, the in-place connection provides faster performance than you'd get without the preconnect.

Of course, sometimes you may not want to cache the connection. You can prevent idle connections from being cached by setting the ODBC connection timeout value to a low number. By default, you get a ConnectionTimeout value of 600 seconds (10 minutes). To change this value, set the ConnectionTimeout value in the ODBC section of the application's registry entry, or set the value in the \Jet\3.0\Engines\ODBC registry entry—this sets it globally for all apps using Jet.

Jet closes the connection after the specified number of seconds of idle time unless prevented by a condition. Thus Jet closes connections even if forms displaying remote data are still open. Jet automatically reconnects when the application again requires a connection.

DATA-RETURN BOTTLENECKS

Once the app is using linked tables and preconnecting, we can turn our attention to the data being brought back. The biggest bottleneck in client/server access stems from the amount of time taken to communicate with servers across networks. Finely tuned apps request only needed data. Less finely tuned apps tend to choke because they request more fields and/or records than they need.

Start your data-return tune-up by making sure you request only the columns you actually need. Some programmers lean on the old standby "Select * from <tablename>" no matter what fields they want. This can really get you in trouble if your record set contains large value fields (such as memo fields) and you are opening a snapshot. In this case all the data stored in these large fields will march into your application whether or not you read from the fields.

The lesson to learn in this regard is to carefully evaluate the columns you request. You may want to hold off requesting the larger data fields until the user specifically asks to see that data: you could create a detail form for displaying more detail about a specific record. One way or another, never request fields that you don't intend to use.

You can also limit rows returned across the network. While DAO provides a lot of great functionality, you don't have to use it—like bringing back all rows from a table and then using FindFirst and other DAO find functions to rummage for the rows actually needed.

Instead, try using a Where clause on the initial SQL statement that will filter out unneeded records. You get the speed that usually comes from letting the back end do the work of filtering out records before they go down the wire.

We can see a good example of this optimization in BookSale's LoadListBox routine in its BOOKS.BAS module, which loads the books list box first, without any limits on data being passed back to the app:

```
strWhereClause$ = " where"
'Set up where clause
'for count and select
strWhereClause$ = strWhereClause$ & _
   " titles.title_id = _
   titleauthor.title_id"
strWhereClause$ = strWhereClause$ & _
   " and titleauthor.au_id = _
   authors.au_id"
```

And then we optimize it, adding an extra line to the Where clause to limit rows:

```
strWhereClause$ = " where"
'Set up where clause for
```

```
'count and select
strWhereClause$ = strWhereClause$ _
   & " " & strSearchCriteria$ & " and"
'We add the criteria on to
'the string so it can choose
'only the data we need.
strWhereClause$ = strWhereClause$ _
   & " titles.title_id = _
   titleauthor.title_id"
strWhereClause$ = strWhereClause$ _
   & " and titleauthor.au_id = _
   authors.au_id"
```

The Where clause's new filter has it selecting only records that match the search criteria. Because only needed rows are returned now, we don't need to use find methods to select data to add to the list box. All records will now be added to the list box.

Now we're only bringing back the data needed, and using the fastest overall connection method to the remote database, so the actual records being returned have been optimized. Next let's optimize how those records are returned.

SOMETIMES DYNASETS ARE FASTER

DAO provides two different types of record sets that you can use with remote databases: dynasets and snapshots (DAO's table-type record set won't work on remote databases). Dynasets, usually updatable and dynamic, show changes that others are making to the underlying data.

Snapshots provide a picture of the data at a given instant. You can't update snapshots and they don't usually show changes to the data after being opened. Because snapshots don't need to support as much functionality, they usually work faster. But if a snapshot contains any long value fields (such as memos or OLE objects), it delivers them whether or not you ever read them. So in this case you may get more speed from a dynaset, which only brings across the data you ask for specifically.

But you don't want to use a dynaset accidentally. Many applications don't specify a type when opening record sets. They then use the default type—for ODBC data, a dynaset. Here you can dramatically improve data-return speed by requesting snapshot-type record sets when you only to read data and you won't be pushing long value fields down the wire (for more information, see the sidebar "Dynasets versus Snapshots").

