Apress™
Books for Professionals by Professionals™

Sample Chapter: "Planning the Upgrade"
(pre-production "galley" stage)

From Access to SQL Server

Migrate your Access and Jet Databases to SQL Server

by Russell Sinclair
ISBN # 1-893115-24-0

CHAPTER 3

Planning the Upgrade

When an application is migrated from Microsoft Access to Microsoft SQL Server, it is usually done using one of two methods. The first type of migration is a database migration. It involves migrating the backend Access database to SQL Server while maintaining the application interface in Access. The other type of migration involves migrating the interface to another development environment, such as Visual Basic, as well as migrating the backend database to SQL Server. The second type of migration is considerably more complex as it involves two migrations and extensive knowledge in each area. In this book, we will only be looking at the first type of migration, migrating the backend to SQL Server. However, the lessons learned in this type of work can easily be applied to a complete migration of the application and data. The steps involved are very similar, and the planning that is necessary in each type of migration is very important.

Before you begin your migration, you must create a plan. Your plan should address a couple of key issues. It should address how SQL Server data will be accessed from your application. Your plan should also address what objects will be migrated and how. All of this information should be thoroughly documented, so that you go into the migration with a good idea of what has to be done and how much work is involved. Without this analysis, the migration can easily fail and the credibility of the application could be jeopardized.

Planning How Data Will Be Accessed

Before any part of the migration can take place, you should decide how you plan to access data from SQL Server from your application. Although you can make modifications to the data-access methods as the migration progresses, deciding how data will be accessed has a pronounced effect on what development will be necessary and how you should plan the migration.

Data Access Methods

The method that you use to access data can change depending on where and how you are using it. This includes whether the data needs to be bound to objects in the Access application, such as forms and reports. The data access method you choose also depends on how you plan to use SQL Server data in code or macros.

Any analysis should take into account what object model will be used for programming and what the capabilities are of the version of Access you are using.

There are two main methods used to connect to a SQL Server database from Access: ODBC and OLE DB.

ODBC

Open Database Connectivity (ODBC) is a standard protocol for accessing relational database systems, such as Microsoft SQL Server, Oracle, or Sybase Adaptive Server. It is designed so that computers can connect to a SQL data source and work with the data from that system. In Access 97, when using data from another relational database, most data manipulation takes place through ODBC. In fact, if you want to bind non-Jet data to Access objects in Access 97, you must use ODBC. ODBC binding is still supported in Access 2000.

When you connect to ODBC data in Access, Jet retrieves the data using one of two methods. If the data has a unique index that Access is aware of and the data is updateable, Access retrieves the requested fields for the current record (with the exception of text and image fields, which are downloaded when they are accessed) and also retrieves the fields for 100 records surrounding the current record to make browsing the data faster (this can be configured through a Registry setting). It also generates bookmarks for the entire recordset based on the unique index in use.

> **NOTE** *The unique index that Access uses is not necessarily the primary key. Access selects the first unique index on a table in order of index name and uses the values in that index to generate the bookmark values. This knowledge can be very useful because you can force Access to select a particular unique index in a table by naming it, so that it will be sorted (alphabetically) before all others. This trick can be handy if you want to force Access to select a unique index built on a single field over another index that is built on more than one field.*

When your application is idle, or as you browse the records, Access queries the details of other records in the database. This method of retrieving data allows much faster access to immediately visible records and reduces the overhead to access any other records. If the data does not have a unique index that Access can use or is not designed to be updateable (through security or some other definition), Access retrieves all of the fields and rows requested. This ODBC data is returned as a snapshot-type recordset. This type of recordset cannot be updated. Storing and working with the entire recordset in Access does require more memory to be used locally, but the benefit is that Access does not maintain a constant connection to the server. Because the connection to the server does not need to be maintained in order to work with the recordset, network traffic is reduced. Only the initial

transfer of the data takes place over the network. The traffic associated with a connected recordset, where the client is requesting data from the server regularly, is not generated.

Access provides a number of ways of using data through ODBC. You can link ODBC tables directly into your Access database much as you would a regular Access table. With this type of connection, you use the ODBC data through a Jet interface. The Access Jet engine maintains some of its own indexing on the connected data in order to allow the data to be navigated and updated. This means that you are really using two different interfaces to the data: the Jet engine connection to ODBC and the ODBC connection to the server. This type of connection can be used in linked tables or linked views, and the data in the linked ODBC table can be manipulated using Data Access Objects (DAO).

