



<http://www.progressivebusinesstechnologytraining.com/>

## Optimizing Microsoft Office Access Applications Linked to SQL Server

By: Andy Baron

November 2006

Applies to:

Microsoft SQL Server 2005

**Summary:** One way to create applications that use Microsoft Office Access for creating user interfaces and that use Microsoft SQL Server for data storage is to link Office Access tables to SQL Server tables. This is the type of application created by using the SQL Server Migration Assistant for Office Access. This white paper presents techniques for improving performance and updatability in Office Access applications that use tables linked to SQL Server. (22 printed pages)

### Introduction

Microsoft Office Access supports three primary options for connecting to data stored in Microsoft SQL Server databases:

- Use the Office Access database engine—originally called the Jet database engine—to communicate with SQL Server over ODBC connections.
- Create Office Access Project applications that use an OLE DB connection to communicate with SQL Server.
- Write Microsoft Visual Basic for Applications (VBA) code that uses DAO, ActiveX Data Objects (ADO), or middle-tier objects to connect and manipulate SQL Server data.

This paper focuses on the challenges encountered by Office Access developers who rely on the Office Access (Jet) database engine to connect to SQL Server over ODBC. The most common way this is done is by creating linked tables in Office Access that use the SQL Server ODBC driver to connect to tables in SQL Server databases.

The SQL Server Migration Assistant (SSMA) for Office Access enables you to convert an Office Access database to this type of application by moving your Office Access data to new SQL Server tables and linking to these tables. Any forms, reports, queries, or code that previously worked with the original Office Access tables are automatically connected to the new SQL Server tables.

In an application that uses linked SQL Server tables, two different database engines are at work: the Office Access/Jet database engine that runs on the Office Access client and the SQL Server database engine. The interaction of these two engines can sometimes yield results that are inferior to those obtained by using only the Jet database engine with native Office Access tables. This white paper discusses several of these issues and presents strategies for resolving them. Most of these issues relate to performance or updatability.

## **Understanding and Addressing Performance Issues**

Developers often migrate data to SQL Server expecting an improvement in application performance. Although performance does often improve, there are many cases where it remains the same or even degrades. In some cases, performance of certain queries degrades to an unacceptable level.

The major cause of query performance degradation is when a query involving very large tables requires that all of the data from one or more tables be downloaded to the client. This can happen even when joins or criteria appear to limit the result set to a small number of records. This occurs because sometimes the Office Access database engine determines that it cannot submit an entire query to SQL Server. Instead, it submits multiple queries, often including queries that request all of the rows in a table, and then it combines or filters the data on the client. If the criteria require local processing, even queries that should return only selected rows from a single table can require that all the rows in the table be returned.

The primary strategy for improving performance is to minimize the amount of data returned to the Office Access client and maximize the amount of processing that occurs on the server. To accomplish this, you need to be able to analyze the SQL commands that Office Access is submitting.

### **Diagnostic Tools**

There are two tools that you can use to see how Office Access is communicating with SQL Server. To listen in on the conversation from the server side, you can open the SQL Server Profiler and create a new trace. Select a template that shows TSQL to see all the statements being processed by the server. From the client side, you can edit a Microsoft Windows registry setting that allows you to see the commands that the Office Access database engine is submitting to ODBC.

As always, be very careful when editing the Windows registry. For more information on backing up and editing the registry, see [How to Modify the Windows Registry](#).

To enable tracing of all ODBC commands from the Jet database engine:

1. From the Windows **Start** menu, select **Run**.
2. Type **Regedit** to open the Registry Editor.
3. If you are using a version of Office Access prior to Office Access 2007, navigate to the following registry key, which appears as a folder in the Registry Editor.
4. `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Jet\4.0\Engines\ODBC`

Office Access 2007 uses a customized version of the Jet database engine, named the Office Access Connectivity Engine (ACE), which is not shared with other Windows applications. If you are using Office Access 2007, navigate to the following registry key.

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Office\12.0\Access Connectivity Engine\ODBC
```

5. Double-click the **TraceSQLMode** setting, change the value from 0 to 1, and click **OK**.
6. If Office Access is open when you make this change, you must close and reopen Office Access for the change to take effect.

After making this change in the registry, queries submitted to any ODBC data source are logged in a text file named `Sqlout.txt`. Unless you delete this file or its contents, it continues to grow as new queries are executed and the tracing activity degrades performance. It is very important to return to the Registry Editor and turn the feature off by changing the **TraceSQLMode** setting back to 0 when you are done testing. Running SQL Profiler also has a negative impact on performance, so try to avoid using it on a production server and close your Profiler traces when you are done testing.

Before you can make productive use of these diagnostic tools, you must understand how Office Access interacts with SQL Server. Without that understanding, the SQL statements that you see in Profiler traces and in `Sqlout` logs can be quite puzzling.

## Understanding Dynasets

When you observe how Office Access communicates with SQL Server over ODBC, you will notice that most queries are executed very differently from the way you would expect. For example, if you migrate the **Northwind** data to SQL Server, link to the tables, and open the **Shippers** table in datasheet view while tracing is enabled, you probably expect to see a simple query such as `SELECT * FROM Suppliers`, or perhaps a query that includes the schema name with the table, `dbo.Suppliers`, and that explicitly names the three columns in the table. Instead, both the `Sqlout.txt` file and the SQL Profiler trace show that three statements are executed. The following is what is written to `Sqlout.txt`.

```
SQLExecDirect: SELECT "dbo"."Shippers"."ShipperID" FROM "dbo"."Shippers"
SQLPrepare: SELECT "ShipperID", "CompanyName", "Phone"
FROM "dbo"."Shippers"
WHERE "ShipperID" = ? OR "ShipperID" = ? OR "ShipperID" = ?
OR "ShipperID" = ? OR "ShipperID" = ? OR "ShipperID" = ? OR
"ShipperID" = ? OR "ShipperID" = ? OR "ShipperID" = ? OR "ShipperID" = ?
```

```
SQLExecute: (MULTI-ROW FETCH)
```

**SQLExecDirect** indicates execution of a non-parameterized query. All the quotation marks that you see around object names are comparable to the brackets that Office Access uses (and that also can be used in SQL Server) to handle spaces or other illegal characters in names.

**SQLPrepare** is used to define a parameterized query that is then executed with **SQLExecute**. The question marks are placeholders for parameters. **MULTI-ROW FETCH** indicates that parameter values are submitted, based on values retrieved by the first query, to retrieve up to 10 rows.

A Profiler trace shows the three corresponding Transact-SQL statements that are processed on the server.

```
SELECT "dbo"."Shippers"."ShipperID" FROM "dbo"."Shippers"

declare @p1 int
set @p1=-1
exec sp_preprexec @p1 output,N'@P1 int,@P2 int,@P3 int,@P4 int,@P5 int,@P6
int,@P7 int,@P8 int,@P9 int,@P10 int',
N'SELECT "ShipperID","CompanyName","Phone" FROM "dbo"."Shippers"
WHERE "ShipperID" = @P1 OR "ShipperID" = @P2 OR "ShipperID" = @P3
OR "ShipperID" = @P4 OR "ShipperID" = @P5 OR "ShipperID" = @P6 OR
"ShipperID" = @P7 OR "ShipperID" = @P8 OR "ShipperID" = @P9 OR
"ShipperID" = @P10',1,2,3,3,3,3,3,3,3,3
select @p1

