White Paper

Migrating Your Visual FoxPro Application to a Client/Server Platform

By Daniel R. LeClair, Senior Developer

Abstract

This white paper details issues you may encounter in migrating your monolithic Visual FoxPro (VFP) application to a client/server platform, specifically with SQL as the database. In addition, this white paper offers many tips for ensuring a successful migration.

Contents

Overview

Microsoft® SQL Server™ offers the database developer incredible power and flexibility, but learning to work with it is a lot like learning to play chess. The basics are easy, but mastering the product takes some effort. This white paper will not discuss SQL Server in-depth, as there are many excellent books already available on the subject. Instead, I will discuss the basics of migrating your monolithic Microsoft Visual FoxPro (VFP) application to a client/server platform, specifically with SQL as the database.

To that end, I will cover the following:

<u>Conversion issues</u> – what do your users expect, what do you hope to accomplish with the conversion, and most importantly, can your application be converted?

<u>Data conversion</u> – what design issues will you face, and what are some of the differences between SQL-stored data and VFP-based data.

<u>Data Access Techniques</u> – how do you access SQL-stored data from within VFP? We will investigate remote views, SQL pass-through, ADO, and how they compare with each other.

<u>System design issues</u> – what are some of the more important performance and “mechanical” issues you will need to consider?

<u>Supporting your application</u> – how to deal with table changes, updates to the code, error reporting, and so on. This topic is often ignored, but is just as critical as deciding how you’re going to get to the data.

Before the Conversion

I want to begin by talking a little about the expectations you and your users might have regarding data conversion.

One common misconception that developers have is that moving to SQL Server will increase your application’s performance. In general, client/server databases are more efficient than file server-based systems because only requests and results are traveling across the network. However, to achieve that efficiency, requests and results must be small and carefully designed. A lot of requests and large result sets will cause performance problems. It is important to remember that simply switching to SQL Server from a VFP-based file-server system will not necessarily result in increased performance for the user. You have to carefully consider which portions of the system to migrate in order to get the best performance possible.

Before beginning the conversion of your application, you should first consider the future of the application as well as scalability. Is there a possibility that you’ll need to access to the data from outside the system? Do you need to consider alternative interfaces? Do you need to consider issues regarding replication? Where do you and the client see the application going in the next several years? If, in the near future, you need to expand the scope or capabilities of the application beyond its immediate purpose, a conversion, as opposed to replacement or refactoring, is probably a waste of time and resources. This is not to say that converting an application is always a waste of time. There are a lot of good, solid reasons to move to an SQL Server back end, perhaps the most compelling reasons being improved table size and increased security options. A converted system may also provide you with a platform from which to expand in the future. The point is that you should at least think about the future before beginning any conversion project.

Will Your Application Survive Conversion?

Perhaps the single biggest concern you will face in converting an application will be whether the application will even survive the conversion. In attempting to address that concern, consider the following:

Are there any VFP-specific functions built into your data access methods or validation routines? Obviously, these will not translate into SQL Server. If VFP-specific functions do exist, where are they hidden? They could be in index keys or the business logic. Older VFP applications often have functions deeply embedded in critical areas of the system.

Where is the business logic embedded in your application? Will you be able to alter it to deal with differences in SQL syntax and data access techniques? Business rules in the user interface (UI) can also relate to more trips to the server, which affects performance. If you do have business logic in UI forms or if you don’t know where it is located, it might be prudent to think about rewriting everything from the ground up.

You also need to consider SQL capacity limitations. This might seem like a strange concern, given that the data storage capacities of SQL 2000 are measured in the thousands of terabytes (1,048,516 TB to be exact). However, certain aspects of its components, like tables and index keys, do have limits that differ from VFP. Perhaps the most notable is row width. Row width in SQL Server is limited to 8096 bytes. VFP row width can be up to 65,500 bytes. That’s a big difference (although, realistically, it is limited by the allowable table size and number of fields allowed). Key width is a similar problem. Again, SQL is more limited than VFP, with an upper limit of 16 columns per index and 900 bytes per key. Both of these issues, if they exist in your application, need to be addressed before you move forward.

Also consider the degree of normalization in your application. Do the current tables have compound primary keys? Are they optimized for data storage or constructed for easy reporting? SQL Server performs best with highly normalized tables, but there are often very sound business reasons for not doing that. In considering this issue, keep in mind that legacy VFP applications tend to have normalization issues simply because the original programmer didn’t know any better

Lastly, you need to consider the data access issues. As an example, does the application have navigation toolbars? Does your system provide total access to all the records in all the tables?

This is very common in VFP applications, but can be the kiss of death for client/server applications with high record counts. Even moderate numbers of records, in the order of tens of thousands of records, will be difficult to navigate simply due to the amount of data that needs to be sent to the client.

Data Design Issues

Another bump in the road that you will likely encounter is the issue of differences between VFP and SQL Server with data types and data behavior. In my projects, these differences have almost always been evident. These include:

1) Differences in data types, including definitions and values stored in SQL Server

2) Dealing with null values, empty values, and the importance of default values

3) The importance of key definitions and the options offered by SQL Server

4) Differences in index types available in SQL Server

I will discuss each of these in the next few paragraphs.

Data Type Differences

First I'll discuss the data types that I call the "SQL oddities." These are the data types for which there are no equivalents in VFP and, therefore, probably will not be used much.

The first data type I will discuss is TIMESTAMP, which is a unique, eight-byte binary value generated by SQL Server. Ostensibly used to "brand" records when they are created, the value translates into unreadable binary data for VFP. However, in certain circumstances, having a TIMESTAMP column can help increase query performance on an SQL table.

The second data type is CURSOR, which holds a reference to a special SQL data type used to access data at individual row levels. Related to CURSOR is the third type, TABLE, which is used to store result sets for later processing. TABLE is not available for definition within permanent tables (only for temporary tables), and is also used, for example, for processing within a stored procedure. Both of the CURSOR and TABLE data types are resource-intensive and you should use them sparingly.

The last data type is UNIQUEIDENTIFIER, which actually is a very useful data type that I will cover later.

SQL Server supports all of the other data types that VFP developers are familiar with, but in some odd flavors. Obviously, the first question you have to ask when defining SQL table columns is what type of field to use. The next question to ask, and if you haven't played around with SQL yet this is going to seem a strange question, is what size field? By the "size" of the field I am not referring to the length of the field, that's a different issue. Instead, I am referring to the fact that SQL offers several field sizes for a given data type, each with different minimum and maximum capacities.

For example, consider the integer field. SQL actually supports five sizes of integer fields. The smallest integer field, called BIT, is equivalent to the VFP logical field because it can store only values of 1 or 0. The other integer fields, TINYINT, SMALLINT, INT, and BIGINT, support progressively larger ranges of values. All of these options make the issue of field size an important one. In general, you will want to use the smallest field type possible. Doing so reduces storage requirements and can improve performance.

Speaking of the BIT data type, I want to discuss it in a little more detail. VFP ODBC translates BIT values to True or False, even though they are stored in SQL Server as 0 or 1 values. In addition, unlike in VFP tables, you cannot index BIT columns, which means that queries on them will be inherently slow. On the other hand, you can index TINYINT columns. So, for a small storage cost, the TINYBIT column is a good alternative to the BIT column.

Another data type that offers a variety of capacities is the character data type. There are two you will be most familiar with, CHAR and TEXT. The TEXT type is the equivalent of VFP's MEMO type. SQL offers a third type, called VARCHAR, which stands for “variable character.” The VARCHAR data type performs like the TEXT type, in that the length stored in a row varies according to the data. VFP treats this, for display purposes, as a MEMO data type. However, unlike the TEXT data type, VARCHAR columns are searchable (TEXT data types are searchable as well, but only if you enable SQL full-text search abilities, which is a topic unto itself) So, which data type should you use? Microsoft recommends using the CHAR data type for columns where the data is of a consistent length (like a Social Security or telephone number) and the VARCHAR data type for varying-length values (like names and addresses). However, there can be performance issues with VARCHAR columns due to the way SQL stores the data. We'll address that issue later.

Similar to CHAR, there are three different sizes for binary data types (BINARY, VARBINARY, and IMAGE), as opposed to the single GENERAL type in VFP. Again, the difference is capacity. BINARY is recommended for values with the same fixed length while VARBINARY is for variable-length values. Both BINARY and VARBINARY have an upper limit of 8K. The IMAGE data type, on the other hand, is variable-length and can be used for values in excess of 8K.

