Click & Retrieve Source CODE!

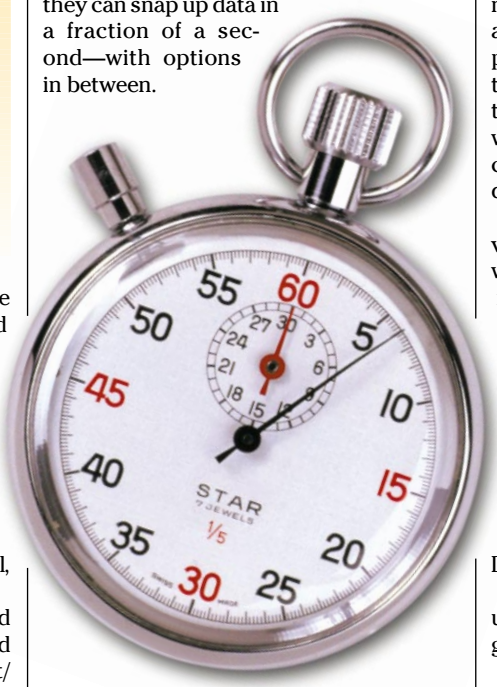
Clocking Data Access

BY STEVE JACKSON

Query execution can vary by a ratio of 24:1 depending on the data access technique. Study these benchmarks to determine the right approach for you.

Just as VB developers can choose between different versions of VB and different operating systems, those of us doing client/server database development must choose between several data access techniques, including Jet options, direct ODBC API calls, and most recently RDO. VB4 comes with a new Jet engine and Data Access Object (DAO) model, 16-bit and 32-bit versions, and the new Remote Data Object (RDO) model, available only in 32-bit mode.

As part of a team assigned to define and execute benchmark tests, my task entailed defining and performing a suite of client/server data-access benchmarks to complement Ward Hitt's feature on VB speed tests (see "Benchmark Battle," this issue). During the definition phase, I worked closely with *VBPJ* database columnist and VBITS speaker Andrew Brust. While conducting tests, I consulted with colleagues at Loral, as well as *VBPJ* staff and contributing editors. I learned some interesting optimization tricks and discovered a 24-to-1 performance ratio between the fastest and slowest data access techniques. Users can wait nearly half a minute for query results, or they can snap up data in a fraction of a second—with options in between.



Maximizing data access speed depends for the most part on your choice of techniques. I'll walk you through the options based on my test results.

Speed is the key consideration in making the right data access choice for client/server database design, although other criteria, such as ease of use, maintainability, and portability must be weighed. In my company, database performance is foremost in every developer's mind. If an application is too slow, no one will use it no matter how inspired the user interface or how clever the code. I wrote a database-performance test program and ran it on several different platforms to obtain hard data on the fastest back-end techniques.

Here at Loral, my results have started several lively debates on the benefits and trade-offs of different programming techniques. Applying some of the lessons to an actual application at Loral resulted in a 33 percent reduction in the overall startup time by improving the speed of seven database queries. Developers here are still weighing the alternatives, but one thing is certain: everyone is looking closely at their database access code.

My testing criteria included trade-offs in versions of VB, operating systems, hardware configuration, and data access techniques. I wanted answers to a variety of questions, including: How is client/server data-access performance affected when moving from Windows 3.11 to Windows 95, or to Windows NT? What is the impact of the machine processor and RAM? What are the performance differences between VB3, VB4/16, and VB4/32? How do different programming techniques compare, and how does the new Remote Data Object (RDO) perform?

The test program timed a series of tests using these platforms and database programming techniques:

- Jet DAO snapshots and dynasets.
- Jet DAO using an MDB attached to the remote database server tables.
- DAO using SQL passthrough to bypass the Jet engine.
- Jet DAO QueryDefs.
- ODBC API calling ODBC directly.
- Remote Data Object (32-bit only).
- VB3, VB4/16, and VB4/32.
- Windows 3.11, Windows 95, and Windows NT 3.51.
- 486/66 with 8 MB RAM, Pentium/100 with 32 MB RAM.

Steve Jackson develops network apps using VB, SQL Server, and other tools at Loral Aeronutronic in Southern California. Steve is a VBPJ *author and CompuServe section leader. Pradeep Shah and other colleagues at Loral contributed to this article. Reach Steve on CompuServe at 72040,1640.*

I restricted tests to remote SQL database servers using SQL Server and Oracle, and did not run any tests on file-based databases, such as Access, Paradox, or FoxPro.

I tested client/server performance because in our corporate environment (and in most large organizations), we need the recovery and integrity features built into a true database server. What is optimal coding for an Access database is not always the best technique for a remote database server.

The test program was compiled in VB3, VB4/16, and VB4/32, with only slight modifications between VB3 and VB4 to use the new VB4 Jet DAO syntax. The VB4 16-bit and VB4 32-bit code was identical except for some API call declarations. RDO was tested only in the VB4/32 test program because it's available only in VB4/32 Enterprise Edition.