BookSale used dynasets when it didn't need to, as in the LoadStores and LoadListBox routines in the BOOKS.BAS module:

```
Set snpBooks = dbPubs.OpenRecordset_
   (strSelectBooks$, dbOpenDynaset)
```

Changing this load to a snapshot-type record set helped a lot (see Listing 2):

```
Set snpBooks = dbPubs.OpenRecordset_
   (strSelectBooks$, dbOpenSnapshot)
```

Beyond picking the right record-set type for the situation, you can also place different options on a record set to make it run faster. For instance, you can use Append Only to open a dynaset that you'll only use to add data to the database. This saves time, because the record set does not have to retrieve the pre-existing records from the database. Instead it needs only to add new records.

Another option, dbForwardOnly, creates a record set that only scrolls forward one record at a time. While more restrictive than a regular record set, it does not have to keep track of as much supporting information (such as bookmarks), making it faster than using a regular record-set.

Clever record-set management certainly boosts application and network performance (see Listing 3). Add to that judicious use of the SQL passthrough facility and you can achieve escape velocity. SQL passthrough was made just for client/server apps. It takes a SQL statement and passes it across the network to the database server.

The server returns the results to the app. For Select statements, Jet usually sends the entire statement to the server, but for operations such as Update, Jet does a mixture of local and remote processing (to provide partial update capability). This requires more network traffic and more Jet operations, making it slower.

SQL passthrough bypasses Jet in executing non-Select operations (see Listing 4). Say you're deleting many records from a table. If you can build a single SQL statement to select the records, then you can delete the records with a single delete statement, using the same Where clause. But you can use dbSQLPassthrough to send the statement directly to the server for efficient execution. Otherwise Jet evaluates the statement, deleting each record one by one, as our original BookSale app does:

```
db.execute "Delete * from authors
where
au_id like 'TT*'"
```

Now let's boost by 22 times BookSale's performance during deletion of records from tables in the pubs database. We accomplish this by adding the dbSQLPassthrough flag to the DataCleanUp routine in the DATALOAD.BAS module:

```
db.execute "Delete authors where au_id
like 'TT%', dbsqlpassthrough
```

The SQL statements don't match exactly, though they provide the same functionally. This demonstrates that when you use SQL passthrough you must write the SQL statement in the dialect of the database server. If you don't use SQL passthrough, you must use the Jet syntax. In most cases the SQL syntax will be identical.

STORE PROCEDURES ON SERVERS

dbSQLPassthrough helps get you into true client/server territory. And using it to invoke stored procedures makes you a client/server pro. Because stored procedures are precompiled, preoptimized execution plans, calling one avoids any parsing—not so when you deliver a SQL statement. In this case the database server must compile and optimize that statement before executing it. Stored procedures let you avoid all that overhead.

VB4

```
Sub LoadListBox(stCriteria$)
   Dim dbPubs As Database
   'Database to retrieve data from.
   Dim snpBooks As Recordset
   'Recordset pointing at the database.
   Dim snpCount As Recordset
   'Recordset to contain count of records.
   Dim strListBoxRow$
   'Row to be added to the list box.
   Dim strSelectBooks$
   'Used to put together sql strings.
   Dim strCurrentTitle$
   Dim strSearchCriteria$
   'Criteria String to search on,
   'based on what field is being searched.

   Dim intSpotInTitleArray%
   Dim strAuthors$
   Dim strWhereClause$
   Dim intRecordCount%
   Dim intRecordSetOptions%
   Dim strSearchChar$
   Dim intRecordSetType%

   intRecordSetOptions% = dbSQLPassThrough + _
      dbForwardOnly

   intRecordSetType% = dbOpenSnapshot

   frmBookList.lstBox.Clear
   'Clean out all old values from database.
   Set dbPubs = OpenDatabase(App.Path & "\pubs.mdb")
   dbPubs.Connect = gCONNECT$

   strSearchChar$ = "%"

   If frmFind.optAuthor Then
      'Set find string for use later.
         strSearchCriteria$ = "au_lname like '" & _
            stCriteria$ & strSearchChar$ & "'"
         strSearchCriteria$ = strSearchCriteria$ &  _
            " or au_fname like '" & strSearchChar$ _
            & "'"
      Else
         strSearchCriteria$ = "title like '" & _
            stCriteria$ & strSearchChar$ & "'"
      End If

      strWhereClause$ = " where"
      'Set up where clause for count and select
      strWhereClause$ = strWhereClause$ & " " & _
         strSearchCriteria$ & " and"
         'We add the criteria onto the string to
         'choose only the data we need.
      strWhereClause$ = strWhereClause$ & _
         " titles.title_id = titleauthor.title_id"
      strWhereClause$ = strWhereClause$ & " and _
         titleauthor.au_id = authors.au_id"

      Set snpCount = dbPubs.OpenRecordset( "select _
         count(*) from titles, titleauthor, _
         authors" & strWhereClause$, _
         intRecordSetType%, intRecordSetOptions%)

      intRecordCount% = snpCount(0)
      'Get the count.
      snpCount.Close
      If intRecordCount% = 0 Then
         MsgBox "There are no books matching _
            this criteria"
      Else
         strSelectBooks$ = "Select titles.title_id, _
            title, "
         strSelectBooks$ = strSelectBooks$ & _
            "price, _
```