You can also manipulate ODBC data by directly connecting to the ODBC data source and bypassing the Jet engine. This type of ODBC data manipulation can be accomplished through the use of SQL pass-through queries or through the use of ODBCDirect workspaces. SQL pass-through queries are queries in which you can enter a statement that is passed directly to the ODBC driver without the Jet engine validating it or parsing it in any way. When working with linked ODBC data, Access parses a statement locally to translate it for the ODBC driver and determines how much of the statement can be sent to the server. Access also checks the syntax of the statement to ensure that it is valid. SQL pass-through queries bypass this feature of Access. However, there is a drawback to using pass-through queries. Although pass-through queries can return data, that data is always read-only to the Access application. Another method of connecting directly to the server and bypassing Jet's parsing functions is to directly modify data in code by using an ODBCDirect workspace. This is essentially a wrapper around the ODBC driver to allow you to use DAO-like code to modify ODBC data. It can be very useful if you want to perform quick data manipulation and have the server handle all of the work.

OLE DB (ADO)

OLE DB is a new Microsoft standard for accessing data in the enterprise. It goes beyond ODBC in that it is a standard for accessing relational and nonrelational data stores. This means that you can use OLE DB to access not only a SQL database, but also any data store for which an OLE DB driver can be or has been created. These data stores include relational databases, file directories, mainframe data, and email systems. A Microsoft Access programmer cannot directly access OLE DB. Instead, a wrapper called ActiveX Data Objects (ADO) is used to manipulate OLE DB data. This wrapper is basically a programming framework that makes all of the low-level functionality of OLE DB available to you without requiring you to possess the knowledge of what is going on at the system level.

ADO provides some functionality that ODBC does not. A useful capability of ADO data allows you to disconnect from the server once data has been retrieved from the client. This data can then be used locally as if you were still connected, but the network communication that is normally required becomes unnecessary. When processing of the data is complete, the data can then easily be reconnected to the data source and submitted to the server, so that the changes are submitted. ADO also provides the capability to run commands against the server asynchronously (without stopping code execution). When such a command is created, ADO can notify the application when processing is complete. This can be very useful for long running commands that would otherwise freeze the application. ADO also provides one function that ODBC and Jet have never been able to provide: the ability to create recordsets on the fly. Recordsets that are defined on the fly can be filled with data and then added to a data store or saved to disk with only a little added code. This additional capability to database programming of creating datasets on the fly will revolutionize the industry.

ADO and OLE DB can be used in an Access database in a number of ways. In Access 2000, you can use ADO code to modify data local to the database in which the code is running, and you can use ADO code to manipulate data in any other OLE DB data source. Access 97 does not have native support for OLE DB—all data connections use the Jet engine natively; therefore, code is the only place where you can use ADO. And, in order to use ADO with Access 97, you need to install the Microsoft Data Access Components (MDAC) and the Development Kit (DA SDK) available at `http://www.microsoft.com/data`. In Access 2000, the default programming model for database information is ADO, so you can use it without having to add any references to your application. Also, the connections that Access makes to SQL Server in a database project actually use the OLE DB driver for SQL Server to make the connection. This means that ADO and OLE DB are much more tightly integrated into the environment.

Analyzing Objects

Once you understand how Access can connect to other databases, you can proceed with analyzing the objects in your database for the migration. Take the time and analyze each object in the database. You can dramatically improve the performance of your application if you can migrate as much functionality as possible to SQL Server. Let's take a look at all of the objects as well as what you should be looking for when you plan the migration.

Tables

Any migration from Access to SQL Server must start with an analysis of the tables that make up the database. After all, they contain all of the information that you need in your application. There are a number of items you should consider when analyzing the tables: data organization, data types used, indexes, defaults, rules, and relationships. You should take as much time as possible to ensure that you do not miss any items because table migration is where the greatest performance gain can be achieved.

Data Organization

Each table should be analyzed to determine how the data is organized. All organization of data should be logical and resistant to problems as the database grows. Apply all rules for sound database design before the migration takes place. The topic of good database design is well beyond the scope of this book. Tomes have been published on this subject, and your best bet is to purchase one of these books if you are unfamiliar with the rules of good database design. There are many books available on this subject, but my personal preference is *Database Design for Mere Mortals* by Michael Hernandez (Addison-Wesley, January 1997). It is designed for anyone who is unfamiliar with the concepts of normalization, and it is not specific to any RDBMS. If you are looking for something a little more advanced without getting extremely technical, try *Handbook of Relational Database Design* by Fleming and von Halle (Addison-Wesley, August 1988).