To compare the effects of processor speed and RAM, the Windows 95 and Win 3.11 tests were performed on a Pentium/100 processor and 32 MB of RAM and on a 486/66 processor and 8 MB of RAM. The NT machines had slightly different configurations due to hardware availability and software requirements.

For the back end I used Microsoft SQL Server version 6.0 running on a Pentium/100 NT server, and all the client machines ran Microsoft TCP/IP stacks and ODBC drivers. I could not run the 32-bit test on the 486/66 with 8 MB of RAM because SQL Server ODBC drivers have a stated RAM requirement of 16 MB.

A small set of the tests was also run using an Oracle server and a Pentium client. The Oracle results should not be compared to SQL Server because the server hardware, network, and machine load were different—the test was meant to show the relative differences between programming techniques.

Of course your database applications will not perform exactly as the test program performed, but these tests are valuable for comparing performance of various data-access techniques, configurations, and platforms.

You can download the test suite from *VBPJ* sites on CompuServe, MSN, and the Web (see "How To Reach Us" in the Letters section, or the instructions opposite the Table of Contents for details). Search for DBBENCH.ZIP. Eric Busby of QuickStart Technologies duplicated my tests using comparable platforms (see the sidebar, "Duplicating Data Access Benchmarks").

	Jet DAO	Jet Attached MDB	DAO SQL Passthru	Jet MDB Query	ODBC API	RDO 32-bit only	RDO Prepared Stmt
Select rows by indexed key 20 times							
VB3	12.36	3.07	5.90	3.89	0.28	n/a	n/a
VB4/16	12.40	3.22	6.00	3.30	0.26	n/a	n/a
VB4/32	13.78	3.89	5.98	3.85	0.38	1.74	1.25
Select all rows small table							
VB3	0.55	0.25	0.43	0.26	0.10	n/a	n/a
VB4/16	0.59	0.29	0.45	0.29	0.09	n/a	n/a
VB4/32	0.71	0.30	0.48	0.42	0.13	0.33	0.34
Search on nonindexed value							
VB3	0.95	0.49	0.24	0.52	0.02	n/a	n/a
VB4/16	0.95	0.49	0.31	0.54	0.02	n/a	n/a
VB4/32	1.23	0.52	0.29	0.78	0.02	0.10	0.12
Select and update 20 rows							
VB3	14.91	5.67	5.97	6.52	.067	n/a	n/a
VB4/16	14.17	4.80	5.99	4.91	.056	n/a	n/a
VB4/32	14.74	5.50	6.49	5.96	.074	2.24	1.78
Insert 20 rows							
VB3	2.63	2.64	0.29	3.65	0.29	n/a	n/a
VB4/16	1.54	1.60	1.09	3.35	0.29	n/a	n/a
VB4/32	1.53	1.50	1.13	3.47	0.29	1.19	0.40
Delete 20 rows							
VB3	14.94	5.67	0.27	5.67	0.28	n/a	n/a
VB4/16	15.07	5.76	1.08	5.78	0.27	n/a	n/a
VB4-32	16.48	7.90	1.10	8.08	0.26	1.16	0.29
Two-table join							
VB3	3.50	2.90	0.28	2.86	0.06	n/a	n/a
VB4/16	3.31	2.91	0.30	2.84	0.09	n/a	n/a
VB4/32	3.62	3.31	0.35	2.80	0.07	0.17	0.18
TOTAL TIME							
VB3	54.3	22.1	15.5	25.0	2.2	n/a	n/a
VB4/16	52.7	20.4	17.0	22.7	2.1	n/a	n/a
VB4/32	56.2	24.6	17.7	29.3	2.4	9.3	5.5

TABLE 1 ***Testing Under Windows 95 on a Pentium.*** *These database access test results were obtained on a client Pentium/100 with 32 MB of RAM running Windows 95, and a SQL Server 6.0 back-end database server running on Windows NT 3.51. There was a 24-to-1 difference between the different programming techniques, with direct ODBC API calls screamingly fast, and 32-bit RDO close behind. For a given programming technique, VB4/16 and VB4/32 were both slightly slower than VB3. The programming technique made more of a difference than versions of VB used. Timings are in seconds.*

THE TEST DESIGN

I tested a series of simple SQL operations that included queries based on an indexed key, search queries based on a nonindexed column, updates, inserts, deletes, and a two-way table join. These tests simulate a transaction-oriented system such as an order entry application, where typical operations might include retrieval of customer data by an indexed key, adding new rows with order data, and updating the customer data and inventory data. Longer-running queries would be possible if a search were needed on a nonindexed column—for example, retrieving all products with a particular product code. I also included a query to retrieve all the rows in a small 200-row table, like one that might load into a memory array once at the start of a program and be used for code validation.