exec sp_execute 6,1,2,3,3,3,3,3,3,3,3
```

This example shows the typical behavior for processing a dynaset, which is the type of recordset Office Access opens when you open a datasheet or any bound form. In the first step, Office Access picks a "bookmark" column or set of columns, which is usually the table's primary key but could be based on any unique index, and retrieves just those values for every row in the table or query. This is often referred to as a *keyset*. Then Office Access prepares a parameterized SQL statement to select all the columns in the table or query for 10 rows at a time. The final step is to execute this statement, which is assigned a number on the server (6 in the example), as many times as needed, passing in 10 bookmark values at a time. If there are two columns in the bookmark, 20 values are passed in at a time to specify the next 10 rows.

In this example, there are only three rows in the table, so the final bookmark value, 3, which corresponds to the last **ShipperID** in the table, is submitted eight times, because the rows are always fetched in sets of 10.

The statement that fetches 10 rows of data is repeated as many times as necessary to fill the current screen and to provide some room for scrolling in either direction. The remaining rows are not fetched immediately unless the user performs an action such as scrolling that brings additional rows into view. In the background during idle time, the remaining rows are gradually filled in until the recordset is complete. In addition, any rows that remain visible are continually refreshed according to a configurable refresh interval that defaults to 60 seconds. Long memo

and OLE object values are retrieved in separate queries only when then their columns and rows are visible.

Dynasets support a continuous two-way conversation between Office Access and SQL Server for each recordset that is open. The rows that are visible are continually refreshed to show the latest data, creating extra network traffic. However, the dynamic nature of these recordsets can also reduce traffic by immediately retrieving only those rows in the vicinity of data the user is actually viewing. If you create a form that is bound to a table containing a million rows of data (not a recommended practice) and the form shows the data from only one row at a time, only 20 rows are retrieved when the form opens. If the user keeps only the first record visible, Office Access continually retrieves the first 10 rows, every 60 seconds by default. If the form is left open long enough, all the rows are eventually retrieved during idle time in many separate batches, but a snapshot retrieves all the rows right away. Because they work with only a few rows at a time, dynasets minimize the duration that read locks are held on the server. This allows other users to modify data without having to wait as long as is necessary for locks to clear.

When the user edits or deletes a row, Office Access executes an update or delete query with a WHERE clause containing not only the bookmark value, which is used to locate the row to update or delete, but also the values for all the other columns. This ensures that another user or process hasn't changed any of those values since the last refresh. If the table contains a **timestamp** column, which is a column that SQL Server automatically updates when the row is modified, only that one column value is added to the WHERE clause. Issues related to updatability are discussed later in this white paper.

The following is a summary of the statements shown in a Sqlout.txt trace log and what each one means. Remember that tracing consumes resources, so don't forget to turn off tracing when you aren't using it!

<b>Trace-log statement</b>	<b>Description</b>
<b>SQLExecDirect:</b> <SQL-string>	Execute non-parameterized user query
<b>SQLPrepare:</b> <SQL-string>	Prepare parameterized query
<b>SQLExecute:</b> (PARAMETERIZED QUERY)	Execute prepared, parameterized user query
<b>SQLExecute:</b> (GOTO BOOKMARK)	Fetch single row based on bookmark
<b>SQLExecute:</b> (MULTI-ROW FETCH)	Fetch 10 rows based on 10 bookmarks
<b>SQLExecute:</b> (MEMO FETCH)	Fetch Memos for single row based on bookmark
<b>SQLExecute:</b> (GRAPHIC FETCH)	Fetch OLE Objects for single row based on bookmark

<b>SQLExecute:</b> (ROW-FIXUP SEEK)	Fetch single row based on some index key (not necessarily bookmark index)
<b>SQLExecute:</b> (UPDATE)	Update single row based on bookmark
<b>SQLExecute:</b> (DELETE)	Delete single row based on bookmark
<b>SQLExecute:</b> (INSERT)	Insert single row (dynaset mode)
<b>SQLExecute:</b> (SELECT INTO insert)	Insert single row (export mode)

## Adjusting Dynaset Behavior

There are several ways that you can adjust the behavior of dynasets to optimize performance and concurrency.

One important consideration is the choice of which unique index is used to populate the keyset. For example, in a large table with one unique index based on an integer column and another unique index based on multiple character-based columns, it is much more efficient to use the first index, which is probably also the primary key. However, the primary key is not automatically the index that Office Access uses.

When you create an ODBC-linked table, Office Access looks first for a clustered unique index. Each SQL Server table can only have one clustered index, which determines the order in which the data is physically stored. The clustered index does not have to be the primary key and does not even have to be a unique index (if not, SQL Server adds a value for each row that makes it unique). If all your unique indexes are nonclustered, Office Access uses the one that happens to be listed first alphabetically, which may not be the most efficient choice.

If you have an index that you want Office Access to use for the keyset, it should either be a clustered index or it should have a name that sorts to the top of an alphabetical list. To control the choice of index programmatically or when you don't have control over the index names on SQL Server, there is no Office Access property that you can set in code. However, you can do something that seems like it shouldn't be possible: you can execute a Data Definition Language (DDL) query in Office Access that creates an index on the Office Access link itself, without affecting the linked SQL Server table. If you want to change a link from using one unique index to using another, you first must drop the index being used, because Office Access prevents you from creating a second unique index for a link that already has one assigned.

For example, suppose the **Categories** table on SQL Server has no clustered unique index. It has two nonclustered unique indexes named **CategoriesCategoryID**, which is the primary key, and **CategoriesCategoryName**. When you link to this table, the index on **CategoryName** is selected by Office Access. When populating dynasets, Office Access retrieves all the **CategoryName** values first. To populate the dynaset, Office Access fetches 10 rows at a time by **CategoryName**. If you open the linked table in design view in Office Access, you will see

**CategoryName** marked as the primary key, even though **CategoryID** shows as the primary key on SQL Server. To change the linked table to use **CategoryID** for dynasets, without affecting the SQL Server database, first execute the following query in Office Access.

```
DROP INDEX CategoriesCategoryName ON Categories;
```

To designate **CategoryID** as the column to use in dynasets, you do not have to use the same index name used on SQL Server. In fact, it is not necessary for there to be a unique index on the server. You only need to specify one or more columns that Office Access can use to identify each row uniquely. To do this, execute a **CREATE INDEX** statement in Office Access, even though you not creating a real index. For example, you can execute the following query.

```
CREATE UNIQUE INDEX LinkCategoryID ON Categories (CategoryID);
```

After executing this query, **CategoryID** will show as the table's primary key when you open the link in design view in Office Access. **CategoryID** values will be used in the keyset for dynasets, instead of the less efficient **CategoryName** values. You can also use this technique to designate the unique identifiers for linked SQL Server views.

Another way to adjust dynaset behavior is to modify the **Refresh Interval** setting, which determines how often the set of records that includes whatever is currently visible in Office Access gets refreshed. If your data is not often changed by any user other than the user currently viewing the data, extending this interval can safely reduce network traffic and improve overall performance. If there are frequent updates by other users, extending this interval can increase the frequency of concurrency errors when rows are updated or deleted. This is because the values included in the **WHERE** clause of the update or delete statements might no longer match the values on the server.

To edit the **Refresh Interval** setting in Office Access, open the **Options** dialog box and select the **Advanced** tab. Don't be confused by the presence of a separate setting named **ODBC Refresh Interval**. That setting determines how often the ODBC connection is refreshed, not how often the data is refreshed. In forms bound to large sets of data, avoid creating excess network activity by calling `Me.Refresh` in your code. It is best to rely on the automatic refresh behavior as much as possible.

There are also several registry settings affecting ODBC connections that you may want to adjust. These are in the same registry key as the `TraceSQLMode` setting. For a summary of the available settings, see the Office Access help topic, "Configuring the Microsoft Jet Database Engine for ODBC Access."

## Snapshot Recordsets

Office Access does not always use dynasets when retrieving data from SQL Server. Office Access uses snapshot recordsets for data that is not updatable and that is not continually refreshed. Snapshots retrieve all the data in a table or query in a single operation and cache that static data until the snapshot is closed. For example, Office Access uses snapshots to populate list controls and to run reports.

If you enable tracing and then create an AutoReport based on a link to the Shippers table, the following is the SQL statement you would see in a Sqlout.txt log or in an SQL Profiler trace.