It is important to make sure that your database is in at least third normal form. Problems that can be papered over in a Jet database will become major issues as your database expands. Large SQL Server databases are much less forgiving concerning database design than small Access databases. The level of normalization you choose is dependent on the purpose of your database. Operational databases must be in third normal form to ensure that the database will function properly. If your database is only used for data warehousing or reporting, then first normal form is all that needs to be satisfied.

Data Types

The wizards that are available to upsize a database from Access to SQL Server have their own translation algorithm for creating a SQL data type equivalent to an original Access data type. However, the translation the wizards choose is not always appropriate for your use of the data type. There are a number of items to watch for in your data type conversion when you move to SQL Server. In order to make informed decisions that will lead to a successful migration, you should know the

various data types that SQL Server makes available to you and the sizes and limitations of those types.

Table A-1 in Appendix A contains a complete list of all the SQL Server data types. You can see from this table that there are many more data types in SQL Server than in Access. Most of these types are simply larger versions of data types that Access uses and are reasonably simple to understand. However, one data type requires special attention: the timestamp data type. A timestamp field is not a field that you edit, and contrary to its name, does not hold time or date data. Instead, it is a system field for which SQL Server generates values to uniquely identify each row in a database at a single state in the data composition.

Timestamp values are always unique within a table. When you update data in a row in SQL Server and a timestamp column is in the same row, SQL Server changes the timestamp to a new value. When a user updates the data in a table, the timestamps for the affected rows are also updated. This allows Access and SQL Server to easily determine if the data has been modified by the fact that the timestamp has changed. If a second user is updating one of the same records that the first user already updated, they will get an error telling them that the data is no longer valid because the record has changed.

The timestamp is automatically updated by SQL Server when a change is made to the record. If two users "take out" a record at the same time, the value of the timestamp field comes with it. When the data is returned to SQL Server and modifications have been made, the value in the submitted timestamp field is compared with the value still in the table. If they match, the update is allowed to go through. If the first user updates the record, the timestamp changes. When the second user submits changes, the timestamp in the submitted record does not match the timestamp in the table; therefore, a nasty error message occurs. When the second user attempts to commit the changes to the record, the update fails because the application tells SQL Server to update a record that contains a particular timestamp value. This value no longer exists because the first user has modified the record before the second user could commit his changes.

SQL Server prevents you from creating more than one timestamp column per table. Because the timestamp data type is designed more for the use of the data provider than for users to view, you will only need one timestamp column per table. It is recommended that you give this column the name "timestamp" (this is the only occasion when I will tell you to name a field the same as its data type). The reason for this is, if you open an ODBC linked table in design view, the data type of the field will appear as a binary field. Because you want to ensure that you don't attempt to work with this field as you would other fields, you should name it "timestamp," so that you will always know what the purpose of the field is. Access handles concurrency in tables without timestamps by comparing each field in the table to its original value. Adding a timestamp field to a SQL Server table forces applications, such as Access, to use this column to enforce concurrency without having to compare each field in the table. Only the unique index Access uses to

identify records and the timestamp are compared against the table to enforce concurrency. Adding a timestamp field to frequently updated tables can dramatically improve the performance of your application when applying updates.

You should ensure that the data types you use in SQL Server are large enough to hold all possible values for the field in the future. Autonumber fields in Access (called Identity fields in SQL Server) are always of type Long Integer. This data type, equivalent to SQL Server's *int* data type, can hold whole number values between -2,147,486,648 and 2,147,483,647. Although this may sound like a very large range of numbers, you must keep in mind that not all numbers are used, especially in Access. Some numbers may not be used due to failed or canceled updates or if records are deleted. SQL Server also skips numbers for the same reasons. If an update fails or if a record is deleted, an identity number becomes unavailable for use. Identity columns in SQL Server can be of any Integer data type (except bit), so you are not limited to the int data type (equivalent to Long Integer in Access) when creating identity columns. You must set their seed value (starting ID) and the number that will be used to increment them. By selecting a data type that is appropriate for the data requirements of the table, you can either reduce the storage space necessary for the field in the parent and any related tables, or choose a larger Integer data type that will ensure that you do not run out of numbers any time in the near future.

Text data types should also be looked at to ensure that your data types are sufficient for the application they will be used in. In Access, any Text field over 255 characters must be a Memo field. Text and Memo fields in Access are variable length, meaning that the data only occupies as much space as is necessary. In SQL Server, varchar and text data types are both variable length data types. However, the char data type is always fixed-length and always occupies the same amount of disk space. If you do not enter enough characters to fill the char column, it will be padded with spaces when the data is stored in the table. When running comparisons on a text field, you should be aware of whether or not it is a char field because the extra padding may cause comparisons to fail. The advantage to using the char data type is that it is easier for SQL Server to work with than varchar is. Varchar requires extra processing from SQL Server to handle the fact that the length can vary. SQL Server performs better when using a char column. This performance gain only applies to small fields. In fields that are greater than approximately six characters, SQL Server handles variable length fields much better. If you require a small text field, use the char data type if possible. For larger fields, use varchar.