Dates are another important area of difference between VFP and SQL Server. SQL always uses the DATETIME data type for dates. There are two data types, DATETIME and SMALLDATETIME, which differ in their precision as well as their upper and lower limits, but there is no DATE equivalent. This is important to remember when constructing select statements, because actual date ranges often have to be specified when building queries. To illustrate this, I would like to present an example.

The following code[1] generates a small data set based on the "pubs" customer table, then tries to query the results using two different data comparisons. The first comparison uses a perfectly VFP-acceptable date comparison, but results in only half the records.

The second uses a high/low date comparison to get all the records properly:

```
CLOSE ALL
CLEAR ALL
CLEAR

LOCAL ;
        lcAddress,;
        lcCompany,;
        ldDate,;
        lcString,;
        lnStart
OPEN DATABASE sqltest

SELECT 0
use customer

WAIT window 'Importing...please wait' nowait

lnStart = SECONDS()

* open a connection
lnConnect = SQLCONNECT("con_local")

* clear existing records
lnResult = SQLEXEC(lnConnect, "truncate table importtest")

* first we will generate entries with DATE() only
FOR ln1 = 1 to 100
        ldDate = DATE()
        lcCompany = STRTRAN(company,"'","''")
        lcAddress = STRTRAN(address,"'","''")

        lcString = 'insert into importtest ('company,contact,title,' + ;
        address,city,region,postalcode,phone,fax,maxordamt,' + ;
        'ddatetime) values (' + ;
        '?lcCompany,?contact,?title,?lcAddress,?city,' + ;
        '?region,?postalcode,?phone,?fax,?maxordamt,?ldDate)'

        lnResult = SQLEXEC(lnConnect, lcString)

        SKIP IN customer
        IF EOF()
                GO top
        ENDIF
```

[1] The code sample was constructed using a sample DBC, which you can download from a link on this Web page.

```
NEXT ln1

* next we will use proper datetime values
FOR ln1 = 1 to 100
   ldDate = DATETIME()
   lcCompany = STRTRAN(company,"'","''")
   lcAddress = STRTRAN(address,"'","''")

   lcString =   'insert into importtest (' + ;
   'company,contact,title,address,city,region,postalcode,’ + ;
   ‘phone,fax,maxordamt,'   + ;
   'ddatetime) values (' + ;
   '?lcCompany,?contact,?title,?lcAddress,?city,' + ;
   '?region,?postalcode,?phone,?fax,?maxordamt,?ldDate)'

   lnResult = SQLEXEC(lnConnect, lcString)

    SKIP IN customer
        IF EOF()
                GO top
        ENDIF

NEXT ln1

* now test getting the results
ldDateFrom = DATE()
ldDateTo = DATE() + 1
SQLEXEC(lnConnect, "select * from importtest where ddatetime = ?ldDateFrom",
"testthis")
WAIT window "Record count looking for single date only: " + ALLTRIM(STR(RECCOUNT()))
* The right way
SQLEXEC(lnConnect, "select * from importtest where ddatetime >= ?ldDateFrom AND
ddateTime < ?ldDateTo","testthis")
WAIT window "Record count looking for higher/lower data comparison: " +
ALLTRIM(STR(RECCOUNT()))
```

The first test, which performs a comparison on the date only, returns only half the records that the second test (which takes the time into account) returns. Obviously, this could be a serious problem, so keep that in mind as you are constructing queries.

International Data and Other Issues

International Data

Another level of complexity is in dealing with international issues, and in particular storing international characters. Remember the varieties of character data types we just discussed? Actually, there’s more: the Unicode data types NCHAR, NVARCHAR, and NTEXT. These store data in Unicode form based on the UNICODE UCS-2 character set. VFP treats them as the equivalent non-Unicode character types, but in actuality the storage cost is double that of CHAR and VARCHAR. These data types should only be used when absolutely necessary.

Row-Level Access vs. Set Access

One other important difference that I would like to highlight is how you access the data within the two different databases. VFP developers are used to having access down to the individual rows of a table. SQL Server, on the other hand, deals with data as sets of records. SQL “select” statements return results in sets, and operations are (in general) performed on these sets. This difference is transparent when accessing SQL-based data through VFP remote views or ADO; however, within an SQL stored procedure, for example, you cannot get to a specific row without loading the data set into a CURSOR data type. (Remember, the term “cursor” has an entirely different meaning in SQL than it has in VFP). Although this isn’t so important if you are using remote views, it can be a stumbling block when writing stored procedures that have to process records one by one.

What this means is that there is no equivalent to VFP’s GO TO, SKIP, and SCAN/ENDSCAN commands. Even if your data is physically ordered in the way it would be accessed, you still have to execute some type of query to get the next record.

That is, of course, unless you load the result set into a CURSOR variable type. You can “walk through” cursors, similar to VFP tables. However, use them sparingly because cursors are costly in terms of performance.

Paging Issues, and the Mechanics of SQL Data

How SQL stores records can also affect system performance. Remember what I mentioned earlier about the maximum table width being 8K? SQL Server’s basic unit of data storage is a page, which just happens to be 8K, and any given row cannot span a page. SQL crams as many contiguous rows as it can onto one page before having to find space on another. As you can imagine, having row data scattered over many different pages can affect performance, and this will happen in systems with intense data input and modification.

This paging issue is why variable character fields can cause performance issues. If you update a row with a VARCHAR field, and the row no longer fits the page it was originally on, SQL will have to move it to a different page. In an active system with lots of data changes, this increases the number of required operations (thus affecting performance).

Nulls, Defaults, and Empty Values

Besides differences in data types, there are a number of differences in the way SQL stores data internally within a table, which can cause some migration problems. Null tolerance is one such issue. Even though VFP has been able to deal with NULL values for some time, older applications can’t deal with them, and this may cause you some heartache when trying to adapt to the new database.

Null tolerance doesn’t simply mean that the UI will display them as spaces instead of “.NULL.” (although this is one aspect of it). More importantly, you’ll need to take “null propagation” (meaning, X + .null. = .null.) into account in computations. Dealing with this issue may involve a great deal of code refactoring if you decide to allow null values in the data.

By the same token, you can’t simply exclude null values from your tables because of the problem with empty field values. While it is allowed in VFP, SQL Server does not understand empty values in some circumstances. Columns and variables have to either be populated or have null values (although spaces are valid values for character fields). This is a particular problem with “DATETIME” fields, as attempting to store an empty value into a SQL column results in the “base” default value being stored. For “DATETIME” columns, the result is a stored value of “01/01/1900,” which is probably not the desired result.

This illustrates the importance of properly setting up default values in your table definitions. Some fields have a “natural” default value when nulls are not allowed; character columns can store empty character strings (“ ‘’ ”), numerics can store zeros. Two fields don’t: DATETIME, as we just covered, and UNIQUEIDENTIFIER. DATETIME fields, when not-nullable and consistent with your business model, should have a default value specified. For example, you can use the GETDATE() function to store the current date-time when the row is created. UNIQUEIDENTIFIER fields, like DATETIME, must either be null or have a value that fits the algorithm.

There’s one other column that I usually recommend as being non-nullable, and that’s the BIT field. Null is valid, but usually doesn’t make much business sense. A better default is either one or zero. In general, specifying a default value on non-nullable columns is good practice, as it shortens the number of columns required when inserting new records, which is important if you are doing this with SQL pass-through.

Keys

One important data design issue I would like to spend a little time on is the concept of keys. Hopefully, this is a familiar subject to you, but it is important enough that we should review it anyway.

Unlike in VFP, there is no natural row numbering within SQL tables. Rows are identified by a “key,” a column (or combination of columns) that uniquely identify a record within a table. This is crucial, as there must be some way of echoing updates back to the database from clients that access it. Keys come in two flavors: “natural” keys, which have some meaning to database users (like a phone number or SSN), and “surrogate” keys, which have no meaning because they are simply values generated to identify records. Surrogate keys won’t be affected by changes in business requirements. Regardless of which type of key is being used, they must always be unique within a table and cannot be null.

Key columns are also further defined as “candidate,” “primary,” and “alternate” keys. Since a table can have more than one column with unique values, each is known as a “candidate key.” Only one candidate key can be chosen to be the formal “primary key” (“PK”), specifically marked as such within the SQL table structure. Marking a column as the PK will automatically build a unique index on that column. Any candidate keys not marked as the PK are known as “alternate keys,” and if there is a unique index created (which must be done manually), they can be used to relate the table to other tables in which the columns are present.