```
SELECT "ShipperID" , "CompanyName" , "Phone" FROM "dbo"."Shippers"
```

All queries use dynasets by default to populate forms or datasheets, even queries that return recordsets that are not updatable. To change this, open a query in design view, view its properties, and change the **Recordset Type** property from **Dynaset** to **Snapshot**. This property is also available in forms where it affects query behavior only when the query is used as the record source for the form.

For large result sets, snapshot recordsets can consume more resources than dynasets, because they automatically fetch all the data right away. Dynasets have the advantage of delaying data retrieval. Also, for frequently changing data, dynasets provide more up-to-the-minute accuracy. However, if you do not need updatability and if you are not concerned about continually refreshing the data, snapshots can consume fewer resources by confining data access to a single query execution.

You may be curious about a third **Recordset Type** property setting in the Office Access user interface: **Dynaset (Inconsistent Updates)**. This creates a standard dynaset, but in queries against native Office Access/Jet tables, this setting allows you to perform updates that wouldn't otherwise be possible by circumventing referential integrity. You cannot use this setting to circumvent referential integrity that is enforced by SQL Server.

## Moving Query Processing to the Server

One of the advantages of using linked Office Access tables to connect to SQL Server is the ability to combine your SQL Server data with data in local Office Access tables or in other linked data sources. You can even call custom VBA functions from your queries. The local Office Access/Jet database engine makes this all possible. However, it can be dangerous to rely on the local engine to process queries when you work with large tables. Often, Office Access needs to load all of the data from linked SQL Server tables before it can process the query.

Office Access creates a query-execution plan that is tree-shaped, with the source tables as the leaf nodes and the result set as the root. When working with linked SQL Server tables, Office Access tries to create a single SQL statement that it can submit through ODBC to retrieve the root result set. However, it is sometimes unable to do so. It may be able to combine only a subset of all the joins and criteria into this type of SQL statement. In these cases, Office Access submits two or more queries to SQL Server and combines the results locally to yield the final results. The following are a few common causes for this:

- Nested queries that use an outer query to process the results of one or more inner queries
- Complex combinations of inner and outer joins
- Queries that reference data from multiple data sources, even multiple SQL Server databases
- Queries that pass row values to VBA functions
- Queries that use built-in functions or expressions that aren't successfully translated
- Queries that use Office Access-specific SQL syntax, such as PIVOT...TRANSFORM

One of the main reasons for using SQL Profiler or a Sqlout.txt log is to detect when Office Access is pulling down more data than necessary. It is important to test your queries using sample data with enough rows to simulate real-world conditions. Sometimes when a SQL Server table contains a small number of rows, the Office Access query optimizer calculates that it is more efficient to bring down the entire table than to formulate a query that can execute on the server. This could require you to spend time reformulating a query that would actually run as desired with more data on the server.

When you spot a problem, you can try to resolve it by changing the local query. This is often difficult to do successfully, but you may be able to add criteria that are sent to the server, reducing the number of rows retrieved for local processing.

In many cases you will find that, despite your best efforts, Office Access still retrieves some entire tables unnecessarily and performs final query processing locally. In these cases, the best strategy is to work with SQL Server views or ODBC pass-through queries to ensure that processing occurs on the server.

## Working with SQL Server views

SQL Server views are saved SELECT queries, and they are always processed on the server. If you can recreate an Office Access SELECT query as a SQL Server view, you can select from the view or use it in another query. You will be assured that the processing required to create the result set for the view occurs on the server.

Creating a view is easy. SQL Server provides a graphical tool that is very similar to the Office Access query designer, or you can write and execute Transact-SQL code. For example, the following code creates a view that joins data from the **Northwind Categories** and **Products** tables.

```
CREATE VIEW dbo.vwExpensiveProducts
AS
SELECT ProductID, ProductName, UnitPrice,
       dbo.Products.CategoryID, CategoryName
FROM dbo.Products INNER JOIN
     dbo.Categories ON
     dbo.Products.CategoryID = dbo.Categories.CategoryID
WHERE UnitPrice >= 50
```

Views also support features that aren't available in Office Access queries. For example, if you create a view using **WITH CHECK OPTION**, you cannot perform an update that uses the view and that would change a row so that it no longer meets the criteria specified in the view. For example, the following statement alters the view to prevent using an update against the view to lower the price of one of the products below 50.

```
ALTER VIEW dbo.vwExpensiveProducts
AS
SELECT ProductID, ProductName, UnitPrice,
       dbo.Products.CategoryID, CategoryName
FROM dbo.Products INNER JOIN
```

```
    dbo.Categories ON
    dbo.Products.CategoryID = dbo.Categories.CategoryID
WHERE UnitPrice >= 50
WITH CHECK OPTION
```

Having executed that alteration of the view, the following update statement will cause an error.

```
UPDATE dbo.vwExpensiveProducts
SET UnitPrice = 30
WHERE ProductID = 9
```

In Office Access, you are not limited to linking only to tables in SQL Server databases. You can also link to views and then work with them in the same way that you work with a linked table. When you select a view in the **Link Tables** dialog box, Office Access presents a list of the columns returned by the view and asks you to pick one that uniquely identifies each row. You can cancel this step and the link will still be created but the data will be read-only. To be able to modify the data, you must select a unique record identifier, or you can run an Office Access query to create a unique index on the linked view, as was demonstrated in the previous section, "Adjusting Dynaset Behavior."

One commonly used SQL Server strategy is to remove user permissions on base tables, and instead give them permissions on views that can restrict the columns and rows that the users can work with.

## View Limitations

Despite the advantages of working with SQL Server views, there are several limitations to be aware of. The most obvious limitation is that you must use valid Transact-SQL syntax. If you are migrating a number of complex Office Access queries to views, you need to know the many areas where Office Access SQL and Transact-SQL differ. For example, the wildcard characters used with LIKE are % and \_, not \* and ?. Also, single quotation marks—not double quotation marks and pound signs—are used to delimit both literal strings and dates. In addition, many of the built-in functions for manipulating strings and dates are different. Instead of using IIF to create conditional expressions, you use CASE statements. If you are moving your database from Office Access to SQL Server, you need to learn Transact-SQL.

Unlike saved SELECT queries in Office Access, views do not support parameters. You can get behavior similar to parameterized SELECT queries with table-valued user-defined functions in SQL Server, but you can't link to functions.

You also cannot sort views by adding an ORDER BY clause. Like tables, the order of rows in views is undefined and should be specified in queries that select rows from views. A common workaround for this limitation was to use SELECT TOP 100 PERCENT when defining the view. Sorting is allowed with TOP so that SQL Server can determine which rows are the top ones.

In SQL Server 2000, view results were reliably returned in the order specified in TOP 100 PERCENT queries. In SQL Server 2005 this behavior has changed. ORDER BY is still

supported in views that use TOP 100 PERCENT, but the rows are not returned in that order. There is a new workaround: Instead of using TOP 100 PERCENT, you can use TOP 99999999 with any number greater than or equal to the number of rows in the result set. However, this technique is unsupported and could prove as unreliable in the next version as the old workaround did. In addition, adding this sort can hurt performance if the view is joined to other data in a query. So stick with selecting from the view and adding an ORDER BY clause to create a sort order, just as you do with linked tables.

The final limitation to using views is that an update statement executed against a view can only modify columns from one table at a time. Later in this paper, in the discussion about how to address updatability issues, you'll learn how you can work around this limitation.

## Working with Pass-Through Queries

Sometimes you need to bypass completely the processing performed by the local Office Access database engine and by the ODBC driver. You may want to create and execute Transact-SQL code directly or call a SQL Server stored procedure or user-defined function. To do this, you can use ADO code to define and execute a command object, but that may not support easy integration with your Office Access application. Instead, you can use a type of Office Access query that allows you to write Transact-SQL that is passed directly through to SQL Server without any attempt at translation. This is called an *SQL pass-through query*.

To build an SQL pass-through query in Office Access, create a new query in design view and select **Pass-Through** as the query type. This is one of three SQL-specific query types. (The other two SQL-specific types are union queries and DDL queries.) SQL-specific queries can only be created in SQL view, not by using the graphical designer. You also must set the **ODBC Connect Str** property to a valid ODBC connect string.

You are then free to write any valid Transact-SQL script code, including parameterized calls to stored procedures. You can use SQL pass-through queries in other local queries and to set the record source for forms and reports or the row source for list controls. There is, however, one important limitation: the results returned by SQL pass-through queries are always read-only. If you want to enable users to perform updates based on the data retrieved, you must write code to handle this. Pass-through queries are ideal for read-only data, such as report record sources, list control row sources, and forms that display summary information.

If you want to use a pass-through query that calls a parameterized stored procedure in an Office Access application, your application must modify the query's SQL to set the parameter values. You can do this by writing DAO code that manipulates a **QueryDef** object's SQL property, as in the following code example.