For the SQL tests I created the tables Customer, ProductCode, and OrderEntry

	Jet DAO	Jet Attached MDB	DAO SQL Passthru	Jet MDB Query	ODBC API
Select rows by indexed key 20 times					
VB3	12.70	4.66	5.60	4.77	0.38
VB4/16	13.77	3.79	5.88	4.99	0.39
Select all rows small table					
VB3	0.61	0.33	0.55	0.49	0.11
VB4/16	0.66	0.38	0.49	0.38	0.11
Search on nonindexed value					
VB3	0.88	0.82	0.27	0.82	0.05
VB4/16	1.04	0.93	0.32	0.93	0.01
Select and update 20 rows					
VB3	14.80	6.43	7.14	7.19	0.77
VB4/16	14.16	6.81	6.92	6.76	0.77
Insert 20 rows					
VB3	2.63	2.30	0.66	3.35	0.33
VB4/16	1.10	1.32	1.10	3.34	0.33
Delete 20 rows					
VB3	17.31	7.30	0.49	7.69	0.38
VB4/16	17.36	7.42	1.10	7.84	0.38
Two-table join					
VB3	3.62	3.13	0.28	2.86	0.06
VB4/16	3.74	3.30	0.31	2.91	0.08
TOTAL TIME					
VB3	57.2	27.1	17.1	28.8	2.2
VB4/16	56.3	25.2	17.7	28.1	2.4

TABLE 2 ***Moving to Win 3.11 on the Pentium.*** *The tests run on Windows for Workgroups 3.11 and a Pentium/100 with 32 MB of RAM show VB3 times slightly slower than Windows 95 times, although VB4/16's times were very close to Windows 95's times. Like the Windows 95 test, the big difference was between programming techniques, with the ODBC API 24 times faster than using Jet/DAO. No 32-bit tests were run because Windows 3.11 is 16-bit.*

on SQL Server, and wrote a VB program to populate them with test data. The Customer table was loaded with 101,000 rows of data created at random by the load program. The ProductCode table was loaded with 200 rows, and the OrderEntry table was loaded with 20,000 rows.

The larger Customer table was used for most of the tests, the smaller ProductCode table was used for a test that read all the rows in the 200-row table, and Customer and OrderEntry tables were used for a two-table join.

I called the Windows API call GetTickCount() to time the tests before and after operations by getting the number of milliseconds since the system was started. The VB timer control uses this same internal timer. GetTickCount() is accurate only to about 55 milliseconds because it's based on a system timer that fires 18.2 times a second. However, Windows has a multimedia timer that fires more frequently and allows finer measurements.

Early in the coding of the test program I used the multimedia timer API, but I experienced random variations in network throughput and server response, so it was difficult to get a consistent measure of an individual database call down to the millisecond.

VB4

```
'In the declarations section
Dim henv As Long
Dim hdbc As Long
Global Const SQ = "'"   '1 single quote

Function ODBCGetRowByKey(sKey As String)
As String
Dim sSQL As String, hstmt As Long
Dim rc As Integer, lValueLen As Long
Dim sLastName As String
Dim sFirstName As String
lValueLen = 100
sLastName = String$(100, Chr$(0))
sFirstName = String$(100, Chr$(0))
sSQL = "Select LastName, FirstName from Customer _
  where CustID = " & SQ & sKey & SQ
rc = SQLAllocStmt (hdbc, hstmt)
' SUB ShowStatus() writes to log file
If rc <> SQL_SUCCESS Then
  ShowStatus "Error in AllocStmt " & Str$(rc)
  Exit Function
End If
'run the query
rc = SQLExecDirect(hstmt, sSQL, Len(sSQL))
If rc <> SQL_SUCCESS Then
  ShowStatus "Error in ExecDirect " & Str$(rc)
  rc = SQLFreeStmt(hstmt, SQL_DROP)
  Exit Function
End If
' get a row
rc = SQLFetch(hstmt)
If rc <> SQL_SUCCESS Then
  sLastName = "ERROR - NOT FOUND!"
  sFirstName = ""
Else
  ' get a data value
  rc = SQLGetData(hstmt, 1, SQL_C_CHAR, sLastName, _
    100, lValueLen)
  rc = SQLGetData(hstmt, 2, SQL_C_CHAR, sFirstName, _
    100, lValueLen)
  If rc <> SQL_SUCCESS Then
    sLastName = "ERROR - NOT FOUND!"
    sFirstName = ""
  End If
End If
'SUB TrimNulls() strips C style terminating nulls from
'the string
Call TrimNulls(sLastName)
Call TrimNulls(sFirstName)
'De-allocate statement
rc = SQLFreeStmt(hstmt, SQL_DROP)
ODBCGetRowByKey = RTrim$(sLastName) & ", " & _
  RTrim$(sFirstName)
End Function
```

LISTING 1 ***The ODBC API Trade-Off.*** *ODBC API code to retrieve a row is much longer than simply opening a dynaset or snapshot. The ODBC API is blazing fast, but requires more work to code. There is no trappable error handling, so each API call must be checked for errors, and the API does not support bound controls. Changing the column definitions in the table would require modifications to the code.*

I went back to the the GetTickCount() API call for ease of use between 16-bit and 32-bit platforms, and decided instead to measure a series of database calls and measure the speed of the entire series.