While there is no rule that says a table must have a formal primary key, it is a highly recommended practice.

Optimally, the values should be generated at the time a record is inserted, in order to ensure uniqueness before the value gets stored. This can be a problem with natural keys, as the algorithms required to generate them can be complex and may involve interaction from a user. Surrogate keys, which have no business meaning, are much better choices as primary keys. Even if your planned table structures have some obvious natural key combination, it is still a good idea to use SQL-generated surrogate key columns as the primary keys.

Generating Keys in SQL Server

Visual FoxPro (up through version 7) has no native key generation abilities; you have to code your own. SQL Server, on the other hand, offers two native surrogate key column types with automatic value generation: IDENTITY columns and UNIQUEIDENTIFIER (also known as “Globally Unique Identifier” or GUID) data types.

Let’s examine these two in a little more detail.

IDENTITY Columns

IDENTITY columns are specially flagged integer columns, the values of which are generated automatically on record insert. Integer columns have a small footprint—4 bytes for INT and 8 bytes for BIGINT—yet have a very high upper limit in terms of values. With a maximum value of 2 to the 63rd power—hundreds of trillions—for BIGINT, it is highly doubtful that you’ll ever exceed the number of keys possible. Identity columns can be seeded with whatever value you choose to start with, and you can also set the increment rate. You can assign multiple identity columns for a table with SQL, but only one can be flagged as the primary key.

Value generation can be enabled and disabled with the SET IDENTITY INSERT command, the state of which is maintained within any given connection (more on connections later). This enables you to force values into the column if necessary (a good example of when you might need to do this is during table maintenance, which will also be covered later).

High possible value range coupled with efficient storage makes identity columns a good choice as a primary key. But, there are a couple of issues to keep in mind. First is that the value, while unique within a table, is not unique within a database. This may cause problems if you hare going to do any kind of replication. The second—and this applies equally to GUIDs—is how to get the new key value from a parent table in order to fill foreign key fields in child tables.

What Did You Just Generate?

SQL Server provides several system functions and variables to help you determine the last identity column value generated.

IDENT_CURRENT() returns the last identity value generated for a specific table in any session and any scope (pass the table name as a parameter). SCOPE_IDENTITY() returns the last identity value generated for any table in the current session and the current scope. You will have to store the value of this to a local variable immediately after an insert in order to make use of it.

@@IDENTITY returns the last identity value generated for any table in the current session, across all scopes. A word of caution: @@IDENTITY will return the wrong value if a table trigger inserts a record into another table with an identity column specified.

For VFP users, a "session" is the amount of time a connection is held.

You can take advantage of these within triggers and stored procedures to populate foreign keys within child tables (although they are only available to VFP thru a stored procedure or SQL pass-through).

Globally Unique Identifiers (GUID)

The other option is a data type called UNIQUEIDENTIFIER, or GUID. This means that the value is unique for every table in every database on every computer in the known universe (no guarantees beyond the border of the galaxy, however). This makes the GUID a good choice as a primary key for tables likely to be involved in replication. GUIDs, however, are 36-character-wide field types with greatly increased storage requirements.

GUIDs are not automatically generated like identity columns; you must call the NEWID() function in the column default in order to generate them. There are no equivalent functions to get the last GUID generated, but there really isn't a need for one, since they are generated by a function separate from the data type. Although you can use the NEWID() function as the default value, you can just as easily call it separately first to get a GUID, then include that value when inserting a record. Here is a small SQL script that illustrates this:

```
use pubs
create table testthis (guid_key uniqueidentifier, cname char(36))
declare @guid uniqueidentifier
set @guid = newid()
insert into testthis(guid_key, cname) values (@guid, 'This is a test')
select * from testthis where guid_key = @guid
```

Indexes

Let's go over one additional data design issue you need to keep in mind: indexes. Fortunately, as VFP developers, we have long been aware of the impact of good table indexing, as that is key to Rushmore optimization and fast queries. Good indexes have just as direct an impact on

SQL Server database performance; in fact, this is the one of the first places to look when you have a performance issue with a SQL-based system.

SQL Server offers some options for indexes that you may not be familiar with. The first is making the index a “clustered index.” This means that the rows in the table are physically sorted in the index key order for faster access time. Only one clustered index is permitted per table, and you’d want to build it on a key pattern that is the most likely search path (last name/first name on a customer table, for example). Building a clustered index on a surrogate primary key field is usually not recommended, as it will not increase performance.

One other option that you need to be aware of when constructing indexes is “fill factor.” In short, this determines the amount of space available for adding new keys within an index before performance starts to degrade. If you anticipate adding a lot of records to a table, you want a high fill factor. Relatively static table indexes should be built with a low fill factor to reduce space requirements. This option really illustrates some of the complexities of constructing SQL databases; never rush into this blindly.

Fortunately, SQL Server offers several tools to help you optimize your indexes (and the queries that use them) for best performance. These include the Query Analyzer, SQL Profiler, and the Index Tuning Wizard. Time doesn’t permit us to go into these in detail; I recommend that you spend time experimenting with them on your own, or hire a DBA to assist you. The long and short of it is that good index construction is key to getting the most out of SQL Server.

Data Conversion and Data Access Techniques

Data Conversion

So how do you get the data in your current system over to SQL Server? Just as importantly, how do you get the data out again for use within your application? We are going to spend the next several paragraphs examining these crucial issues, since they are related.

First, let’s examine some of the options that SQL Server has for populating data. There are a couple of built-in options available, but in my experience these are less than optimal when working with VFP tables for a variety of reasons. The first is the “old standby” Bulk Copy Program, or BCP (this is actually a command-line interface that harkens back to DOS days), and the newer BULK INSERT command. Both of these offer the ability to quickly load data into existing SQL tables from comma-delimited text files, but only if the data is already in the right format. If you have to do any data transformation, this won’t be much use to you. The one advantage this has is that it will bypass the logging of new records. When bulk-loading data, there really isn’t much point to logging the transactions.

The second, and Microsoft-recommended option, is to use “Data Transformation Services” (DTS). DTS has a wide range of options and types of data sources that it can work with. Itcan be scheduled to run at off-peak hours and can use VB-Script to transform data into the format expected by your tables. However, you can't dynamically alter existing SQL database objects as DTS only imports data. Furthermore, when importing from a VFP data source, DTS uses ODBC, which doesn't properly handle empty dates that may come in from your VFP tables.

There are two other methods, though, that we will look at in more detail: using the VFP upsizing wizard and writing your own data import program.

The VFP Upsizing Wizard

The VFP Upsizing Wizard does a decent job of porting over simple, DBC-bound VFP tables and table data (one of its failings is that it will not import free tables). It will, unlike DTS, generate table structures, indexes, defaults, RI information and validation rules, meaning the database doesn't even have to exist in SQL at the point you run the wizard. It will not upsize triggers or stored procedures, which makes sense because these are most likely to have VFP-centric function calls (which, naturally, won't port over).

When running the wizard, you have the option of specifying individual tables or groups of tables to upsize, but the data within each table is all-or-nothing, and there are few data transformation options beyond specifying the field type on the back end. So, for complex imports that involve some degree of data remapping or value changes (like dealing with empty dates), this is not your best choice. It does, however, generate the scripts required to do the import, and produces some useful reports, which may provide you with a “jumping-off” point for writing your own data conversion routine. In addition, you have the option to replace your local tables with remote views (more on them later) for relatively seamless switchover of your data source, provided you have a well-behaved application.

Some words of caution when using the Upsizing Wizard:

1) Table/field/index names that are reserved words in SQL will be created using the word plus an underscore (how useful is THAT?)

2) Tables with capacity issues will not be generated, although the wizard will actually try to do SOMETHING. But a more serious problem exists in that the Upsizing Wizard thinks table width is limited to 1962 bytes (instead of 8060 bytes). This may be a holdover from SQL 6.5 days, but in any event it means that the Wizard will not be able to handle tables of even moderate size.

3) Changing over to remote views is pretty destructive to your local tables. Make sure you really want to do this before running this option!

Writing a Custom Converter

Sooner or later, I think everyone ends up writing their own conversion routine, simply because the tools that are available (DTS and the Upsizing Wizard) don't meet the real-world needs of getting complex data over into SQL Server. What do I mean by "real-world"? Think about the following:

1) What is the condition of your data? Can you guarantee that every record in your ten-year-old application is free of errors or missing key values? Will you need to map to new columns, break out or combine tables, or change values? In all probably, the answer is yes to all three.