```
Dim db As DAO.Database
Dim qdf As DAO.QueryDef

Set db = CurrentDb
Set qdf = db.QueryDefs(strPassthroughQueryName)
' Set the connection string
```

```
qdf.Connect = strConnectionString
qdf.ReturnsRecords = True
qdf.SQL = "EXEC procGetCustomersByRegion 'NY' "
```

## Using Pass-Through Queries to Optimize Bulk Updates

When you perform bulk updates to linked data by executing Office Access queries, the rows are modified one at a time. The Office Access database engine executes a separate statement for each row being affected, instead of using more efficient set-based operations that affect many rows at once.

For example, you link to the **Northwind Customers** table and create the following Office Access query.

```
UPDATE Customers
SET Customers.Country = "United Kingdom"
WHERE Customers.Country="UK";
```

If you run a Profiler trace and execute this query, you may be surprised to learn what Office Access does. First, it populates a keyset with all the **CustomerIDs** of customers with U.K. as their country (fortunately, it doesn't start by selecting all customers, even though there is no index on **Country**). Then, for each row in the keyset, Office Access creates and executes two prepared statements: one to select the **CustomerID** and **Country** corresponding to the **CustomerID** in that row of the keyset, and then one to update each row.

This is a very inefficient way to perform a bulk update that could instead be accomplished by simply executing the **UPDATE** statement shown above without using dynasets. To execute that statement from Office Access in one efficient operation, you can use an ADO command object in code. Or, you can use DAO code to create or modify, and then execute, a pass-through query. In general, using ADO is best when writing VBA code in Office Access that works with server-side objects and DAO is best when writing code that works with client-side objects. Of course, you can also create and execute pass-through queries manually in the query design window.

Using code to work with SQL pass-through queries and to execute ADO commands is covered in subsequent sections of this paper that discuss techniques for creating unbound Office Access applications.

## Server-Side Performance Optimizations

Using views, pass-through queries, or code to move query processing to the server doesn't necessarily mean that you are getting optimum performance. You still must ensure that the query plan being used on the server is optimal. SQL Server provides several tools to help you do this.

In the SQL Server query editors in both SQL Server 2000 and SQL Server 2005, you can view the query-execution plan in a graphical form that shows the different steps and their impact on performance. This allows you to spot operations such as full-table or clustered-index scans that

hurt performance. By modifying the design of the query, you can try to maximize the use of efficient joins and index-based seeks. This may require you to add new indexes to the database.

Indexes are a double-edged sword. They often increase the efficiency of retrieving data, but they also add overhead to data modifications, because the indexes must be maintained. The proper mix depends on the pattern of usage for your application.

SQL Profiler allows you to save a set of trace results to a file or to a table. You can let SQL Server analyze this workload and recommend design changes—mostly, changes to indexes—that will improve performance. Of course, it is very important for the trace to include typical usage with a typical quantity of data. You can also submit individual queries for analysis.

Windows Performance Monitor is another tool that can be useful in diagnosing SQL Server bottlenecks, because SQL Server exposes quite a few performance counters. For example, high CPU utilization could indicate that you aren't getting good reuse from your query plans, because query plan creation is very CPU-intensive.

Another factor to consider is the transaction isolation levels used in your queries and the impact this has on database locking. By default, SQL Server uses the READ COMMITTED isolation level, which ensures that data being modified is never retrieved until the modification is complete. This guarantees that you will never see data that ends up being rolled back by a transaction, but it also creates locks that can slow performance when there is a lot of database activity. If you are comfortable taking the data as it is, even if some of it may still be in the middle of a transaction, you can specify the READ UNCOMMITTED isolation level or use the WITH NO LOCKS hint in your query. SQL Server 2005 introduces a new isolation level option, called SNAPSHOT, that copies the latest committed values to a temporary database named tempdb. Enabling the SNAPSHOT isolation level creates some extra overhead, but it allows you to get data that is not still being worked on without relying on locks.

If your performance problems are primarily related to reporting and analysis, consider using SQL Server Analysis Services to create multidimensional databases. Doing this not only provides fast responses to complex queries, but also a more business-friendly way of inspecting the data. In addition, you can use data mining algorithms to identify patterns and make predictions.

Finally, consider purchasing better hardware. Unfortunately, it is often easier to get budget for consultants to tune your queries than for new servers or more memory. But hardware improvements often provide the highest performance return on investment.

## **Forms-Based Performance Optimizations**

You may find that you need to change your Office Access forms so that fewer rows are fetched and fewer recordsets are active. Because of the multiple connections and network traffic required by dynasets, forms that contain many subforms may be slow to load and may consume more resources than necessary.

A common example is the use of multiple subforms on tab control pages. Instead of keeping all the subforms active, you can use the tab control's **Change** event to detect the control's value, which indicates the page that is active. You can then load only the subform required for that page. One approach is to use a single subform control and to change its **SourceObject** property to load different subforms as different tab pages are selected.

When you work with potentially large result sets, it is best to guide the user toward selecting filter criteria before retrieving data. This can dramatically improve application performance. There is rarely justification for retrieving more than a few hundred records at a time. Even in combo boxes, you can require the user to type a couple of characters before populating the list, or to make a selection in one combo box before seeing the available choices in another related list that is filtered based on the first selection.

## Understanding and Addressing Updatability Issues

After migrating native Office Access tables to linked SQL Server tables, you may find that some of your Office Access queries, or even some of your tables, are no longer updatable. You also may encounter updatability issues with linked SQL Server views.

To diagnose and fix these problems, you need to understand the capabilities that Office Access relies on when performing updates. As discussed previously in "Using Pass-Through Queries to Optimize Bulk Updates," all Office Access updates are based on dynasets. Office Access needs a set of unique keys for any rows being updated. Office Access also needs a way to verify that the rest of the data in each row to be updated hasn't changed since the data was retrieved. In addition, you must contend with the fact that, unlike the Office Access database engine, the SQL Server database engine assumes that a single update statement can only modify columns in a single table. The Office Access database engine has a different set of rules affecting what types of queries can be updatable.

### Specifying a Unique Index

If you find that a linked table or view is not updatable, see if the link has been assigned a unique index in Office Access. To do this, open the linked table in design view (click **Yes** in the dialog box that asks if you want to continue even though some design properties of linked tables can't be modified). You should see the familiar primary key icon to the left of a column or set of columns in the table. If you don't see this, Office Access isn't able to open dynasets that include this linked table.

In "Adjusting Dynaset Behavior," earlier in this paper, you learned how Office Access selects this index and how you can execute SQL statements in Office Access to designate a unique index. If the link is to a table, you can also create a new unique index on the server and recreate the link. For views, you can recreate the link and select one or more columns in the dialog box that appears after you select the view. It is not necessary to create an index for the view on the server.

### Supporting Concurrency Checks

Probably the leading cause of updatability problems in Office Access–linked tables is that Office Access is unable to verify whether data on the server matches what was last retrieved by the dynaset being updated. If Office Access cannot perform this verification, it assumes that the server row has been modified or deleted by another user and it aborts the update.

There are several types of data that Office Access is unable to check reliably for matching values. These include large object types, such as **text**, **ntext**, **image**, and the **varchar(max)**, **nvarchar(max)**, and **varbinary(max)** types introduced in SQL Server 2005. In addition, floating-point numeric types, such as **real** and **float**, are subject to rounding issues that can make comparisons imprecise, resulting in cancelled updates when the values haven't really changed. Office Access also has trouble updating tables containing bit columns that do not have a default value and that contain null values.

A quick and easy way to remedy these problems is to add a **timestamp** column to the table on SQL Server. The data in a **timestamp** column is completely unrelated to the date or time. Instead, it is a binary value that is guaranteed to be unique across the database and to increase automatically every time a new value is assigned to any column in the table. The ANSI standard term for this type of column is *rowversion*. This term is supported in SQL Server.

Office Access automatically detects when a table contains this type of column and uses it in the WHERE clause of all **UPDATE** and **DELETE** statements affecting that table. This is more efficient than verifying that all the other columns still have the same values they had when the dynaset was last refreshed.

The SQL Server Migration Assistant for Office Access automatically adds a column named **SSMA\_TimeStamp** to any tables containing data types that could affect updatability.

Note that data types that cause updatability problems can also cause problems when included in keysets. If Office Access fails to find a matching value on the server, it assumes that the row has been deleted. When choosing unique identifiers, pick columns with values that can be matched reliably.

## Overcoming Query Updatability Limitations

Both SQL Server and the Office Access database engine have updatability limitations that affect updates performed against queries that use joins.

In SQL Server, the main limitation is the rule that only the columns from one table can be updated by a single **UPDATE** statement. This affects linked views that join data from multiple tables. An earlier discussion of views used the following example.