CONTINUED ON NEXT PAGE.

LISTING 2 ***Using Snapshots.*** *With this subroutine you use forward-only SQL passthrough snapshots to retrieve data that does not need to be updated. For best performance, we've chosen the type of record set to use based on the definition of the data that the record set will retrieve. You can also see how to create a Where clause in your SQL call to limit the amount of data returned to what you need—always a good practice.*

They also let you put application logic on the server instead of the client program. For example, a stored procedure might add a record to a table and also add all the foreign keys to other tables within that record if they didn't already exist. You can execute such a stored procedure with a single call across the network to the database server. In contrast, without stored procedures you must create an insert SQL statement for the main table and query each of the foreign tables to see if the key needed to be added, and then send insert statements to any foreign tables that needed keys added.

For every step, you must send a SQL statement across the network, and some will cause data to be retrieved. If you're dealing with a table that has two foreign tables associated with it and the keys aren't already in the tables, you'll send out five server calls. Yet only two of those calls will return data (the ones querying to see if keys exist).

BookSale was built using the usual calls, adding a record using a record set and the addnew/update functionality:

```
Set dbPubs = OpenDatabase("", False, _
   False, gCONNECT$)

Set rstSales = _
   dbPubs.OpenRecordset("Select * _
   from sales", dbOpenDynaset)

rstSales.AddNew
rstSales!title_id = _
   gstrTitleArray(lstboxidx%)
rstSales!stor_id = gstrStoreId$
rstSales!ord_num = gstrOrderNumber$
rstSales!ord_date = Now
rstSales!qty = gintQuant%
rstSales!payterms = gstrPayTerms$
rstSales.Update
rstSales.Close

dbPubs.Close
```

Now we'll optimize all this with a stored procedure to do the same record-adding update. We call the stored procedure on the server directly, using dbSqlpassthrough:

```
Set dbPubs = OpenDatabase(App.Path & _
   "\pubs.mdb")
dbPubs.Connect = gCONNECT$
s$ = "InsertSale '" & _
   gstrStoreId$ & "', '" & _
   gstrOrderNumber$ & "', '"
s$ = s$& Now & "', " & _
   gintQuant% & ", '" _
   & gstrPayTerms$
s$ = s$ & "', '" & _
   gstrTitleArray(lstboxidx%) & "'"
```

CONTINUED ON PAGE 68.