2) How about the size of the table that you'll be importing? Porting in large record counts can be a time-consuming task, and you may need the ability to import only specific amounts of records at first for prototyping and testing. And remember – what has been imported might need to be deleted when starting over.

3) How about final installation or unmanned installation? Reinstalling? You'll need the ability to generate the final database—and all of its components—and load it with production.

Fortunately, this is not as complex as it might seem, for two reasons: one, all SQL Server objects can be generated via scripts. Secondly, VFP can pass those scripts directly to SQL Server via SQLEXEC() to make the changes necessary.

Data Access Techniques

And now the discussion that a lot of you probably have been waiting for: what's the best mechanism for getting the data into SQL Server? VFP offers several options here, all with their good points and bad. These include:

1) VFP remote views

2) VFP SQL Pass-Through, either as dynamically constructed statements or as calls to SQL stored procedures

3) ActiveX Data Objects, or ADO, using either the RecordSet object or the Command object

With SQL Server 2000, we now also have the ability to accept and generate XML natively; however, we will not be including this in our examination, as it would require a pretty thorough overhaul of an existing system to enable it to deal with XML, which is outside the scope of this white paper.

VFP Remote Views: Pros

Remote views are DBC-defined SELECT statements working thru an ODBC-based connection. Basically, they are just like local views, except that the data is going through ODBC to a remote back end. Typically, remote views are predefined in a DBC, although they can be defined on the fly.

Remote views have gotten a bad rap in the past, but they do have a number of good points to offer:

1) They are convenient, because remote views are as close to "drop-in" technology as you can get for application conversion, as we saw earlier with the Upsizing Wizard

2) Remote views also offer the ability to detect update conflicts and revert changes, and will only send changed data back to the server

3) Remote views offer the ability to easily switch back ends (by changing the connection definition)

4) The ODBC driver automatically handles differences in data types (remember the difference between bit and logical fields?)

VFP Remote Views: Cons

If remote views are so convenient, what's the downside of using them?

For starters, the VFP View Designer, which most developers start out using, has a few well-known faults. Complex views, which are joins of three or more tables, tend to be difficult to do because the View Designer often doesn't generate the SQL statement quite right. In addition, you can create views, using perfectly legal SQL exec statements and DBCSETPROP(), that the Designer simply will not be able to deal with. This will cause anyone problems if they try to open the view in the Designer.

Performance used to be a big issue with remote views, but with SQL Server 2000 and VFP 7 this has become less of a problem, although remote views still typically slower when compared to the other data access techniques.

Remote views require the use of a VFP database container. This can cause you grief in a couple of different ways. Most notably, if the remote views specify a field list, the DBC can "break" if the table structure changes on the server. If you allow dynamic creation of remote views (for user-modifiable views) you'll have to be concerned with DBC bloat.

Remote views can't call SQL stored procedures. This is a problem, because right there you lose a lot of the power of SQL Server and nearly all of the inherent horsepower that SQL hardware can offer.

But the biggest issue, in my opinion, is that if you want to make changes to the data, it has to take place on the client. This means that you could be passing lots of data back and forth to the server, which has a direct impact on performance. Every trip to the server affects performance. Server-based operations are faster, and speed is something you need to keep thinking about in client/server operations. This is also a problem in that remote views are basically unaware of data changes—default values, key generation, and so forth—that may occur on the server side. While true for any data on the client side (no matter how you got it there), if you APPEND

BLANK in a remote view, you stand a good chance of accidentally overwriting default values with blank or null values.

SQL Pass-Through: Pros

VFP provides a second method to get at SQL-based data, called SQL Pass-Through (SPT). SQL statements are sent through the connection to run directly against the server, via the SQLEXEC() function. Connections can be DBC-defined (using ODBC), or opened either thru SQLCONNECT (again, using ODBC) or SQLSTRINGCONNECT (for DSN-less connections).

SPT calls are typically faster than remote views, although this is not as dramatic as it was with earlier versions of VFP. But the big advantage of SPT over remote views is their ability to call stored procedures. SQL stored procedures are stored in the database in an optimized or compiled form. Just like a compiled VFP program, stored procedures execute faster than ad hoc queries (which must be compiled before running). In general, you will get better performance, not to mention flexibility, by using stored procedures instead of building select statements dynamically and passing them directly through SQLEXEC(). The exception to this is in building simple, one-line SQLEXEC() statements. There appears to be no noticeable difference between single-line statements and stored procedures in terms of speed.

SPT gives you the ultimate flexibility in terms of what to do with the data. Normally, returned result sets are passed into a VFP cursor; however, with the use of the SQL OUTPUT parameter, single-row results can be passed directly to VFP variables when executing a stored procedure. Here’s a small example that runs against a custom stored procedure in the PUBS sample database. First the stored procedure:

```
create procedure sel_first_by_fname

        @fname varchar(40),

        @au_id id output

as

set nocount on

select top 1 @au_id = au_id from authors where au_fname = @fname
```

To call this from VFP, you need to first establish a connection (I’ve gone with a DSN-less connection for simplicity), and then pass in a local variable by reference as the “receiver” for the outputted value:

```
lcConnectString = ;

"DRIVER={SQL Server};SERVER=(local);DATABASE=pubs;Trusted_Connection=Yes"
```

```
lnConnect = SQLSTRINGCONNECT(lcConnectString)

lcResults = ''


lcString = "exec sel_first_by_fname 'Johnson', ?@lcResults"

SQLEXEC(lnConnect, lcString)

?lcResults
```

You don’t have to generate a result; statements like INSERT and UPDATE can execute directly against the backend. However, for interactive changes, the data still needs to be sent to the client.

SQL Pass-Through: Cons

SPT has its downsides as well, is the top one being that there is more work required on your part. SPT has none of the “knowledge” of how to communicate results back to the server. Where this is handled automatically by the remote view, you will have to work out the details of updates and conflict resolution.

Here’s an example: when you fire an SQLEXEC call, the resulting cursor is non-updating, and by that I don’t mean you can’t modify the data, it means that if you do modify the data you are responsible for passing the updates back to the server. To get around this, you can use CURSORSETPROP method calls to tell the cursor how to communicate back to the server. In this example, I use the AFIELDS function as well to set all of the fields (other than the primary key field) to be updatable:

```
LOCAL ;
         laFields[1],;
         lcKeyField,;
         lcUpdateNameList,;
         lcUpdatableFieldList,;
         lcString, ;
         lcTableName, ;
         ln1, ;
         lnConn,;
         lnResult

lnConn = SQLCONNECT('importtest')
lctableName = "importtest"
lcKeyField = "ikey"
lcString = "select * from importtest"
lnResult = SQLEXEC(lnConn, lcString, lcTableName)
store '' to lcUpdateNameList, lcUpdatableFieldList

lnResult = CURSORSETPROP("Tables","dbo." + lcTableName, lcTableName)
lnResult =CURSORSETPROP("KeyFieldList", lcKeyField, lcTableName)
=AFIELDS(laFields)
* use afields to build the update list:
FOR ln1 = 1 to ALEN(laFields,1)
```

```
         * allow update on non-PK fields only
         IF laFields[ln1,1] == UPPER(lcKeyField)
         ELSE
                  lcUpdatableFieldList = lcUpdatableFieldList + ;
                  IIF(EMPTY(lcUpdatableFieldList),'',',') + laFields[ln1,1]
         ENDIF

         lcUpdateNameList = lcUpdateNameList + ;
                  IIF(EMPTY(lcUpdateNameList),'',',') + laFields[ln1,1] + ;
                  " dbo." + lcTableName + "." + laFields[ln1,1]

NEXT ln1

* now set the final properties
lnResult = CURSORSETPROP("UpdatableFieldList", lcUpdatableFieldList)
lnResult = CURSORSETPROP("UpdateNameList", lcUpdateNameList)
lnResult = CURSORSETPROP("SendUpdates", .t.)
* brow and change records
BROWSE
=TABLEUPDATE()
```