```
CREATE VIEW dbo.vwExpensiveProducts
AS
  SELECT ProductID, ProductName, UnitPrice,
         dbo.Products.CategoryID, CategoryName
  FROM dbo.Products INNER JOIN
         dbo.Categories ON
```

```
    dbo.Products.CategoryID = dbo.Categories.CategoryID
WHERE UnitPrice >= 50
```

This view exposes five columns, four from the **Products** table and one, **CategoryName**, from the **Categories** table. Both of the following statements will execute successfully in SQL Server.

```
UPDATE dbo.vwExpensiveProducts
    SET ProductName = 'Super Prime'
    WHERE ProductID = 9;
UPDATE dbo.vwExpensiveProducts
    SET CategoryName = 'Carnage'
    WHERE ProductID = 9;
```

However, the following statement will raise an error.

```
UPDATE dbo.vwExpensiveProducts
    SET ProductName = 'Super Prime', CategoryName = 'Carnage'
    WHERE ProductID = 9;
```

The same query in Office Access, if using native Office Access/Jet tables, would execute without a problem. So it is possible that your Office Access application may rely on the fact that columns in multiple tables can be updated in one operation. For example, if you have a form bound to a query with bound controls that have control source columns in multiple tables, the form will no longer be as updatable if you migrate the query to a SQL Server view. Users will get an error if they try to modify columns that aren't in the same underlying table. To work around this, you can take advantage of a type of trigger that can be applied to views: an INSTEAD OF trigger.

SQL Server supports two types of triggers, both of which can run code when an insert, update, or deletion occurs. The most common type of trigger, which has been around since the early days of the product, is an AFTER trigger (also called a FOR trigger). AFTER triggers can only be applied to tables.

INSTEAD OF triggers, which were introduced in SQL Server 2000, can be created for tables or views, and they are especially useful with views. As the name implies, these triggers don't run after the original insert, update, or delete operation; they run instead of that operation. Like an AFTER trigger, an INSTEAD OF trigger can roll back the transaction if necessary, and it has access to both the original and proposed row values in two virtual tables named "deleted" and "inserted," respectively.

The following is an example of an INSTEAD OF trigger on a view named vwExpensiveProducts, which allows columns from both the **Products** and **Categories** tables to be updated.

```
CREATE TRIGGER dbo.trExpensiveProductsUpdate
    ON dbo.vwExpensiveProducts
    INSTEAD OF UPDATE
AS
    UPDATE dbo.Products
    SET dbo.Products.ProductName = inserted.ProductName,
        dbo.Product.CategoryID = inserted.CategoryID,
```

```

        dbo.Products.UnitPrice = inserted.UnitPrice
FROM dbo.Products JOIN inserted
    ON dbo.Products.ProductID = inserted.ProductID;
UPDATE dbo.Categories
SET dbo.Categories.CategoryName = inserted.CategoryName
FROM dbo.Categories JOIN inserted
    ON dbo.Categories.CategoryID = inserted.CategoryID;

```

One limitation in this trigger is that it doesn't support updates to the **ProductID**. If the **ProductID** in the inserted table has been modified, the joins won't work. One way to work around this limitation in SQL Server 2005 is to take advantage of the new **ROW\_NUMBER()** function to create a row identifier in the view that won't be affected by changes to the **ProductID**. The following shows how you could alter the view to add a new durable **RowID**.

```

ALTER VIEW dbo.vwExpensiveProducts
AS
    SELECT ROW_NUMBER() OVER (ORDER BY ProductID) AS RowID,
           ProductID, ProductName, UnitPrice,
           dbo.Products.CategoryID, CategoryName
FROM dbo.Products INNER JOIN
    dbo.Categories ON
        dbo.Products.CategoryID = dbo.Categories.CategoryID
WHERE UnitPrice >= 50

```

Another problem in the trigger is that it allows changes to both the **CategoryID** and the **CategoryName** columns. This could lead to unexpected results, because changing the **CategoryID** assigns the product to a different category, but changing the **CategoryName** modifies the name of the original category. One way to resolve this is to roll back the transaction and raise an error if both **CategoryID** and **CategoryName** are updated. Another strategy would be to alter the view to include **CategoryID** twice—once for the **Category** table, and once for the **Products** table—but that could also be confusing.

The following code alters the trigger to use the new **RowID** for handling updates to **ProductID**, and it prevents updates to both **CategoryID** and **CategoryName**. This code uses the built-in **UPDATE** function to check which columns were modified. As an alternative, you could use the **COLUMNS\_UPDATED()** function to get a bitmask indicating which columns were updated.

```

ALTER TRIGGER dbo.trExpensiveProductsUpdate
    ON dbo.vwExpensiveProducts
    INSTEAD OF UPDATE
AS
    UPDATE dbo.Products
    SET dbo.Products.ProductID = inserted.ProductID,
        dbo.Products.ProductName = inserted.ProductName,
        dbo.Products.UnitPrice = inserted.UnitPrice
    FROM inserted INNER JOIN deleted
        ON inserted.RowID = deleted.RowID
        INNER JOIN dbo.Products ON
            dbo.Products.ProductID = deleted.ProductID
    IF UPDATE(CategoryID) AND UPDATE(CategoryName)
    BEGIN
        RAISERROR ('Cannot change both CategoryID and CategoryName

```