CONTINUED FROM PREVIOUS PAGE.

```
         pubdate, au_lname, "
      strSelectBooks$ = strSelectBooks$ & _
         "au_fname from titles, "
      strSelectBooks$ = strSelectBooks$ & _
         "titleauthor, authors "
      strSelectBooks$ = strSelectBooks$ & _
         strWhereClause$ & " order by title"
      Set snpBooks = _
         dbPubs.OpenRecordset(strSelectBooks$, _
         intRecordSetType%, intRecordSetOptions%)
         'This should return records since
         'we have a count.

      'Verify you have found records before
      'assuming they are there.
      ReDim gstrTitleArray(intRecordCount%)
      'Allocate Space.

      strCurrentTitle$ = ""
      'Clear Title String.
      strListBoxRow$ = ""
      'Clear String for Inserting into list box.

      intSpotInTitleArray% = 0
      While Not snpBooks.EOF
      'The listbox only takes one column, so we must
      'parse the data into a string to be added to
      'the list box.
         If strCurrentTitle$ <> snpBooks!title_id Then
         'First we see if we are looking at a new
         'title, or if this is one we have already
         'started working on
            If strListBoxRow$ <> "" Then
               'If it is new then as long as this is
               'not the first record--
               strListBoxRow$ = strListBoxRow$ & _
                  strAuthors$
                  'Add the Author string to the rest
                  'of the title
               frmBookList.lstBox.AddItem _
                  strListBoxRow$
                  'Add the whole thing to list box.
            End If
            strAuthors$ = ""
            'Clear author string since it
            'is a new title.
            gstrTitleArray(intSpotInTitleArray%) = _
               snpBooks!title_id
               'Put title in title array, for access
               'from listbox index.

            intSpotInTitleArray% = _
               intSpotInTitleArray% + 1
               'Increment array counter.
            strListBoxRow$ = snpBooks!Title & "   " _
               & snpBooks!price & _
               "   " & snpBooks!pubdate & "   "
               'Put title info together.
         End If
         If strAuthors$ <> "" Then strAuthors$ = _
            strAuthors$ & ", "
         'Add each author to authors list.
         strAuthors$ = strAuthors$ & _
            snpBooks!au_fname & " " & snpBooks!au_lname
         strCurrentTitle$ = snpBooks!title_id
            'Save current title before
            'switching records.
         snpBooks.MoveNext
      Wend
      If strListBoxRow$ <> "" Then
      'We still must add last item to list.
         strListBoxRow$ = strListBoxRow$ & strAuthors$
         frmBookList.lstBox.AddItem strListBoxRow$
      End If

      frmBookList.Show
      'Now show the form.
      snpBooks.Close
   End If
   dbPubs.Close
End Sub
```

```
dbPubs.Execute s$, dbSQLPassThrough
dbPubs.Close
```

It's true that you can change the add from opening a record set to simply creating an insert statement and using passthrough to get it to the server. But creating this stored procedure lets the server compile and optimize the SQL insert string in advance for greater speed.

Another feature that lets you play squarely in client/server territory, data caching, comes with DAO 3.0. Use data caching with dynaset-type Recordset objects, which generally take longer to read data than snapshot-type Recordset objects do. You may still want to use dynaset-type records in many common scenarios, though, due to their convenient updatability. To scroll through the data faster—especially when a form is using the dynaset—use the Recordset CacheSize and CacheStart properties and the FillCache method. In BookSale we use this code to open our form:

```
dsBooks.CacheSize = 20
dsBooks.CacheStart = dsBooks.Bookmark
dsBooks.FillCache
```

This preloads an internal buffer with 20 records' worth of data. As the user moves through those records from your app's UI, the apps display quickly because the app doesn't need to retrieve them from the server.

To use caching, specify the number of records to be stored by the value of CacheSize (a Long integer) and the beginning record by the bookmark stored as the value of CacheStart (a String variable). Apply the FillCache method to automatically retrieve every value in the cache range and fill the cache with server data. This method works faster than filling the cache because each record is fetched. If you know ahead of time that all records in the cache range will be visited, call FillCache every time you move CacheStart.

Fetches within the cache boundary occur locally, speeding display of the cached records in a data sheet or in a continuous form. CacheSize allowable values can range between five and 1200 records. If the cache size exceeds available memory, the excess records spill into a temporary disk file. You'll typically set CacheSize's value to 100. To recover the cache memory, set CacheSize to zero.

Once you set the CacheSize and CacheStart properties, as you move through records, the program will cache fetched data until you leave the defined range. Once you've hit the end of the range defined by CacheSize and CacheStart, move the CacheStart setting to a new position to stay synchronized with a particular set of records. Caching will continue with the new range, reusing values appropriately if the new cache range overlaps the old.

Watch performance jump when you specify a cache—especially if your app requires backward scrolling within the cached region. Depending on your scenario, using a cache may deliver more performance than using a read-only, forward-only snapshot, especially if the snapshot contains memo or long binary fields that get referenced only occasionally.