One other disadvantage is that you are required to know more of the details regarding the back-end table structures. Personally I don't have a problem with this, and with the improvements to the Enterprise Manager, it is a lot easier to examine the components of an unfamiliar database. A more important problem is that you may have some data translation issues. Names like "O'Brady" can cause a problem, since the data often needs to be "wrapped" in a string to be passed back to the server. To illustrate two different ways around this, consider the following VFP SQLEXEC statements. The first requires that any values that might have embedded single quotes be wrapped:

```
select customer

lcCompany = STRTRAN(company,"'","''")

lcAddress = STRTRAN(address,"'","''")

lcString = "insert into importtest (" + ;

"company,contact,title,address,city,region,postalcode,phone,fax,maxordamt,ddatetime) values ('" +
;

lcCompany + "','" + contact + "','" + title + "','" + lcaddress + "','" + city + "','" + region +
"','" + ;

postalcode + "','" + phone + "','" + fax + "'," + ALLTRIM(STR(maxordamt)) + ",'" + ;

TTOC(DATETIME()) + "')"

lnResult = SQLEXEC(lnHandle, lcString)
```

This string, which passes values by reference, does not require the values to be wrapped:

```
lcString = 'insert into importtest (' + ;

'company,contact,title,address,city,region,postalcode,phone,fax,maxordamt,' + ;

'ddatetime) values (' + ;

'?company,?contact,?title,?Address,?city,' + ;

'?region,?postalcode,?phone,?fax,?maxordamt,?ldDate)'

lnResult = SQLEXEC(lnHandle, lcString)
```

Which style you choose to use is up to you; the former appears to function faster, but you have the added overhead of having to run STRTRAN() to wrap strings. But a bigger benefit is that, in case of an error, the string can be copied to your clipboard and popped into a Query Analyzer window for debugging. The second method, since it doesn't get evaluated until it is run, can't be debugged in that manner.

ADO – Pros

ADO is a collection of COM objects published by Microsoft and used to access any data source that has an OLE-DB provider available. The PEM's available in ADO offer a lot of power & flexibility to the developer, while hiding the complexity of dealing with OLE-DB. If you are interested in the details of working with ADO, I highly recommend reading John Peterson's excellent white paper on the Microsoft web site (see the end of this article for a link).

ADO is the choice for *n*-tier development, but shoehorning it into an existing application can be a difficult chore. Most VFP controls will work with ADO recordset objects; ADO data is available as properties within the RecordSet or Command object, so it is easy to use these properties as control data sources. The exception are grids, and you'll have to substitute third-party classes for the native VFP grid control if you want to display ADO-bound data that way.

Like SPT, changes to the data can be made on either the client side or server side. However, ADO handles SQL data on either side more like VFP developers are used to. What do I mean by that? As mentioned earlier, SQL is constructed to handle data in sets – not in individual records. This is great for batch processing – a single command, for instance, can make changes across an entire table – but interactive manipulation requires the data to be brought to the client with a remote view or SPT (think performance). If you need to navigate through a result set, the ADO RecordSet object has native methods that allow you to do that with either client-side or server-side cursors, so you don't have to bring the data down.

One other advantage that ADO has is that transaction processing and update conflict detection are built-in. Remember, with SPT, constructing this is on your shoulders.

ADO – Cons

As I just mentioned, one of the principle objections many developers have to ADO is that some of VFP's native controls will not work with ADO recordsets. A bigger problem, though, is that the result sets may need to be "converted" to VFP-understandable cursors in order to take advantage of some of VFP's data handling abilities. This can be a time-consuming business, and not one I recommend doing on large result sets.

As with any Active-X object, you can run into DLL issues with ADO. There have been a number of different releases, and you can find yourself in "DLL Hell" trying to make sure the client machines have the right version with all of the PEMs that you are calling in your system.

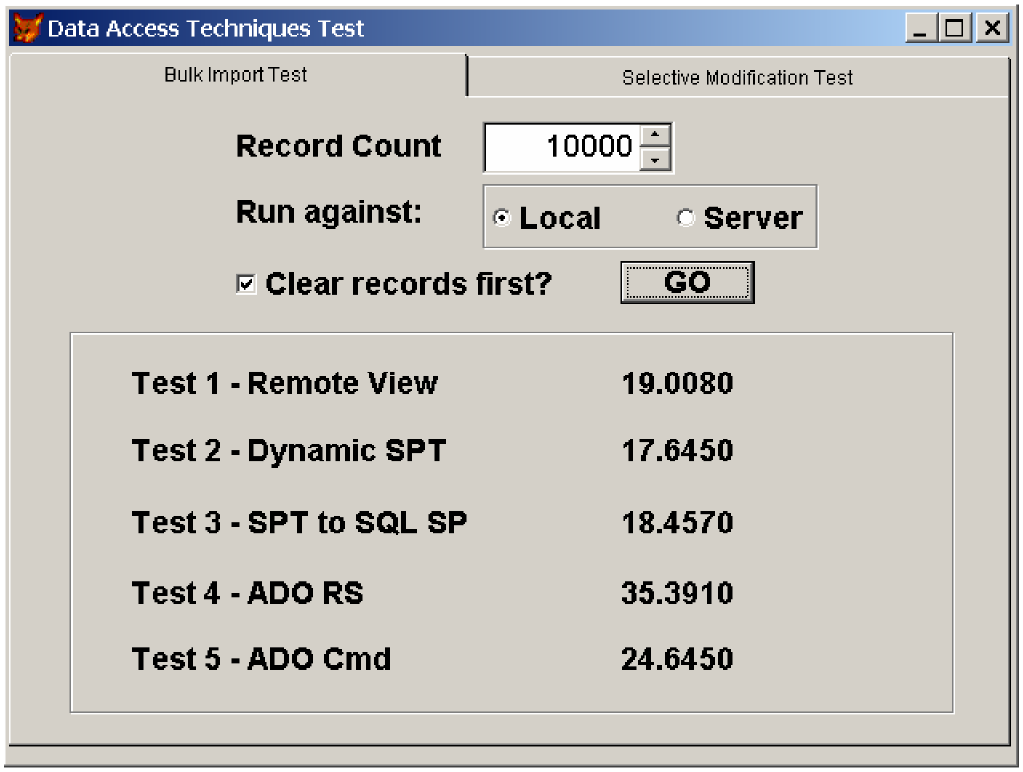
Lastly, ADO takes some work to implement, and that may require a great deal more refactoring that you (or your budget) may want to do.

A Comparison

I have prepared a small benchmarking form[2] that helps to illustrate the speed differences between the various methods, and some of the coding it takes to implement them. The test is fairly simple: it takes the customer table out of the sample "pubs" database, and using a predefined table on SQL Server, imports the records up to as many as you care to do. This is not a comprehensive test by any means, but it does serve to illustrate the differences between the data access techniques that I've outlined above.

The form contains two tests. The first does a straight copy of data from a VFP table into a SQL table, via five different methods. Here's a screen shot of one run, showing the five methods and resulting times:

[2] Available in the samples as data_access_tests.scx.



Record Count: 10000

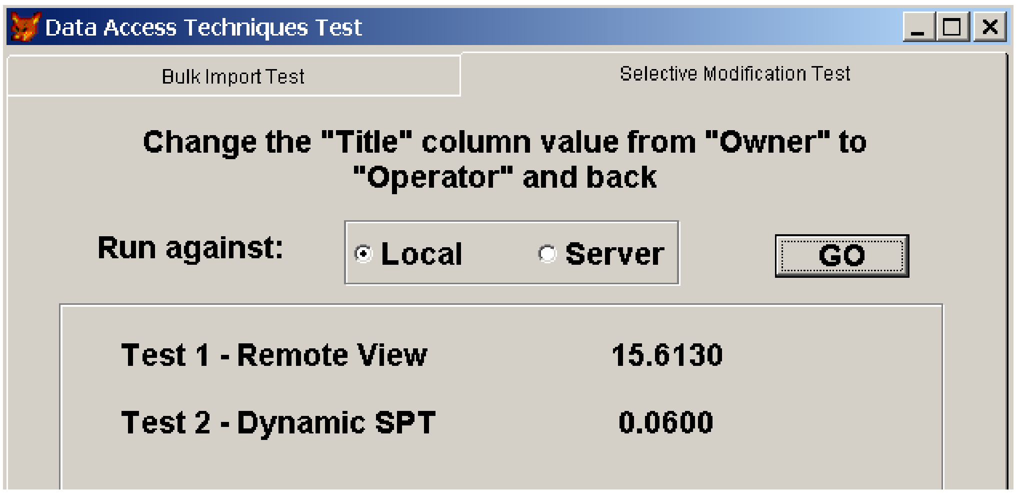
Run against: [✓ Local] Server

[✓] Clear records first?