To accurately measure the faster operations, I generated arrays of random keys for select, insert, and delete tests before the code that started the timer, then started the timer, executed the operation 20 times using the array of random keys, and ended the timer count. The elapsed seconds were then computed as (EndTime - StartTime)/1000, and written to a log file. I ran the tests several times on each machine during weekend hours when there was little network and database server traffic,

	Jet DAO	Jet Attached MDB	DAO SQL Passthru	Jet MDB Query	ODBC API	RDO
Select rows by indexed key 20 times						
VB3	16.51	4.93	6.53	6.46	0.51	n/a
VB4/16	16.03	4.59	6.91	6.16	0.50	n/a
VB4/32	12.6	4.16	3.04	4.35	0.49	1.09
Select all rows small table						
VB3	0.77	0.32	0.51	0.38	0.13	n/a
VB4/16	0.78	0.30	0.57	0.41	0.11	n/a
VB4/32	0.61	0.37	0.43	0.29	0.12	0.38
Search on nonindexed value						
VB3	2.05	1.47	0.45	0.83	0.06	n/a
VB4/16	2.15	1.25	0.51	0.79	0.06	n/a
VB4/32	0.95	0.77	0.55	0.48	0.02	0.11
Select and update 20 rows						
VB3	21.18	10.82	8.64	10.75	1.02	n/a
VB4/16	20.70	10.47	8.93	10.58	0.96	n/a
VB4/32	20.28	5.76	3.96	5.95	0.90	1.46
Insert 20 rows						
VB3	4.10	3.07	0.45	4.10	0.51	n/a
VB4/16	4.05	2.89	0.51	3.98	0.47	n/a
VB4/32	1.43	1.61	1.31	3.03	0.25	0.29
Delete 20 rows						
VB3	24.45	9.79	0.32	11.39	0.38	n/a
VB4/16	25.21	9.26	0.26	10.89	0.39	n/a
VB4/32	15.71	7.61	1.20	8.42	0.23	0.29
Two-table join						
VB3	5.50	3.14	0.51	3.78	0.06	n/a
VB4/16	5.36	3.05	0.58	3.44	0.07	n/a
VB4/32	3.63	2.75	0.18	3.19	0.13	0.19
TOTAL TIME						
VB3	81.22	35.97	21.36	40.13	3.39	n/a
VB4/16	83.14	36.72	20.68	41.34	3.51	n/a
VB4/32	55.10	25.92	12.87	29.48	2.46	5.41

TABLE 3 ***Evaluating NT Pentium Performance.*** *Test results for a Pentium/100 with 24 MB of RAM running Windows NT Workstation 3.51 favor 32-bit apps. The big suprise here was the wide difference between 16-bit and 32-bit tests, with 16-bit results as much as 50 percent slower than 32-bit. Windows NT has to convert 16-bit operations to 32-bit, slowing down the 16-bit code.*

so results weren't influenced by heavy network traffic.

THE ENVELOPE, PLEASE

The ODBC API is the fastest data access technique, turning in a total test time of 2.2 seconds using 16-bit VB4 on a Pentium/100 running Windows 95 (see Table 1). When evaluating all results, I was surprised by the wide difference in data access speed.

For example, the ODBC API result of 2.2 seconds was more than 24 times faster than the VB4/16 time of 52.7 seconds using Jet Data Access Objects. VB4/16 was slightly faster than VB3 on the Pentium, but only by 3 percent (Jet/DAO) to 5 percent (Attached MDB), and was 9 percent slower using the SQL passthrough option. VB4/32 was somewhat slower than 16-bit VB4 and VB3, but at 5.5 seconds, 32-bit RDO was faster than any other test except for the ODBC API. Clearly, the choice of

	Jet DAO	Jet Attached MDB	DAO SQL Passthru	Jet MDB Query	ODBC API
Select rows by indexed key 20 times					
VB3	19.11	7.84	0.92	9.11	0.50
VB4/16	25.81	6.29	1.16	5.58	0.61
Select all rows small table					
VB3	1.06	0.68	0.71	0.71	0.21
VB4/16	1.12	0.90	0.77	0.84	0.18
Search on nonindexed value					
VB3	1.12	0.58	0.10	0.90	0.03
VB4/16	1.35	0.54	0.13	0.72	0.04
Select and update 20 rows					
VB3	26.01	9.42	2.21	11.95	0.79
VB4/16	25.21	10.18	2.04	10.01	0.91
Insert 20 rows					
VB3	2.6	2.99	0.58	6.23	0.50
VB4/16	2.0	2.12	1.52	7.50	0.52
Delete 20 rows					
VB3	27.34	12.78	0.49	12.64	0.32
VB4/16	29.59	14.66	1.19	13.01	0.45
Two-table join					
VB3	5.0	4.0	0.81	1.23	0.21
VB4/16	6.3	4.8	0.91	0.78	0.26
TOTAL TIME					
VB3	86.9	34.3	5.3	43.5	2.9
VB4/16	94.7	35.5	7.3	40.6	3.1

TABLE 4 ***Working with Windows 95 on a 486.*** *A 486/66 with 8 MB of RAM running Windows 95 showed an even wider perforamnce range between programming techniques. The ODBC API was 30 times faster than Jet/DAO. Due to the extra overhead required, Jet suffered from the slower processor and lack of RAM. Note that the ODBC API total time of 2.9 seconds was faster than any other technique on the Pentium, including RDO.*