You can also make improvements on how precise your data is with SQL Server, or reduce the precision if high precision is not required. SQL Server decimal and numeric data types allow you to set the scale of a field (the number of digits right of decimal) with more flexibility than Access. If your application deals with statistical data and requires a specific degree of precision, you can use this characteristic to set exactly how precise your data will be. You can also choose larger data types,

such as float (scale cannot be modified, set at 53 digits), to store your data if your numbers are larger or if you require more precision. Access, on the other hand, is limited to the Single and Double data types, which have variable precision but a fixed maximum and minimum size. If you want to ensure that your SQL Server data conforms to the data rules required by your application, setting the scale and precision of the numeric data you use can help to fulfill these rules.

Indexes

In Access, indexing is implemented using one of a few methods. You can set indexing directly by setting a field's Indexed property in the field properties in the table design to Yes (No Duplicates) or Yes (Duplicates OK). You can create indexes in the Indexes property sheet in table design view. And, you can have Jet create indexes by specifying a foreign or primary key. All indexes in a table should be analyzed to determine if they are needed. Indexes are not required on fields that are not often used as the basis for criteria in searching, on fields that are not related to fields in other tables, or on fields that are not used to sort the data in the table or in queries.

SQL Server indexes have a characteristic that is not revealed to a user in Access tables: clustering. A clustered index is the index that a table is sorted by on disk. Each table can have only one clustered index. When SQL Server needs to access data in an index, and that index is not clustered, SQL Server must look at an index that is stored off-table. Off-table indexes contain pointers to the location of their data in the parent table. This means that searching a nonclustered index is slower than accessing a clustered index because the server must look at the index and follow the pointers back to the original data. In Access, the primary key is always a clustered index. In SQL Server, you can choose which index is to be clustered, and it doesn't have to be a unique index. If your data is most commonly accessed by a nonprimary key, you should set the index on that field to clustered because it will improve the access time for that field.

You should also ensure that all of your indexes are necessary and logical. Adding an index to a table adds overhead for updates to the table. Each time a record is inserted, SQL Server must maintain all of the indexes. You should check all of the indexes on your table and ensure that they do not add unnecessary overhead to your application. You may require an index for any number of reasons: criteria are regularly used against the column to search for or update data; data needs to be sorted by this column often in forms or reports; or the column is a primary key. Indexes can benefit an application in any of these situations. However, you must balance the need for faster performance when querying the database against the need for faster performance when updating data. Searching data should receive greater consideration because it is usually more cumbersome and can span large numbers of records. In nonindexed columns, SQL Server must scan the entire table when you search on that column. This is in contrast to indexed columns,

where SQL Server can quickly move to the first match in the index and just retrieve the matching data from that point on. But keep in mind that if you have too many indexes on a single table, you could make updates to the data extremely slow. In a reporting database, lots of indexes are good, but in an operational databases, fewer indexes are better because of the maintenance overhead.

Defaults and Rules

Defining default values in Access is quite simple. In fact, if created using the user interface, all of the numeric data types in Access automatically define a default value of zero when you create a numeric field (a feature that can be very annoying for those of us who don't want it to happen). In SQL Server, defaults are just as easy to set and can be configured in one of two ways: added to the field as a constraint or created as a Default object and linked to a field. The constraint method is the better choice if the default value is not complex and is not reused in many tables. In cases where the default is complex and must be applied to multiple fields or if you need to bind it to a user defined data type, use a Default object.

A problem arises when you use defaults for fields in SQL Server and bind the fields to an Access form. Access, when working with Jet data, shows the default value in a form even before the record is added to a table, saving the user from mistakenly thinking that the value will be Null. When using SQL Server, the default values for the fields are not shown on an Access form until the record is saved. This may cause the user to think that they must enter data in the field at all times. If a field is not used in the data entry form, you should set the default on SQL Server if you want to prevent the field from being set to Null when a record is inserted.

Rules in Access can be defined against a field or against a table. Rules are defined by setting the Validation Rule property for the fields against which they are to be applied. Each field can have only one validation rule as can each table. Validation rules can be very complex if the application requires them to be, but they are usually fairly simple. Validation rules in Access are limited to 2048 characters in length.