Test 1 - Remote View	19.0080
Test 2 - Dynamic SPT	17.6450
Test 3 - SPT to SQL SP	18.4570
Test 4 - ADO RS	35.3910
Test 5 - ADO Cmd	24.6450

As you can see, the remote view vs. SQL passthrough times are very close. In fact, remote views can be faster, depending on caching, whether the database size has to be increased, and whether you are hitting a local machine or a server.

The real difference, however, is shown in the second test, which changes the value of data. In order to do that, all the data must be brought down to the client, modified, and then returned, if done through a remote view. In order to do it via SPT, a simple “update…where” clause gets sent to the server. Here are the resulting times:



Change the "Title" column value from "Owner" to "Operator" and back

Run against: [✓ Local] Server GO

Test 1 - Remote View	15.6130
Test 2 - Dynamic SPT	0.0600

As you can see, there is a dramatic difference in times, because the SPT call does not have to bring all the data down to the client first.

Which is Best?

So, which is the best method? There is no “right” answer here. If you were expecting a definitive word on the subject, I can’t give it to you. The method you choose needs to be based on several factors:

1) Your immediate needs, such as how fast do you want to be able to switch from VFP to SQL? Remote views may be the best option, although you can construct your own classes to use SPT as well to deal with many of the data handling chores.

2) How much refactoring do you want to do now? Obviously ADO has some strengths you may want to exploit, but it will take a lot more work to make it fit into an existing application.

3) What is the future scalability of the application? If you anticipate going to a distributed platform or need access through a Web interface, ADO is your best choice.

Again, what I hope I have done here is make you aware of the differences and relative performance of the various data access methods you can use. Ultimately, your business requirements should drive the decision and not a personal preference to one method over any other.

Constructing and Supporting the Application

More Design Issues

So, what else is there besides knowing about data differences and getting the data into SQL? Well, lots, especially if you are taking advantage of the changeover to “address a few problems” within your existing system. Before getting started, let me point out that the issues I’ll be covering in the next few paragraphs will be concerned more with OLTP (on-line transactional processing) systems rather than decision support systems. Decision support systems, such as data warehouses, have different construction and optimization requirements, which are outside the scope of this white paper.

Connections

No matter what data access technique you decide to use, all require some type of connection being made to the back end. Remote views are limited to using ODBC through a defined DSN; SPT can use that, or make its own DSN-less connection depending on the operating system; ADO can go directly through OLE-DB.

If, for whatever reason, you decide that defined ODBC DSNs are what you need, here's a bit of warning: defining a DSN during installation is a pain. It is possible, but the code required is operating-system specific. If, on the other hand, you have decided to go with SPT as your data access method, you can bypass this by using a DSN-less connection. The output parameters example above contains an example of this.

One of the things that really surprised me when I first started experimenting with SQL is that there is no native VFP function to determine whether a connection is still good. This is especially important with interactive data changes (data in grids, for example).

One obvious solution to some of these problems is to have a defined connection class. The following example class will establish a connection, fire passed SQLEXEC statements, and also test to make sure the connection is still good:

```
**************************************************

*-- Class:        cus_connection

#INCLUDE "foxpro.h"

*

DEFINE CLASS cus_connection AS Custom

        ** 08 May 01 DRL - don't define this as a session, it will close

        ** the result set



        *-- connection handle to server

        nhandle = 0

        *-- Holds the name of the server to connect to

        cservername = ""
```

```
*-- Holds the name of the database to connect to

cdatabasename = ""

*-- holds value of displogin so it can be restored when object is released

PROTECTED nhold_displogin

nhold_displogin = 0

*-- holds value of ConnectTimeOut so it can be restored when object is released

PROTECTED nhold_connecttimeout

nhold_connecttimeout = 0

NAME = "cus_connection"

*-- the string to be executed

cexecstring = ''

*-- the name of the result set cursor

ccursor = ''

*-- SQL connection string

cconnectstring = ''

*-- if .t., do not disconnect after each sql exec operation

lpersistconnection = .T.

*-- if Win2k, use trusted connection

ltrustedconnection = .T.

*-- Holds the user id used to make the connection if not trusted

cuid = ''

*-- Holds the password used to make the connection if not trusted

cpassword = ''


*********************************************************************************
PROCEDURE INIT

       * These settings may be more appropriately done on an application level

       * DISPLOGIN: Contains a numeric value that determines when the ODBC Login
```

```
        * dialog box is displayed
        THIS.nhold_displogin = SQLGETPROP(0, 'Displogin')
        * set so the login dialog box is never displayed
        SQLSETPROP(0, 'Displogin',DB_PROMPTNEVER)
        * save value of ConnectTimeOut too
        THIS.nhold_connecttimeout = SQLGETPROP(0, 'ConnectTimeOut')
        * return after trying for 3 seconds
        SQLSETPROP(0, 'ConnectTimeOut', 3)

        IF EMPTY(This.ccursor)
                This.ccursor = "T" + SYS(2015)
        ENDIF

ENDPROC
*********************************************************************************
*-- similar to ado .open method (this.source text is executed with
* this.open method)
PROCEDURE Execute
        LOCAL lnError
        lnError = 0

        * can't do anything if we don't have a SQL statement
        IF EMPTY(THIS.cexecstring)
                RETURN .F.
        ENDIF

        WITH THIS
```

```
                lnError = SQLEXEC(.nhandle, .cexecstring, .ccursor)
                IF lnError < 0 && very useful for debugging
                        * this is a pretty simple error display device that you
                        * would not want to have in a com object
                        .displayerror()
                        _CLIPTEXT = .cexecstring
                ENDIF
                ASSERT lnError > 0 MESSAGE "Error executing SQL command."
                IF NOT .lpersistconnection
                        .Disconnect()
                ENDIF
        ENDWITH

        RETURN TYPE("lnError")="N" AND lnError > 0
ENDPROC
*********************************************************************************
*-- connect to database
PROCEDURE CONNECT
        LOCAL lnResult
        * this might be best hidden
        WITH THIS
                IF .ltrustedconnection
                        .cconnectstring = "DRIVER={SQL Server}" + ;
                                ";SERVER=" + .cservername + ;
                                ";DATABASE=" + .cdatabasename + ;
                                ";Trusted_Connection=Yes"
                ELSE
```

```
                        .cconnectstring = "DRIVER={SQL Server}" + ;
                                ";SERVER=" + .cservername + ;
                                ";DATABASE=" + .cdatabasename + ;
                                ";UID=" + .cuid + ;
                                ";PWD=" + .cpassword
                ENDIF
                This.nHandle = SQLSTRINGCONNECT(.cconnectstring)
        ENDWITH
ENDPROC
***********************************************************************************
*-- disconnect from database
PROCEDURE Disconnect
        * this might be best hidden
        LOCAL lcOnError
        * turn the error handler off
        lcOnError = ON("ERROR")
        ON ERROR *
        WITH THIS
                IF .nhandle > 0
                        SQLDISCONNECT(.nhandle)
                ENDIF
                .nhandle = 0
        ENDWITH

        ON ERROR &lcOnError
ENDPROC
***********************************************************************************
```

```
PROCEDURE DESTROY

        WITH THIS

                * make sure the connection is severed

                IF .nhandle > 0

                        .Disconnect()

                ENDIF

                * be polite, reset the connection and login display options

                IF NOT EMPTY( .nhold_displogin )

                        SQLSETPROP(0, 'Displogin', .nhold_displogin)

                ENDIF

                IF NOT EMPTY( .nhold_connecttimeout )

                        SQLSETPROP(0, 'ConnectTimeOut', .nhold_connecttimeout)

                ENDIF

        ENDWITH

ENDPROC

*********************************************************************************

PROCEDURE DisplayError

        * you'd probably want to hold this as a property for use by

        * parent objects

        LOCAL laError[1]

        =AERROR(laError)


        * display the error message (sub your own code here)

        _screen.NewObject('templbl1','lbltemplbl')

        _screen.templbl1.top = 1

        _screen.templbl1.caption = laError[1,2]

        _screen.templbl1.visible = .t.
```

```
                _screen.NewObject('templbl2','lbltemplbl')

                _screen.templbl2.top = _screen.templbl1.top + _screen.templbl1.height + 5

                _screen.templbl2.caption = "SQL Error No. " + alltrim(STR(laError[1,5]))

                _screen.templbl2.visible = .t.


        ENDPROC


        ******************************************************************************** *--
Tests an active connection

        PROCEDURE test_connection

                LOCAL lcOnError, llError

                * turn the error handler off

                lcOnError = ON("ERROR")

                * this is a simple test to make sure the connection is live

                ON ERROR llSuccess = .f.

                This.cExecString = "select 1"

                llSuccess = This.Execute()

                * restore the regular error handler

                ON ERROR &lcOnError

                RETURN llSuccess

        ENDPROC


ENDDEFINE

*-- EndDefine: cus_connection

**************************************************

DEFINE CLASS lbltemplbl as Label

        FontBold = .t.
        FontSize = 18

        Height = 200

        Width = 500

        WordWrap = .t.

ENDDEFINE
```

Performance

Performance is the “holy grail” of client/server applications. We’ve talked about performance throughout this white paper, and I’m going beat the subject to death here. Pure and simple, performance dictates the success or failure of your application in the eyes of the end user, and correcting performance problems down the road can be costly and painful.