	Jet DAO	Jet Attached MDB	DAO SQL Passthru	Jet MDB Query	ODBC API
Select rows by indexed key 20 times					
VB3	19.94	8.13	1.21	9.67	0.61
VB4/16	26.03	6.76	1.76	6.54	0.60
Select all rows small table					
VB3	1.26	0.77	0.71	0.82	0.27
VB4/16	1.43	0.93	0.77	0.99	0.17
Search on nonindexed value					
VB3	1.32	0.71	0.11	0.99	0.05
VB4/16	1.48	0.66	0.16	0.77	0.06
Select and update 20 rows					
VB3	26.25	9.72	2.47	13.57	1.21
VB4/16	25.82	10.49	2.25	10.55	1.15
Insert 20 rows					
VB3	2.80	3.02	0.55	7.63	0.55
VB4/16	2.2	2.25	1.21	8.51	0.55
Delete 20 rows					
VB3	27.74	13.01	0.49	13.46	0.38
VB4/16	29.77	14.46	1.21	13.89	0.55
Two-table join					
VB3	5.1	4.2	0.81	1.46	0.26
VB4/16	6.3	5.1	1.12	0.90	0.28
TOTAL TIME					
VB3	87.3	35.4	5.5	46.1	3.0
VB4/16	95.1	35.6	7.4	41.2	3.0

TABLE 5 ***How About 3.11 on the 486?*** *Test results on a 486/66 with 8 MB of RAM running Windows for Workgroups 3.11 were close to those for Windows 95 on the 486. VB4 was slower than VB3 in most tests. The pattern of differences between programming techniques was similar to that of other platforms.*

programming technique has a greater effect on performance than the version of VB used.

The results for a Pentium/100 running on Windows 3.1 show that VB3 programs run faster on Win95 (54.3 seconds) than Windows 3.1 (57.2 seconds), and VB4 programs run slightly faster on Windows 95 (see Table 2).

The pattern between the different programming techniques shows ODBC API results 24 times faster than the Jet/DAO. Tests run on Windows NT show a big difference between 16-bit and 32-bit programs, with 32-bit results roughly comparable to Windows 95, but 16-bit results as much as 50 percent slower (see Table 3). There's no 16-bit code in NT, so 32-bit programs have a clear advantage.

I ran tests on a 486/66 using the same operating systems. Again, the ODBC API on Windows 95 is the clear winner, with total test times of 2.9 seconds (VB3) and 3.1 seconds (VB4/16), which are 30 times faster than the Jet/DAO times of 86.9 and 94.7 on the same machine (see Table 4).

Like the Pentium results, the programs ran somewhat slower on Windows 3.1 (see Table 5). Running NT tests on the 486, the 32-bit tests proved superior to the 16-bit tests (see Table 6). Notably, ODBC runs faster on the 486 than other techniques running on the Pentium, with the fastest 16-bit 486/66 ODBC API results of 2.9 seconds faster than RDO running on a Pentium/100 at 5.5 seconds. There were a few anomalies in the SQL passthrough and NT results I'll explore in detail later.

I reran and rechecked the ODBC API results several times, and performed separate database queries to ensure it was indeed running correctly, and that the inserts, updates, and deletes were truly taking place. As I stated earlier, these results may not match your experience exactly, because the server and network were under light usage, and the SQL operations were fairly simple. However, I drew a number of conclusions that I think are valid, based on the data.

The easiest way to improve DAO performance is to use an attached MDB: attach an Access database with no data in it to the ODBC data source so that only tables are attached. This is fairly common knowledge among database developers, and my results confirm it. When connected directly to an ODBC data source using the OpenDatabase statement with an ODBC connect string, Jet makes many additional calls to the remote database to gather information about the tables, rows, and

columns with which you are working. When the tables are attached to a Jet MDB database, Jet retrieves this information once and keeps it in the MDB where Jet accesses it at run time. For 32-bit programs, the attached MDB must be in the new Jet 3.0 format used by 32-bit Access for Windows 95.

Using this simple code, I was able to create a 32-bit attached MDB even without a copy of 32-bit Access:

```
Dim ws As Workspace
Dim db As Database
Dim td As TableDef
Dim sConn As String
Set ws = DBEngine.Workspaces(0)
'Insert your own valid data source
'name and user information below
sConn = _
   "ODBC;DSN=SqlServer32;_
   UID=sjackson;PWD=****;"
Set db = _
   ws.CreateDatabase("DBTEST32.MDB",_
   dbLangGeneral)
Set td = db.CreateTableDef("Customer")
td.Connect = sConn
td.SourceTableName = "Customer"
db.TableDefs.Append td
db.Close
```

SQL passthrough significantly improved performance against a remote database server, but required more work to code. To create dynasets and snapshots this way, I created a SQL SELECT statement dynamically, and called CreateDynaset or CreateSnapshot with the SQL passthrough option.

This technique bypasses the Jet SQL parser and sends the command directly to the back-end server, reducing Jet overhead. However, you must create long SQL statement strings with the exact SQL syntax required by your particular database server. For inserts, updates, and deletes, I created dynamic SQL and executed the SQL statement with the Execute method of the database object.