```

        at the same time', 16, 1)
    ROLLBACK TRAN
END
ELSE
    UPDATE dbo.Categories
    SET dbo.Categories.CategoryName = inserted.CategoryName
    FROM dbo.Categories JOIN inserted
    ON dbo.Categories.CategoryID = inserted.CategoryID;

```

This new version of the query allows the category name to be updated, affecting all products with that **CategoryID**. It also allows any of the **Product** columns to be updated, including a product's **CategoryID**. It doesn't allow potentially confusing updates to both **CategoryID** and **CategoryName**.

When working in a linked Office Access application with queries that are processed by the local Office Access database engine, you may also run into updatability issues that are unrelated to SQL Server. For example, Office Access does not allow queries that include aggregate expressions to be updatable, whether connected to SQL Server or to native Office Access tables. For a good summary of the most common updatability issues encountered in Office Access queries with tips on how to work around them, see the Knowledge Base article [How to Troubleshoot Errors that May Occur When You Update Data in Access Queries and in Access Forms](#).

Office Access queries against linked SQL Server tables can be updatable even if they do not select the column or columns in a table's unique index. Office Access will get the column values it needs to execute the updates. However, you will not be able to enter new rows in datasheets or forms bound to those queries. If you need to be able to add new rows to a linked table or view, include the unique index columns in the query's SELECT clause.

## Addressing Application Logic and Coding Issues

After migrating your data to SQL Server, you may find that some of your application logic and VBA code no longer function as they did. This section discusses how you can address several common issues.

### Data-Type Incompatibilities

Boolean values are stored in native Office Access tables using the same two values used for Boolean variables in VBA, 0 and -1. However, Office Access Boolean columns (Yes/No columns) are usually migrated to SQL Server bit columns, which use 0 and 1, not -1. If you have code or query criteria that rely on True being represented by the numeric value -1, you will encounter logic errors. You can either use a **smallint** column in SQL Server, which can store -1, or find and fix the logic that depends on True being -1.

Office Access supports a **hyperlink** data type that stores up to three pieces of data: the address, the text to display, and a screen tip to display when the user hovers over the hyperlink. These three pieces of data are stored in a Memo column delimited by pound signs, with a special

attribute that indicates that the column is a hyperlink. The Office Access user interface detects these columns and handles them differently from regular Memo fields. However, when you migrate the data to SQL Server, the special attribute is gone and all that remains is the pound-sign-delimited data. Office Access application user interfaces won't provide the expected behaviors when displaying the hyperlink data. To work around this, you could use unbound hyperlink controls to display the data or you could store the hyperlink data in native Office Access tables.

SQL Server 2005 adds support for **varchar(max)**, **nvarchar(max)**, and **varbinary(max)** data types, which are easier to work with than the **text**, **ntext**, and **image** data types supported in prior versions. However, Office Access OLE Object data will not be updatable in forms that use the Bound Object Frame control if the bound column has a **varbinary(max)** data type in SQL Server. Instead, use the older **image** data type if you need the data to be updatable.

Watch out for code or queries that make assumptions about how date/time values are stored. Office Access and SQL Server use different ways of storing date/time values. The built-in date functions should work correctly, but if your application uses custom logic based on assumptions about how date/time values are stored as numbers, that logic will fail. Also, watch out for the use of values for dates that are outside the supported range of SQL Server dates, which is smaller than the supported range in Office Access.

## Default Values

In forms bound to native Office Access tables or queries, default values defined in the tables appear as soon as the user navigates to the new record. However, if the tables are linked to SQL Server, any defaults defined on the server aren't added until the new row is submitted to the server for insertion. If your application depends on having these default values present, define them locally as properties of the bound controls.

## DAO Code

If your application contains DAO code that updates and inserts data using recordsets, specify the **dbSeeChanges** option, along with **dbOpenDynaset** when you open the recordset. For more information, see the Knowledge Base article [ACC2000: New SQL Records Appear Deleted Until Recordset Reopened](#).

Unless you are working with DAO to manipulate local objects, such as **QueryDef** objects or form **Recordset** and **RecordsetClone** objects, you should consider rewriting your code to use ADO, which is more efficient for working with SQL Server data and database objects.

## Setting Connection Properties

Many Office Access applications include code for relinking tables by resetting their **Connect** property values. When you are working with ODBC links, these connect strings can be based on defined data sources, called DSNs (defined data-source names), created and stored by Windows in files or in the registry.

The Office Access graphical tools for creating ODBC-linked tables and pass-through queries require you to select or create a named ODBC DSN when specifying a connection. But this is not required. Instead, use code to set these properties using "DSN-less" ODBC connect strings.

One strategy is to use a startup login form that collects login data from the user and that constructs and caches both an ODBC connect string and an OLE DB connection string to use in ADO code. The following example code creates connection strings based on selections made in a login form, using the Microsoft SQL Native Client OLE DB Provider and ODBC driver that were released with SQL Server 2005.

```
Select Case Me.optgrpAuthentication
Case 1      ' NT authentication
  mstrOLEDBConnection = "Provider=SQLNCLI;" & _
    "Data Source=" & Me.txtServer & ";" & _
    "Initial Catalog=" & Me.txtDatabase & ";" & _
    "Integrated Security=SSPI"

  mstrODBCConnect = "ODBC;Driver={SQL Native Client};" & _
    "Server=" & Me.txtServer & ";" & _
    "Database=" & Me.txtDatabase & ";" & _
    "Trusted_Connection=Yes"

Case 2      ' SQL server authentication
  mstrOLEDBConnection = "Provider=SQLNCLI;" & _
    "Data Source=" & Me.txtServer & ";" & _
    "Initial Catalog=" & Me.txtDatabase & ";" & _
    "User ID=" & Me.txtUser & _
    ";Password=" & Me.txtPwd

  mstrODBCConnect = "ODBC;Driver={SQL Native Client};" & _
    "Server=" & Me.txtServer & ";" & _
    "Database=" & Me.txtDatabase & ";" & _
    "UID=" & Me.txtUser & _
    ";PWD=" & Me.txtPwd
End Select
```

The following example cycles through all of the tables in a database and resets the connection properties for all the ODBC-linked tables, assuming that they all are linked to tables or views in the same database. The code sets three properties for each linked table: the name of the link, the name of the source table (or view), and the connect string.

```
Dim fLink As Boolean
Dim tdf As DAO.TableDef
Dim db as DAO.Database
Set db = CurrentDb
For Each tdf In db.TableDefs
  With tdf
    ' Only process linked ODBC tables
    If .Attributes = dbAttachedODBC Then
      fLink = LinkODBCTable( _
        strLinkName:=.Name, _
        strConnect:= mstrODBCConnect, _
        strSourceTableName:=.SourceTableName)
```

```

        End If
    End With
Next tdf

Private Function LinkODBCTable( _
    strLinkName As String, _
    strConnect As String, _
    strSourceTableName As String) As Boolean

    ' Links or relinks a single table.
    ' Returns True or False based on Err value.

    Dim db As DAO.Database
    Dim tdf As DAO.TableDef

    On Error Resume Next
    Set db = CurrentDb

    ' Check to see if the table link already exists;
    ' if so, delete it
    Set tdf = db.TableDefs(strLinkName)
    If Err.Number = 0 Then
        db.TableDefs.Delete strLinkName
        db.TableDefs.Refresh
    Else
        ' Ignore error and reset
        Err.Number = 0
    End If

    Set tdf = db.CreateTableDef(strLinkName)
    tdf.Connect = strConnect
    tdf.SourceTableName = strTableName
    db.TableDefs.Append tdf
    LinkTableDAO = (Err = 0)
End Function

```

It is generally best to use **DAO.CreateTableDef** for linking tables, instead of using **DoCmd.TransferDatabase**, because you have more control over the properties of the link. Also, if you need to create a link to a table or view that does not have a unique index (knowing it will therefore not be updatable), using **TransferDatabase** will cause a dialog box to open asking the user to specify a unique index. Using **CreateTableDef** doesn't cause this side effect.

One technique for making connection strings available throughout the lifetime of your applications is to expose them as public properties of the login form and then hide the login form instead of closing it. When the application exits, the form closes and no credentials are persisted.

If you are using SQL Server Authentication and storing user names and passwords with your links, it is safest to delete the links when the application exits. When the application starts up, code in your application can delete any existing links (in case there was an abnormal shutdown) and then create new links, retrieving SQL Server table names from a local table. For pass-through queries, you can delete the connection data without having to delete the entire query. Here's an example of code that cycles through all SQL pass-through queries to do this. At

startup, you would need to reset the connect property for each of the queries based on credentials the user enters in a login form.