Here are my three “golden rules to remember” about client/server performance:

1) Every trip to the server affects performance

2) Limit your trips to the server for better performance

3) You can improve system performance by limiting trips to the server.

Am I making myself clear?

Limit Trips to the Server

We’ve mentioned a few of these earlier, but here are a few of the ways you can limit trips to the server:

1) Pass only the smallest result set required to accomplish the mission. Never give the user full access to a table. Remember our discussion regarding navigation bars? There is usually no good business reason to maintain these, other than “we’ve always done it this way.” Think of the capacities to which SQL will allow your tables to grow. Does the end user really need to be able to go top/go bottom on a million-record table?

2) Keep static tables on the client if possible. This is more of an option for fat-client systems. Some table data hardly ever changes, such as system codes, state abbreviations, and industry acronyms, so there is no reason to be constantly passing these back and forth. Normal system maintenance patches can handle exporting the revisions out to the clients.

3) Do as much processing on the server as possible. Take advantage of SQL views, stored procedures, and triggers to prepare result sets, populate default values, create child records, and so forth. Think about separating some of the business logic related to preparing and maintaining data onto the server and out of the client.

Other Performance Issues

As I mentioned previously, table design has an impact on SQL performance. Using the right size fields for the job is one area; proper normalization is another. Overall database design is also a factor; by that, I mean how and where the database files are stored in relation to each other? SQL databases can be split into physically separate files on different platters of a RAID 5 system, for example, to speed up access. Here we are getting into the more esoteric aspects of SQL Server, and if your client or corporation can afford to do it, having a DBA will be a big help.

Query efficiency is also very important. Use the execution plan and trace monitoring tools in the Query Analyzer to help determine if your queries are performing at their best. Just like VFP, SQL performance can be dramatically improved by having the right indexes available. Some very good tips for query performance can also be found on http://www.sql-server-performance.com/.

Writing to the transaction log can also affect performance, although in general you will want to do this. Under certain conditions, you may want to avoid writing changes to the log (for example, doing a bulk delete of table rows prior to a data import). SQL 2000 also provides three different recovery models with varying degrees of transaction logging. If you are really having performance problems, this may be worth investigating. Again, under most circumstances, you will want to go with the default full-logging option.

The server itself can also be a place to look for performance improvements. SQL Server is a resource hog, and although you can install other software packages on the same box, running them simultaneously with SQL Server will slow things down. I strongly recommend running your production SQL Server database on a dedicated machine with the most hard drive space and RAM you can afford. Believe me, it will make a difference. SQL Server Enterprise Edition is also capable of running on multi-processor machines, and for very large installations, this is worth considering.

Lastly, one of the things most VFP developers don’t take into account is capacity planning. With SQL Server 6.5 and earlier, this was a much larger issue, as a database’s maximum size had to be established at the time it was created. Expanding its size was very difficult. With SQL 7.0 and SQL 2000, this is no longer the case, but if you fail to take database growth into consideration beforehand, you will see occasional performance slowdowns as SQL adjusts the size of the database. Capacity planning is also important in setting up clustered indexes. Since rows in a table will be stored in the clustered index order, it is important to allow for a certain percentage of growth when constructing the index (done by adjusting the “fill factor”). Exceeding the fill factor will not prevent new records from being added; however, it will fragment the data, causing slow queries.

Metadata

Metadata is data about data – information about the tables, columns, procedures, and other components of a database. As VFP developers, we’re used to certain aspects of metadata being available for UI and report construction, maintenance, and so on. SQL Server does have this information available, but not quite in a manner we are used to.

First, let’s get one thing out of the way. You may have noticed in experimenting with or reading about SQL Server that there is something called “Meta Data Services”. Unfortunately, it is not “metadata” as we VFP developers might understand it. In fact, it is a separately installed service for use more in data warehousing. It’s a dead end if you are looking for information about table structures and so forth.

“Real” metadata is accessible via calls to system tables at several levels:

1) The “master” database contains information on the databases (on any given server) and some replication information

2) The “msdb” database contains info on alerts & jobs managed by SQLServerAgent, backup & restore info, etc.

3) The “distribution” database contains info on replication schedules & transactions

4) Within a database are several tables with info on column sizes, indexes, etc

MS warns, however, that calls to the system tables are unsupported. Specifically, they do not guarantee that the system tables structures won’t change (making any code that you might have that relies on them unstable). Another very grave warning MS issues is about changing the data within system tables. In a word, DON’T. The only exception I can think of would be the “sysmessages” table, which we will talk about later on.

Having said all of that, the system tables have all of the information you might need except one important piece we’ve come to rely on for UI and reports: field captions. Unfortunately, you’re on your own here.

As an example of how you can use system tables, the following SQL user-defined function will return the primary key of a table, where the table name is passed in as a parameter (provided that one column is marked as the PK, which is standard practice here at EPS):

```
CREATE FUNCTION dbo.UDF_GetPkFieldName (@tablename varchar(50) )

RETURNS varchar(50) AS



BEGIN



declare @fieldname varchar(50), @tableid int, @indid int, @colid int



-- get the id number of the table in question from SYSOBJECTS
```

```
select @tableid = id from sysobjects where xtype = 'U' and
        lower(name) = @tablename

-- get the primary key index id from SYSINDEXES for the table
select @indid = indid from sysindexes where id = @tableid and
        name = (select
         sysobjects.name from sysobjects where xtype = 'PK' and
        parent_obj = @tableid)

-- get the column id of the key of the index from SYSINDEXKEYS
select @colid = colid from sysindexkeys where indid = @indid and id = @tableid

-- get the field name of the column found above from SYSCOLUMNS
select @fieldname = name from syscolumns where id = @tableid and
        colid = @colid

return @fieldname
END
```

Transaction Processing

As we've noted before, VFP remote views and ADO have their own transaction processing (TP) abilities built-in. However, if you are constructing a SPT-based application, you will need to handle this on your own. SQL Server has excellent transaction processing mechanisms very similar to FoxPro's. Although there are minor syntactical differences (and other commands for distributed transactions), the concepts of BEGIN/COMMIT/ROLLBACK should be familiar to most developers. But, there are a couple of points that I'd like to make:

First, TP can greatly affect system performance. Microsoft recommends the following strategies for ensuring TP efficiency:

1) Keep transactions short, and perform them within one stored procedure. Conversely, there's no reason to wrap single SQL statements with a transaction.

2) Don't allow user intervention during a transaction.

3) Frequently-used tables should be referenced at the end of a transaction.

MS SQL “Books Online” contains some good examples, so I won’t spend a lot more time on this. However, here’s a point worth mentioning. If you are using dynamic SPT (as opposed to stored procedures), transactions can span multiple SQLEXEC calls within the same connection. By default, transactions are handled automatically; each statement is executed and logged as a separate transaction. You can manually override this, however, via SQLSETPROP() from within your client application. Let’s look at an example VFP code block to see how this is done.

```
ln1 = SQLCONNECT('importtest')
* set transactions manually
* 2 = Transaction processing is handled manually through SQLCOMMIT( ) and
* SQLROLLBACK( )
?SQLSETPROP(ln1,"Transactions",2)
?SQLEXEC(ln1,"update importtest set company = 'Wombats International'")
?SQLrollback(ln1)
?SQLEXEC(ln1,"update importtest set company = 'Wombats International'")
?SQLCOMMIT(ln1)
```

Security

SQL Server has a very robust security model that offers a lot of flexibility to the database developer. Not only is the data protected, but you can also establish rights for particular classes of users down to the column level for particular tables. We won’t cover SQL security in detail – I really recommend having a DBA assist you in developing and implementing a good security plan - but there are a few things that I’d like to highlight, especially if you intend to use SPT to handle database maintenance:

1) On installation, the system administrator password is set to nothing. Empty. And everyone knows it. Be sure to change it.