Some anomalies occurred in the SQL passthrough test. The tests actually ran much faster on the 486/66 than on the Pentium/100, using all versions of VB. Also, there was a large difference between VB3 and VB4 in the insert and delete tests on the Pentium. Because the other tests on the Pentium ran much faster than on the 486, I looked for any differences that could affect SQL passthrough performance, but found none.

I sent the SQL passthrough code to some Microsoft gurus, but they did not have a solution and did not make any recommendations. Because this affected performance using only SQL passthrough and not the other techniques, I am assuming it is not due to the network, and that it is unique to this client computer. Notably, SQL passthrough and DAO with attached tables performed better than Access QueryDefs. QueryDefs may improve performance for Access databases with complex SQL statements because Access precompiles and optimizes each query. Using a remote database server improves performance because the server optimizes the queries and updates and skips the QueryDef overhead.

	Jet DAO	Jet Attached MDB	DAO SQL Passthru	Jet MDB Query	ODBC API	RDO 32-Bit Only
Select rows by indexed key 20 times						
VB3	13.76	5.44	7.55	10.62	0.45	n/a
VB4/16	13.44	5.27	7.89	10.20	0.43	n/a
VB4/32	7.2	2.96	2.64	3.02	0.51	1.30
Select all rows small table						
VB3	1.02	0.51	0.70	0.64	0.13	n/a
VB4/16	0.93	0.49	0.81	0.73	0.12	n/a
VB4/32	0.76	0.53	0.44	0.48	0.21	0.43
Search on nonindexed value						
VB3	1.02	0.57	0.32	0.77	0.01	n/a
VB4/16	1.12	0.51	0.40	0.73	0.01	n/a
VB4/32	0.50	0.24	0.14	0.21	0.03	0.19
Select and update 20 rows						
VB3	16.77	9.15	8.38	10.62	0.77	n/a
VB4/16	16.25	8.99	8.41	10.05	0.73	n/a
VB4/32	8.54	4.27	2.92	4.2	0.95	1.78
Insert 20 rows						
VB3	2.62	2.69	0.32	6.53	0.38	n/a
VB4/16	2.11	2.54	0.38	6.29	0.36	n/a
VB4/32	0.87	1.16	0.34	3.2	0.34	0.43
Delete 20 rows						
VB3	19.71	9.98	0.26	10.24	0.44	n/a
VB4/16	19.37	10.02	0.26	9.89	0.42	n/a
VB4/32	8.96	5.29	0.32	6.43	0.28	0.37
TOTAL TIME						
VB3	64.6	35.3	21.3	46.7	3.3	n/a
VB4/16	66.1	34.7	21.9	48.2	4.1	n/a
VB4/32	33.2	18.5	8.6	23.7	3.9	6.4

TABLE 6 ***Tuning a 486 for NT.*** *These tests used a 486/66 with 16 MB of RAM running Windows NT 3.51. Unlike under Windows 95, 16-bit code was much slower than 32-bit code, sometimes more than twice as slow. This machine was highly tuned and optimized—it clocked some tests faster than the Pentium—which shows the effect that hardware differences can have on your application's performance.*

BLAZING-FAST ODBC API

The ODBC results are worth a closer look because they won so handily. The ODBC API blew the doors off all other data access techniques—touching hundreds of different rows in 2.3 seconds on the Pentium client. These tests show conclusively that the ODBC API is lightning quick. To put ODBC results in perspective, consider that the total VB3 ODBC API results of 2.9 seconds on a 486/66 with 8 MB of RAM under Windows 95 were faster than any other test (excluding ODBC) running on a Pentium/100—including 32-bit RDO. Does this mean you should convert your projects to the ODBC API? You need to weigh the pros and cons to make that decision.

Blazing speed comes at a price (can you spell GPF?). The speed of the ODBC API is due in part to bypassing Jet and DAO

completely (gaining about 2 MB of memory used by Jet and DAO). Moving to the ODBC API means you give up ease of programming and maintainability offered by DAO. The API is more difficult to code and less maintainable than DAO code.

I created ODBC test code for getting a row, adding 1 to the order count, and updating the row (see Listing 1). It has hard-coded column widths and column positions. Changes to these aspects of the table require changes to the code or additional ODBC API calls to get this information, which would diminish the performance advantage. Any Windows API call requires care with memory locations and error checking to avoid crashing the application, and perhaps the entire computer.

THE CASE FOR RDO

However, you can get close to ODBC API speed without the pain and loss of features. RDO shows promise as a compromise between the ODBC API and Jet techniques for client/server database access (for background on RDO, see "A Walking Tour of RDO," *VBPJ* March 1996). RDO was much faster than Jet and DAO, and easier to code than the ODBC API.

Unlike the ODBC API, RDO supports data-bound controls. Essentially a wrapper around the ODBC API, RDO exposes certain ODBC statement and environment handles that allow a programmer to combine ODBC API calls with RDO.