```
Dim qdf As DAO.QueryDef
Dim db As DAO.Database
Set db = CurrentDb
For Each qdf In db.QueryDefs
    If qdf.Type = dbQSQLPassThrough Then
        qdf.Connect = "ODBC;"
    End If
Next qdf
```

## Creating Unbound Office Access Applications

One of the most convenient and empowering features in Office Access is the way that it enables users to view and update data in forms that are bound to data through a query. However, the need to maintain an active dynaset for the form usually requires several open connections and continual network traffic. For applications that require maximum scalability and minimal consumption of server resources, using standard bound Access forms is often not a viable option.

Instead, you can create Office Access applications that connect to the server on an "as-needed" basis to execute stored procedures that fetch small numbers of records and perform data modifications. Unbound applications can use forms to display data and to accept user input, but these forms are not bound to live server data and therefore do not hold locks or other resources on the server. Such applications require much more coding than typical Office Access applications, but they can potentially scale to support thousands of users.

### Populating Lookup Tables with Pass-Through Queries

Many applications use lookup tables that contain relatively static data, often used to populate combo boxes. It is inefficient to retrieve the same data over and over again. Instead, you can run an efficient pass-through query to populate a local Office Access table and use the data in the table. You can delete the data and repopulate the table as needed.

Begin by creating a pass-through query and setting the **Returns Records** property to **Yes**. Type the Transact-SQL statement that selects data from a SQL Server table, view, or user-defined function.

```
SELECT DISTINCT Country FROM dbo.Customers ORDER BY Country;
```

You can also execute a stored procedure that returns records.

```
EXEC procCountryList
```

Create an Append query that selects from the saved pass-through query and populates the local table.

```
INSERT INTO CountriesLocal ( Country )
```

```
SELECT qrysptCustomerCountryList.Country
FROM qrysptCustomerCountryList;
```

You can then base combo boxes and list boxes on that local table.

Write DAO code to refresh the data by deleting the rows in the local table and executing the append query.

```
Dim db As DAO.Database
Set db = CurrentDb

' Delete existing data
db.Execute "DELETE * FROM CountriesLocal"
' Run the append query
db.Execute "qappendCountries"
```

## Query-by-Form Techniques

You can use a series of combo boxes to present a cascading list of values for a user to choose from. For example, a user could select a customer name in order to populate a second combo box that displays orders for only the selected customer, instead of all orders in the table. When the user selects a particular order, only that order is loaded.

The following PassThroughFixup procedure modifies a **QueryDef** object by passing in parameter information.

```
Public Sub PassThroughFixup( _
    QueryName As String, _
    Optional SQL As String, _
    Optional Connect As String, _
    Optional ReturnsRecords As Boolean = True)

    ' Modifies pass-through query properties
    ' Inputs:
    '   QueryName: Name of the query
    '   SQL: Optional new SQL string
    '   Connect: Optional new connect string
    '   ReturnsRecords: Optional setting for ReturnsRecords--
    '                   defaults to True (Yes)

    Dim db As DAO.Database
    Dim qdf As DAO.QueryDef

    Set db = CurrentDb
    Set qdf = db.QueryDefs(QueryName)
    If Len(SQL) > 0 Then
        qdf.SQL = SQL
    End If
    If Len(Connect) > 0 Then
        qdf.Connect = Connect
    End If
    qdf.ReturnsRecords = ReturnsRecords
    qdf.Close
```

```
Set qdf = Nothing
End Sub
```

Once the pass-through query has been modified by having its SQL property set, the **RowSource** on a combo box can be set to the name of the pass-through query to requery the list.

```
Me.cboOrders.RowSource = "grysptOrderListbyCustomer"
```

The **AfterUpdate** event of one combo box can use code like this to populate the row source of another combo box or the record source of a subform.

## Caching Server Data in Local Tables

When working with forms that display one record at a time, it is relatively easy to write code that fetches a single row of data and populates unbound controls on the form. Similarly, you can write code to execute a single update, insert, or delete statement for data modifications, passing values from controls to query parameters.

However, data displayed in a datasheet or in a continuous form cannot be bound dynamically at run time the way data in a single-record form can, because there is an indeterminate number of rows. This often affects applications that need to display one-to-many relationships in forms and subforms. One solution to this problem is to fetch the data and cache it in a local table, similarly to the way you fetch and cache lookup data for combo and list boxes. On an Order form, the **OrderID** would be passed as a parameter to load the order-detail line items associated with that order, as the following sample code demonstrates by calling the `PassThroughFixup` procedure shown in the previous code sample. The pass-through query is then executed when the append query runs, populating the lookup table with fresh data.

```
' Clean out OrderDetailsLocal
CurrentDb.Execute "Delete * From OrderDetailsLocal"
' Get the current order detail records using
' an append query based on a pass-through query.
strSQL = "EXEC procOrderDetailSelect " & lngOrderID
Call PassThroughFixup( _
    "grysptOrderDetailsForOrder", strSQL, _
    Forms!frmlogin.ODBConnect)
CurrentDb.Execute "qappendOrderDetailsForOrder"

Me.fsubOrderDetail.Requery
```

## Managing State

To help ensure that users only perform valid actions, it is best to change a form's available user interface based on the state of the data. For example, a form containing modified data should have **Save** button enabled. After saving data changes, the **Save** button would be disabled and a **New** button enabled to allow the form to be cleared for entry of a new record. Since the data in the form is unbound, you can't rely on Office Access to do the work the way it does with its own built-in navigation buttons.

Instead, handle the form's state in code by creating your own **IsNew** and **IsDirty** properties. Write code that ensures that these properties are maintained in accordance with the form's state and ensures that additional code "fixes up" the controls on the form based on the **IsNew** and **IsDirty** values. Maintaining form state in this way gives your form the appearance of a "bound" form and prevents the user from making mistakes. For example, **Save** and **Cancel** buttons are enabled only if data in the form has been edited and the **IsDirty** property is set to **True**. When the **Save** and **Cancel** buttons are enabled, the **New** and **Delete** buttons are disabled.

```
Public Sub FixupButtons()  
    Select Case IsDirty  
        Case True 'Dirty  
            cmdNew.Enabled = False  
            cmdSave.Enabled = True  
            cmdDelete.Enabled = False  
            cmdCancel.Enabled = True  
            cmdClose.Enabled = False  
        Case False 'Not Dirty  
            Select Case IsNew  
                Case True 'Not Dirty, New  
                    Me.FirstField.SetFocus  
                    cmdNew.Enabled = False  
                    cmdSave.Enabled = False  
                    cmdDelete.Enabled = False  
                    cmdCancel.Enabled = False  
                    cmdClose.Enabled = True  
                Case False 'Not Dirty, Not New  
                    Me.FirstField.SetFocus  
                    cmdNew.Enabled = True  
                    cmdSave.Enabled = False  
                    cmdDelete.Enabled = True  
                    cmdCancel.Enabled = False  
                    cmdClose.Enabled = True  
            End Select  
        End Select  
    End Sub
```

## Validating Data

An important part of an unbound application is validating data. You want to ensure that only valid values are submitted to SQL Server. Triggering server-side errors increases network and server load, so it's best to try to handle as many errors on the client as you can. Validation is best handled in a single validation routine that is called when the user clicks the **Save** button. The validation routine visits every control, checking the data against a set of rules, packaging up all violations into a single string variable that can be presented to the user. Creating this code can be tedious and does not eliminate the need to handle data errors raised by the server. In addition, users prefer learning about problems all at once, rather than piecemeal. It's much easier to fix all problems in one pass than to fix one and then be prompted to fix the next one.

## Saving Data with Stored Procedures

Once data has been validated, values can be passed to a stored procedure that performs updates on the server inside of an explicit transaction. Explicit transactions allow you to perform multiple operations as a single unit of work, as shown in the following example that performs updates to both the **Orders** and **Order Details** tables in the **Northwind** database. Either both update operations succeed, or both are rolled back, so the data is always left in a consistent state. One good strategy is to marshal order-detail line items as a semicolon-delimited string, or as XML, that will be unpacked in the stored procedure code on the server so that updates to both the **Orders** and **Order Details** tables can be submitted in a single transaction. The client code creates a **Command** object and populates parameters that match the signature of the stored procedure performing the updates.

This code and the stored procedure use an integer **ConcurrencyID** value that is based on a **ConcurrencyID** column in the **Orders** table. The **ConcurrencyID** value is incremented by the stored procedure code whenever the order or any of its details is modified. This allows the application to prevent users from making "blind" updates to data that has been changed by another user since editing began. However, you must ensure that data is modified only through the stored procedure or the **ConcurrencyID** will not be incremented and your application will not be able to detect a potential blind update condition.