2) Database objects – tables, views, etc – are defined as belonging to a user. Depending on your security plan, most items are going to be assigned to the default DBO (database owner) account. You will want to keep this in mind if you are constructing database objects programmatically, because if they do not belong to DBO, then other users will not be able to access them.

Here are a couple of security features that you might want to take advantage of

1) SQL 7 introduced the concept of roles into the SQL security model. Approles are “temporary” roles that you can assign for the duration of a connection to perform a variety of tasks that might be outside a user’s normal rights. For example, if a table structure needs to be changed, you can set the app role to allow the change to take place (under carefully-constructed conditions, of course).

2) If you are concerned with clients reverse-engineering your stored procedures, SQL offers the ability to encrypt the source code. However, this should be implemented with caution. The actual code will need to be stored somewhere. One possible solution: you can keep the code external (in a VFP table, checked into Visual Source Safe), and create the procedures

programmatically with the ENCRYPTED setting. However, keep in mind that encrypted SP's cannot be replicated.

Errors & Error Trapping

Let's talk briefly about one more design issue you'll have to deal with, and that is what to do about errors. Because you are now dealing with at least two levels – the client side and the server side – your work load for error trapping and reporting has doubled, at a minimum. If you have any kind of mature VFP app, you probably already have the ability to deal with client-side application errors. But surfacing server-side errors are a little more difficult.

To begin with, there are really three different types of errors related to the new data layer you have added with SQL Server. The first we've discussed a little already, and that is when there is a problem with a connection. Your system will need to know what to do if a connection is severed or times out, which is why having a connection class (or some type of application-level connection handler) is a good idea.

Another type of error, which really only applies if you are using SPT, is what to do if SQLEXEC returns a negative value. If the statement you tried to pass can't execute, SQLEXEC returns –1 with no clue as to why this happened. This can really be considered a system bug, and treated as such with ASSERT statements. One habit I have adopted is to always put store the statement to be executed into a local variable, then pop that variable's value into _CLIPTEXT if the SQLEXEC result is negative (see the EXECUTE() method in the connection class outlined above). This way, you can immediately paste the offending statement into the Query Analyzer to figure out what went wrong.

The last type of error – problems that may occur on the server side when data is being updated – is where you will need to do the most work when modifying your error handling routines. For starters, although VFP knows that an error occurred, the message number returned is always the same (#1526), and the message itself is usually very technical. It may make perfect sense to you, but to a user, it will be incomprehensible.

In fact, VFP knows more about the error than is immediately reported, which you can access thru the AERROR() function. The fifth column of the array will give you the SQL Server error number; the second row of the array will tell you what SQL Server did about the offending statement. If your error handler isn't capturing the value of that array, you'll need to include it. There isn't much more you can do here, except to display something a little less scary to the users. One way that you can do that is by customizing the messages that SQL Server itself generates if you are using stored procedures. The RAISERROR command allows you to construct your own error messages on the fly, or you can permanently install them in the "sysmessages" table using the "sp_addmessages" system stored procedure. This is outlined very well in the SQL Books Online; there is also a very good section on error handling in the Urweiler *et al* book.

Supporting the Application

So – with all that we've covered so far, what else could there be to take care of? Probably more than you think. If you are making the leap to client/server, the reach and scope of your application is probably a lot larger than what you've dealt with before. You will need to develop strategies for testing, installation (both on the client and on the server), and database maintenance and backups that may be different from what you've done up to this point.

Development & Testing

Testing is a sore point with me; one of the first client/server projects I was involved in was hampered by a penny-pinching manager who didn't understand the need for a separate development and testing server. It wasn't until a runaway process locked up the SQL Server box – we only had one – that he finally found the funds for a separate computer for development. Fortunately, SQL can now be installed locally for prototyping, development, and testing independently of any server.

Version control is going to be a little more complex than what you are used to. You will need to make sure that the client or middle tier pieces are compatible with the version of the database that you are working against. And conversely, as new pieces of code are migrated from local workstations to the test platform (or test to production), you'll have to make sure that the database is updated (if necessary). This can be a real challenge if you have multiple developers involved.

Again, because there are now at least two layers to the system, testing is also more complex, and if your development team or department doesn't have anything more formal process than "oh yeah, I tested that," there's big trouble lurking around the corner. At a minimum, you will need the ability to recreate the database and repopulate the tables from a known baseline, so that you can get consistent results when testing complex parts of the system.

Lastly, there are a couple of good tools to help you with database design and development. First, SQL itself comes with a very good Database Diagram tool that you can use to create tables, establish relationships between them, and most of the other developmental stuff you might need to do. However, it will not allow you to make modifications to existing objects in a diagram without changing the actual database objects themselves when the diagram is saved (although you can save the changes to a script to be run later). This feature is specific to SQL Server, and the diagrams are kept within the database itself. There are several database design tools on the market, but the one I have most familiarity with is xCase (www.xcase.com) from Resolution Ltd. xCase has excellent diagramming abilities, can construct the diagrams independently of the database, will reverse- and forward-engineer models for VFP and Oracle as well as SQL, and the diagram files are external, meaning they can be shared and protected by your version control software. And if you don't think that is important, I have some real estate to sell you.

Installation and Updating

Installation and updates are another area where having more than one layer will change the way you work. It probably goes without saying that you'll have different install packages for the various tiers, but more importantly, you will have different update mechanisms as well. Let's talk about installs and update to the client software here for the moment.

VFP comes with its own application builder, but it includes all possible VFP runtime files which makes the installation set pretty big – bigger than what you might want to download across a network. Again, there are a number of third-party tools you can use (InstallShield and Wise Install are two) to create smaller install sets and handle all of the details of a client installation (like putting an icon on the desktop).

Updates are a little trickier, because you probably don't want to have to redistribute the entire install set every time a change is being made. rtPatch by PocketSoft (www.pocketsoft.com) produces small "patch" files by comparing executable versions and building a separate file containing only the changes, without requiring you to know where all of those changes might exist within the application. It was relatively simple to use, requiring only a couple of files to reside on the client (which you can include with the install set…what a concept…). Plan for this now because having to fit this in later is painful.

How you distribute the patch files will most likely depend on your business infrastructure, but it is important to note that you will probably need to coordinate a revision on the client side to a change in the underlying database.

Database Maintenance

This brings us to the topic of database maintenance. SQL Server is a very robust product, but even it needs occasional tune-ups. Depending on your system size, many routine maintenance items, such as backups, reindexing, and replication, can be scheduled for off-peak hours to minimize the impact on users. SQL has excellent tools for setting up these tasks, but even so, a large database can take the efforts of a full-time DBA to maintain it in peak condition.

Non-routine maintenance, such as making structural changes to a table, are a little more difficult to handle. No system is static; changing business requirements might require revisions to the underlying tables and stored procedures that comprise a software database system. Hopefully, you are developing a well-documented system and know all of the possible places a table change might affect the client code (you are, aren't you?), and hopefully you are keeping changes like this to a minimum. Coordinating a change to the server with revisions to client software can be difficult, because no matter how many advanced notices you send out, someone will go home with their system up and locks on a table.

If you are building more of a traditional client/server application (as opposed to an *n*-tier app with a browser interface) you will need to have the ability to lock out clients for the time it takes to apply change scripts or revise data within SQL. How you implement that is really going to be decided by your business needs. You can take the brute-force approach and simply go through the SQL Enterprise Manager, find out who is doing what through the Current Activity option, and kick them out. But this is pretty traumatic for the users, who may have a perfectly valid reason for being in the system at the time. A better solution would be to build some type of gatekeeper device that refuses access for users when database maintenance is going to occur.

Additional Resources

We've covered a lot of ground in this white paper, but really have only scratched the surface of how to convert your application to SQL Server. Here are some additional resources that cover many of the topics we've discussed in more detail.

"ADO Jumpstart for Microsoft Visual FoxPro Developers" white paper by John Peterson

http://msdn.microsoft.com/library/techart/ADOJump.htm

"Transactions in a VFP Client/Server Application" white paper by Hector J. Correa

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnfoxtk00/html/ft00i6.asp

Client-Server Applications with Visual FoxPro 6.0 and SQL Server 7.0 (Urwiler, DeWitt, Levy & Koorhan, Hentzenwerke Publishing)

Mastering SQL Server 2000 (Gunderloy & Jorden, Sybex)