To add and change data with RDO, you open a result set on the table and use AddNew, Edit, Update, and Delete methods; create dynamic SQL and execute it with the Execute method; or use prepared statements. My tests used dynamic SQL and RDO prepared-statement techniques, both of which are faster than result-set methods (see Listing 2).

Prepared statements are extremely

	Jet DAO	Jet Attached MDB	DAO SQL Passthru	Jet MDB Query	ODBC API	RDO 32-Bit Only
VB3	227.7	32.1	14.5	37.1	12.2	n/a
VB4/16	240.6	30.4	13.7	37.9	12.7	n/a

TABLE 7 ***Oracle Data Access.*** *Test results using a Pentium/100 client and an Oracle 7 back end server running Unix reveals performance similar to the overall test trends—ODBC API wins. These results don't directly compare to SQL Server.*

Duplicating Data Access Benchmarks

To duplicate Steve Jackson's data access bench marks, I created a lab environment to provide objective validation of his results. Knowing that Steve performed the tests on a production network during off-peak hours, I was concerned that the results may not be accurate due to minimal variable network traffic.

I set up a standalone network to guarantee that the results weren't skewed. I used similar machines with the same network protocol (TCP/IP) and operating systems as the original tests. The only known difference between his test environment and mine was the slower processor speed of my Pentium/90, compared to his Pentium/100. I had to go with the 90-MHz machine because I didn't have access to a 100-MHz system.

The results of my tests were consistent across the board with Steve's. There were no serious discrepancies in my findings. After running the battery of tests three times and comparing them to the original numbers in a spreadsheet, I found the accuracy of the results to be plus or minus 4 percent overall compared with Steve's findings.

Each battery of tests executed test code that logged results three times for each of the specified procedures. In other words, I tested each operation a total of nine times and found very little variance. The exception was the performance of the 486/66 with 16 MB of RAM running NT.

The machine I used did not seem to perform quite as well as Steve's, but it produced results with consistent ratios relative to the various programming techniques. I think the performance difference may have to do with Steve's 486 being tuned for his LAN.

The great unsolved mystery of this entire project remains the disparity between DAO SQL Passthrough performance on the 486 versus the Pentium. My DAO SQL passthrough tests duplicated Steve's within 4 percent, confirming that this technique is five to six times faster on the 486 than on the Pentium. Developers at several organizations, including Microsoft, studied this quirk and found no reason for the difference. A *VBPJ* editor contacted Intel about it, but as of press time had no explanation. I'll continue researching this anomaly.

A few days after I ran these tests, I was discussing my benchmarks with a colleague, who mentioned a remarkably similar set of tests he ran last month. He created a test app to hit a SQL database with ODBC API, RDO, and DAO techniques.

He too found the ODBC API to be the fastest, but he pointed out that once the additional development time required to make API calls was considered, RDO emerges as the favorite for 32-bit work. As our discussion continued, he boasted that with detailed optimization, a 486 Windows 95 system could outperform a Pentium NT.

This raised several issues in my mind that I don't have space to discuss here, but the bottom line is that the degree of operating system optimization has an effect on the performance of a database application.

We concluded that all environments mentioned here were typical installations, and that operating-system optimization was not key to the general issues. Nonetheless my awareness was raised regarding another variable.

ODBC API calls obviously log the fastest times. But with its ease of use and bound controls, RDO offers performance that's almost as fast with less pain and risk for 32-bit development.

As for the performance differences between VB3 and VB4, take a close look at the numbers. They indicate that there is little difference between the products for data access, unless you plan to implement 32-bit RDOs.

Hardware is always near and dear to a developer's heart. After all, if the code is running a little slow, you can cover a multitude of mistakes by suggesting that the user community needs more horsepower. If this has been your strategy in the past, be careful. The results show that better hardware isn't necessarily going to help. For example, take a look at the speed of the 486/66 using the ODBC API versus the Pentium using any of the other techniques. ODBC API calls on the 486 win hands down over Jet, DAO, or RDO approaches on the Pentium.

Contact me on CompuServe at 74437,77 or at ericb@quickstart.com if you have any questions about my test results.—*Eric Busby*

VB4

```
Function RdoUpdateRow(sKey As String) As Long
  Dim iTimesOrderPlaced As Integer, sSQL as string
  sSQL = "Select LastOrderDate, TimesOrderPlaced from Customer"
  sSQL = sSQL & " where CustID = " & SQ & sKey & SQ
  ' assume the RDO connection has already been made
  Set rdoRes = rdoConn.OpenResultset(sSQL, rdOpenKeyset, rdConcurRowver)
  If rdoRes.EOF Then
    rdoRes.Close
    RdoUpdateRow = 0
  End If
  iTimesOrderPlaced = rdoRes("TimesOrderPlaced") + 1
  rdoRes.Edit
  rdoRes("TimesOrderPlaced") = iTimesOrderPlaced
  rdoRes.Update
  rdoRes.Close
  RdoUpdateRow = 1
End Function
```

LISTING 2 ***RDO: Speed and Ease.*** *This 32-bit RDO code retrieves a row into a resultset, changes the data and updates it. RDO is a wrapper around the ODBC API, and almost as fast, but much easier to code. This example shows an update with dynamic SQL—it does more than the ODBC API example, with less code. RDO Prepared Statements require a little more coding than this, but run faster for repeated SQL operations.*

useful for operations that will be repeated. A prepared statement sends the SQL string to the server only once when it is initially created. The server parses the SQL and compiles an execution plan once.