```
Set cmd = New ADODB.Command
With cmd
.ActiveConnection = myConnection
.CommandText = "procOrderUpdate"
.CommandType = adCmdStoredProc

.Parameters.Append .CreateParameter( _
    "RETURN_VALUE", adInteger, adParamReturnValue)
.Parameters.Append .CreateParameter( _
    "@OrderID", adInteger, adParamInput, , Me.OrderID)
.Parameters.Append .CreateParameter( _
    "@CustomerID", adVarChar, adParamInput, 5, Me.CustomerID)
.Parameters.Append .CreateParameter( _
    "@EmployeeID", adInteger, adParamInput, , Me.EmployeeID)
.Parameters.Append .CreateParameter( _
    "@OrderDate", adDBTimeStamp, adParamInput, , Me.OrderDate)
.Parameters.Append .CreateParameter( _
    "@OrderDetails", adVarChar, adParamInput, 7000, strDetails)
.Parameters.Append .CreateParameter( _
    "@ConcurrencyID", adInteger, adParamInputOutput)
.Parameters.Append .CreateParameter( _
    "@RetCode", adInteger, adParamOutput)
.Parameters.Append .CreateParameter( _
    "@RetMsg", adVarChar, adParamOutput, 100)

.Execute
```

The design pattern of the stored procedure is to define input parameters for the values that are being submitted along with two output parameters to return success/failure information to the client so that client code can branch accordingly. The order-detail line items are passed as a delimited string. The *@ConcurrencyID* parameter is defined as an output parameter so that the incremented value can be passed back to the client. Although defined with the **OUTPUT**

keyword, output parameters in SQL Server are actually input/output parameters and can accept data passed to them.

```
CREATE PROCEDURE [dbo].[procOrderUpdate](
    @OrderID int,
    @CustomerID nchar(5) = NULL,
    @EmployeeID int = NULL,
    @OrderDate datetime = NULL,
    @OrderDetails varchar(7000) = NULL,
    @ConcurrencyID int OUTPUT,
    @RetCode int = NULL OUTPUT,
    @RetMsg varchar(100) = NULL OUTPUT)
```

The code in the body of the stored procedure, much of which has been omitted here for brevity, validates the input parameters. As each parameter is validated, the **@RetCode** and **@RetMsg** variables accumulate information which can then be passed back to the client if any of the validations do not pass muster. The following code fragment checks the value of the **ConcurrencyID** in the **Orders** table. If the **ConcurrencyID** has changed, the **RETURN** statement unconditionally exits, passing back the **@RetCode** and **@RetMsg** to the client. The **@RetCode** value indicates success/failure so that the client code can branch accordingly, and the **@RetMsg** value can be displayed to the user or logged.

```
SELECT @CheckOrder = ConcurrencyID FROM Orders
WHERE OrderID = @OrderID

IF @CheckOrder <> @ConcurrencyID
BEGIN
    SELECT @ConcurrencyID = @CheckOrder,
        @RetCode = 0,
        @RetMsg = 'Another user updated this order ' +
            'while you were editing it.'
    RETURN
END
```

The delimited string containing the order-detail line items can be parsed into a temporary table or table variable and validated. If there is an error parsing the string and creating the temporary table, a **RETURN** statement exits the stored procedure, returning the error code and message to the client.

Once all of the input parameters have been validated and the temporary table created for the line items, an explicit transaction can be started. The first **UPDATE** statement in the transaction updates the **Orders** table and increments the **ConcurrencyID**. The transaction is rolled back on any errors and the **RETURN** statement exits the procedure, returning information to the client.

```
BEGIN TRAN
UPDATE Orders SET
    CustomerID = @CustomerID,
    EmployeeID = @EmployeeID,
    OrderDate = @OrderDate,
    ConcurrencyID = @ConcurrencyID + 1
WHERE
```

```

        OrderID = @OrderID AND ConcurrencyID = @ConcurrencyID

-- Check if update succeeded.
SELECT @CheckOrder = ConcurrencyID FROM Orders
    WHERE OrderID = @OrderID

IF @CheckOrder = @ConcurrencyID + 1
    SELECT @RetCode = 1
ELSE
    SELECT @RetCode = 0,
        @RetMsg = 'Update of order failed.'

-- If order update failed, rollback and exit
IF @RetCode = 0
    BEGIN
        ROLLBACK TRAN
        RETURN
    END
END

```

If the first **UPDATE** statement succeeds, the second operation updating the **Order Details** table is initiated. First, all existing rows matching the **OrderID** are deleted from the **Order Details** table. This is more efficient than attempting to determine which order details have been changed, inserted, or edited since the last update.

```

DELETE [Order Details]
    WHERE OrderID = @OrderID

```

The final section of code in the stored procedure inserts the line items from the temporary table into the **Order Details** table and commits the transaction.

```

INSERT INTO [Order Details]
    SELECT @OrderID, ProductID, UnitPrice, Quantity, Discount
    FROM #tmpDetails

-- Test to see if all details were inserted.
IF @@ROWCOUNT = @DetailCount AND @@ERROR = 0
    BEGIN
        COMMIT TRAN
        SELECT @RetCode = 0,
            @RetMsg = 'Order number ' +
                CONVERT(VarChar, @OrderID) + ' updated successfully.'
    END
ELSE
    BEGIN
        ROLLBACK TRAN
        SELECT
            @RetCode = 0,
            @RetMsg = 'Update failed. Order Details couldn't be saved.'
    END
RETURN

```

The code in the client application then processes the results of the stored procedure, displaying information to the user about whether the order was updated successfully. If a concurrency error occurs indicating that the server data was changed by another user, the code can let the user

choose to see the current data or to overwrite it by resubmitting the update with the new **ConcurrencyID**. Additional code then adjusts the form state to synchronize the controls on the form to reflect the state of the data.

Attempting to create such code-intensive unbound applications isn't right for every developer or for every application, but when you need to support large populations of users or large amounts of data, it can enable Office Access to be as efficient and scalable as any development platform. For added maintainability and code reuse, much of the data access code can be moved to middle-tier objects that handle all communications with the server and that can be shared with other applications. Your Office Access code would then instantiate and work with methods and properties of the middle-tier objects. Another alternative is to consider migrating the application to the Microsoft .NET Framework, which offers excellent support for working with disconnected data and middle-tier objects. For many Office Access developers, however, the advantages of remaining in the Office Access environment are worth the work required to create unbound forms when necessary.

## Conclusion

Security or scalability requirements often motivate Office Access developers to consider migrating their data to SQL Server. In addition, Office Access is often used to work with data that has been stored in SQL Server all along. Using ODBC-linked tables in Office Access enables Office Access applications to continue benefiting from the convenience and versatility of the client-side Office Access database engine. By understanding how Office Access interacts with SQL Server, and by taking steps to move as much query processing to the server as possible, developers can take advantage of SQL Server features while continuing to create rich applications quickly with Office Access.

© 2009 Microsoft Corporation. All rights reserved. [Terms of Use](#) | [Trademarks](#) | [Privacy Statement](#)