VB4

```
Sub LoadStores()
   Dim dbPubs As Database
   Dim snpStores As Recordset
   Dim intStoreArrayIndex%
   Dim strStoreInfo$
   Dim intFieldIndex%
   Dim snpStoresCount As Recordset
   Dim intRecordCount%
   Dim intRecordSetOptions%
   Dim strSelectStmt$

   intRecordSetOptions% = dbSQLPassThrough + _
      dbForwardOnly

   Set dbPubs = OpenDatabase(App.Path & _
      "\pubs.mdb", False, True)
   dbPubs.Connect = gCONNECT$

   Set snpStoresCount = _
      dbPubs.OpenRecordset("Select count(*) _
      from stores", dbOpenSnapshot, _
      intRecordSetOptions%)
   intRecordCount% = snpStoresCount(0)
   snpStoresCount.Close

   strSelectStmt$ = "Select stor_id, stor_name, _
      stor_address, "
   strSelectStmt$ = strSelectStmt$ & "city, _
      state, zip from stores "
   strSelectStmt$ = strSelectStmt$ & "order by _
      stor_name, State, Zip, "
   strSelectStmt$ = strSelectStmt$ & "city, _
      stor_address"

   Set snpStores = _
      dbPubs.OpenRecordset(strSelectStmt$, _
      dbOpenSnapshot, intRecordSetOptions%)

   ReDim gstrStoreArray(intRecordCount%)
   intStoreArrayIndex% = 0
   While Not snpStores.EOF
      gstrStoreArray(intStoreArrayIndex%) = _
         snpStores("stor_id")
      strStoreInfo$ = ""
      For intFieldIndex% = 1 To _
         snpStores.Fields.Count - 1
      'Start at the second field.
         If Not IsNull(snpStores_
            (intFieldIndex%)) And _
            snpStores(intFieldIndex%) <> "" Then
            strStoreInfo$ = strStoreInfo$ & _
               snpStores(intFieldIndex%) & " "
         End If
      Next intFieldIndex%
      frmWelcome!cboStoreName.AddItem _
         Trim$(strStoreInfo$)
      snpStores.MoveNext
      intStoreArrayIndex% = _
         intStoreArrayIndex% + 1
   Wend
   snpStores.Close
   dbPubs.Close

End Sub
```

LISTING 3 ***Tuning Queries.*** *LoadStores provides an example of how your program can first query the database to find out how many records to expect, then query the database to return the records. Always check the number of records being returned. If it exceeds what can be retrieved with acceptable performance, then ask users to limit the criteria of the records selected.*

The new Remote Data Access objects (RDO) in VB4 Enterprise Edition complements the client/server methods we've discussed so far. Unlike DAO/Jet version 3.0, RDO writes directly to ODBC, exploiting next-generation ODBC features such as bulk fetching of records. RDO also improves support for server features such as stored procedure parameters.

Use RDO's extra functionality if you're building complex client/server applications in VB4. But don't despair if you have a large existing code base written to DAO. The next version will bring RDO functionality and performance to all Microsoft apps using the same DAO object model you know and love.

If you take advantage of the techniques and new features we've discussed here, from linked tables to RDO, from code changes to new ways to interface with the

Dynasets vs. Snapshots

Both dynasets and snapshots have strengths you should know about. Of the two record sets, dynasets are trickier. The simplest dynaset-type record-set is a collection of *bookmarks* that lets you uniquely identify each record in a server database table. Each bookmark corresponds to one record on the server. Normally, the value of the bookmark corresponds to the primary key value for that record. For example, if you have an Orders table on the server database, with a primary key on its OrderID field, then internally to Microsoft Jet, the dynaset contains all the OrderID values corresponding to the records that satisfy the query.

When you access data in dynaset fields, Microsoft Jet uses the bookmark for the record to issue a query in the form "SELECT *field1*, *field2*,... FROM ORDERS WHERE ORDERID=*bookmark*." This statement is then sent to the server. (For best performance, Microsoft Jet actually includes up to 10 bookmarks in the Where clause.) Then, as you request the data on a field-by-field basis, Microsoft Jet calls the ODBC SQLGetData function to return the data from each field.

Due to this field-by-field behavior, data for a field is not retrieved from the server unless that field is explicitly retrieved by your application's code. For example, you would only retrieve the Photo field from the server if your code contained a line such as:

```
strNew = rstOrders!Photo
```

Thus if the table you're accessing has binary or memo fields with large amounts of data, Jet does not have to retrieve them from the server if you don't reference those large fields in your code.