Then when a result set is built from the prepared statement, only parameters are passed to the server, and the server can execute the compiled SQL plan without the extra overhead of parsing, optimizing, and compiling the SQL statement.

Using prepared statements dropped the total RDO test time from 9.3 seconds to 5.3 seconds in my tests on the Pentium. This technique is a wrapper around the equivalent ODBC API calls that create and execute prepared statements.

In my ODBC API tests, I used only the dynamic SQL option, which was still faster than RDO prepared statements. A truer comparison would also use ODBC API prepared statements, but this requires complex VB code to bind parameters to fixed-string addresses in memory, and was beyond the scope of this project.

OTHER LESSONS LEARNED

Identical 16-bit test programs generally ran slightly faster in Windows 95 than in Windows for Workgroups 3.11. This could be the result of improved Windows 95 internals and memory management, a faster TCP/IP stack, or both.

Generally, Win16 apps run faster on Windows 95 due to improved caching, more efficient code, and protected-mode drivers. The improvement is about 10 percent, which is beneficial, but won't speed a sluggish app.

In Windows 95 tests, the 32-bit test program ran slightly slower than the 16-bit version, but the real shocker was the test on Windows NT (see Tables 3 and 6). Unlike Windows 95, the 32-bit test program ran much faster on Windows NT than the 16-bit programs. The difference was pronounced for techniques with large overhead and many DLLs, such as Jet DAO, and less pronounced for lower overhead techniques, such as the ODBC API.

Due to availability limitations, the Windows NT tests were run on different machines than the Windows 95 and Windows 3.11 tests: a 486/66 with 16 MB of RAM, and a Pentium/90 with 24 MB of RAM. The 32-bit performance of the Pentium/100 running NT was similar to the 32-bit performance on a Pentium/100 running Windows 95 and Windows 3.11, but 16-bit performance was slower on NT.

Windows NT runs 16-bit programs as a separate task called Windows on Windows (WOW), which may account for some of the differences. Because WOW converts all 16-bit function calls to 32-bit operations, it's slower than Windows 95 or Windows 3.11, which executes the 16-bit calls directly. There's no 16-bit code in NT, so the 32-bit code enjoys a clear advantage.

The tests on the 486/66 running NT were faster than those run on the Pentium/100. The 486/66 was heavily tuned by our network performance staff, while NT on the Pentium was installed with little or no tuning.

I was unable to determine the exact cause for the difference in performance in time for this article deadline, but I suspect it could also be due to network interface differences as well as operating-system tuning.

I did not find VB4 to be significantly faster than VB3 for a given database technique. Most of the cycles in these tests occur in the back end rather than in the VB code, so the VB code is not the gating factor for data access. Jet 2.5/3.0 may be improved for Access MDB usage, but the tests did not show significant performance improvement for SQL Server. However, serious consideration should be given to VB4/32 to gain the benefits of RDO.

Upgrading hardware isn't always the only approach to improving data-access speed. The platform results show that throwing hardware at the problem app may not solve the problem as much as changing code. A slow database application may still be sluggish on fast hardware.

I also ran the 16-bit test programs on a Pentium client using an Oracle 7 database server running on Unix (see Table 7). I ran only the 16-bit tests because I did not have a 32-bit version of the Oracle ODBC component, called SQL*Net, at the time the tests were run.

The results should not be compared to the SQL Server results because the server hardware, networking, and machine load were different, but the ratios between different database-programming techniques were similar. Like the other tests, results have a wide range: the ratios between fastest and slowest are about 24 to 1.

CAUGHT IN A TRAP

Code tuning is another important performance aspect not shown in the results. In the early coding phase of the test, I fell into a number of common traps: using dynasets for all the queries, using test tables that were too small, and using the FindFirst method to locate a row in a table.

I was able to improve the performance of the code that read a 200-row table by changing it from a dynaset to a snapshot, eliminating the Jet overhead of building a key set. In my initial testing, the customer table contained only 200 rows. For updates and deletes, I opened a dynaset, positioned on the row with a FindFirst method, and updated or deleted the row.

When I loaded the customer table with 100,000 rows, the program performed poorly because it was taking more time building the dynaset key set. An ODBC spy program showed thousands of ODBC calls going across the network. I changed it to open a dynaset on the desired row and update/delete it, and performance improved significantly. FindFirst is not included as an RDO method, which means it may work fine for Access and files-based ISAM databases, but FindFirst isn't appropriate for SQL remote database servers.

Switching from Jet DAO, Jet MDB, Jet attached MDB, DAO SQL passthrough, RDO, and ODBC calls dramatically affects performance step by step. For 32-bit apps, RDO is a strong contender.

The choice of technique is up to you. I'd be happy to hear of your results if you download the test programs from *VBPJ* online sites and duplicate tests in your own environment. ×