Snapshots are simpler. They are a *complete copy* of all the requested fields in your query. As you move through a snapshot-type record set for the first time, all data is copied first into memory and then, if need be, into a temporary Jet database in the temporary directory on the user workstation. The resulting data set is read-only, and by default can be scrolled forward and backward.—*M.M. and E.K.*

VB4

```
Sub DataAlterTitles(dbPubs As Database, num&)
   Dim intRecordLoop& 'Loop control variable
   Dim strInsert$

   For intRecordLoop& = 0 To (num& Mod gMAXTITLES&)
      strInsert$ = "Insert into Titles values _
         ('" & gTITLESID$ & _
         Format$(intRecordLoop&,  "0000")
      strInsert$ = strInsert$ & "', '" & _
         gTITLESID$ & Format$(intRecordLoop&, _
         "0000")
      strInsert$ = strInsert$ & "', 'business', _
         '1389', 10, 100, 24, 100"
      strInsert$ = strInsert$ & ", 'Memo for " _
         & gTITLESID$ & Format$(intRecordLoop&, _
         "0000")
      strInsert$ = strInsert$ & "', '" & Now & "')"

      dbPubs.Execute strInsert$, dbSQLPassThrough
   Next intRecordLoop&
End Sub
```

LISTING 4 ***Passing Through.*** *This code lets you put together a SQL passthrough Insert statement so you can add records to the titles table in the pubs database. First you create the statement, then you execute it with the dbSQLPassthrough flag. In this case, we use a loop to be able to add more than one record. A stored procedure, as shown in Listing 1, would execute even faster—go to the extra effort of building one if it's a critical part of your app.*

server, you should see real client/server performance gains and better resource consumption. A little more coding effort—using this new way of thinking—can get you a long way.

We've talked about old and new versions of BookSale that embody these principles. If you'd like more detail, download both versions of the complete app. You can find them in the Magazine library of the *VBPJ* forum on CompuServe (GO WINDX), in the *VBPJ* library on The Microsoft Network (GO WINDX), and on *VBPJ's* Development Exchange on the World Wide Web (http://www.windx.com). Both versions are titled PUBS.VBP. The starting version is in the SHPSTART directory; the optimized version in the SHPEND directory. We've included a readme file which will tell you more of what the application does and how we put it together.

When you first run the app, use the Setup Database Button. This will ask you for information on the SQL server that the app will run against. You'll need SQL Server 6.0, including the sample pubs that ship with SQL Server. You'll also need 32-bit ODBC set up on your machine and a registered DSN pointing to your SQL Server.

We hope these sample apps will help you apply what we've discussed here. To keep things in the real world, we purposely limited the development time to emulate a typical client/server app rush job—like your own next project. ☒

VB4

User Tip

WHEN DBGRID REFRESHES INCORRECTLY

If you change a data control's RecordSource at run time to a table or query that shares field names with the previous RecordSource, a bound DBGrid will not refresh correctly; the common field names remain in their original positions rather than appearing in the order specified in the new table or query. If the common field names in the new RecordSource are capitalized differently than in the original, the DBGrid won't show any data in that column. To duplicate the problem, create a form containing a data control (datCtl), a DBGrid, and a command button (cmdButton). Set the DBGrid's DataSource property to the name of the data control. Add this code:

```
Private Sub Form_Load()
   datCtl.DatabaseName = _
      "biblio.mdb"
   datCtl.RecordSource = "SELECT _
      Au_ID, Author FROM Authors;"
End Sub

Private Sub cmdButton_Click()
   ' Reverse field order - DBGrid will
   ' continue to display Au_ID first
   datCtl.RecordSource = "SELECT _
      Author, Au_ID FROM Authors;"
   datCtl.Refresh

   ' If you set the RecordSource to
   ' this (note capitalization)...
   '
   ' "SELECT Author, au_ID FROM
   ' Authors;"
   '
   ' the DBGrid won't display any data
   ' in the second column

End Sub
```

The workaround is to "reset" the DBGrid by pointing it at a recordset that shares no common field names with the current recordset. For example, change the cmdButton_Click event to:

```
Private Sub cmdButton_Click()
   Dim SQL As String
   ' Reset DBGrid by creating empty
   ' recordset with unique field names

   SQL = "SELECT Author AS [Unique _
      Field Name] FROM Authors _
      WHERE False;"
   datCtl.RecordSource = SQL
   datCtl.Refresh

   ' DBGrid will now display fields
   ' as listed
   datCtl.RecordSource = "SELECT _
      Author, Au_ID FROM Authors;"
   datCtl.Refresh
End Sub
```

—Phil Weber, *VBPJ* Technical Review Board

SEND YOUR TIP

If it's cool and we publish it, we'll pay you $25. If it includes code, limit code length to 10 lines if possible. Be sure to include a clear explanation of what it does and why it is useful. Send to 74774.305@compuserve.com or Fawcette Technical Publications, 209 Hamilton Ave., Palo Alto, CA, USA, 94301-2500.