In SQL Server, you can add multiple rules or CHECK constraints to a field or to a table and make them as complex as necessary. Again, you should ensure that the items you add are necessary because they add overhead to your tables. Each time a record is modified SQL Server must run the rules against changed fields. If you put the data validation in the application itself, the amount of time that is required to update the record is reduced once it is sent to SQL Server, and the user receives immediate notification of the violation. When a rule is violated on SQL Server, the validation does not run until the whole record is submitted. You must balance the importance of having a single point of control in SQL Server against user considerations.

Relationships

Relationships in any database can become quite complex. Access is very good with how it manages relationships in that you can create enforced relationships with cascading updates and/or deletes. In SQL Server, this is not possible. You can either create relationships that prevent the insertion of unrelated records or deletion of parent records while child records exist, or you can maintain the integrity of data in related tables through the use of triggers. As described in Chapter 1, triggers are Transact-SQL code that runs in response to data being modified in a table. They can be used to validate changes to a record, cascade updates and deletes, or even maintain data in derived fields or in other tables. Triggers can be defined to run in any or all of three possible situations: record deletion, addition, or modification.

The type of enforcement you use in your relationships depends on the design of your tables. If you use identity columns as the primary keys on your tables, you often do not need cascading features. Cascading an update would be useless because you cannot update the identity column. However, you may want to add cascading deletes to your table, in which case, you cannot define a relationship and you must create a cascading trigger. In SQL Server, relationships always take precedence over triggers and therefore a delete on an enforced relationship will fail because the relationship will prevent the delete and, as a result, the trigger will never fire. In this case, you should remove the relationship and enable the cascading delete by creating a DELETE trigger. This trigger will be fired each time a record is deleted from the table.

Similarly, updating a column that has an enforced relationship will fail if there are related records for that row in another table. If you want updates to cascade to other tables, you must create an UPDATE trigger to make the modifications for you. One of the benefits of SQL Server triggers is that you have the ability to determine whether particular columns were modified in an insert or update. This means that you can shorten the processing SQL Server needs to do by forcing the trigger to only execute commands when certain columns are updated. SQL Server simply executes the trigger to the point where the cascading fields are singled out. If the field being updated is not in this list, the processing of the trigger does not go any further. This allows you to guarantee referential integrity without taking unnecessary performance hits.

When defining relationships you may also want to determine if your primary key in the parent table can be improved upon. Integer data types are by far the best fields to use for primary keys because the data that is stored in them is more readily handled by the computer processor, and they are always the same size and length, regardless of their value. This is in contrast to textual data that can change length or precision numeric data types that can also vary depending on the data they contain.

You can also use Integer data types to increase the speed of those relationships where multiple fields are related in different tables. Multiple field relationships require a great deal of extra processing to handle joins. Linking the tables involved in the relationship requires the RDBMS to search an index that contains a large

amount of data. To avoid this problem, add an identity field to your table, and then use that as the primary key. You can still create a unique index, and even cluster it if you like, on the multiple fields, maintaining your requirements. With the new primary key, queries that join the table to other tables based on this key will perform much faster, and you will be storing less data in the related table, reducing the size of your database.

Queries

Migrating access queries to SQL Server can make dramatic improvements in an application. Determining the correct way to migrate a query depends on the purpose of the query and the purpose, in turn, affects how it should be evaluated for migration.

If your query is used as the source for a form and the data the query returns needs to be updateable, the best way to migrate the query is to move it to a SQL Server view. Views in SQL Server allow you to define a SQL statement that returns records and can be linked into an Access project as a view (Access 2000). It can also be linked into a standard database as if it were a table when using ODBC. Views can include multiple tables in their definition, but only one of those tables can be updated at a time. Attempting to update data for more than one base table in a single record in a view will raise an error from SQL Server. However, views can be very useful for retrieving and organizing data. If your application does not require all of the fields in a table, you can use a view to retrieve only the fields you need. This reduces the amount of data that must be transferred across the network. It also allows Access to work with the data more efficiently because it does not have to track fields that will not be modified.

If your query is used as a source for reports or drop-down lists, you may be better off using stored procedures and/or SQL pass-through queries to get the data, rather than using a view. Stored procedures can be very powerful. One major advantage to using stored procedures is that you can run multiple data manipulation functions in one procedure. For example, suppose you had a sales database that required some data for a report. The data for this report must come from multiple unrelated tables to merge the data for the different sales types. In Access, you would have to create multiple queries to retrieve the data into a temporary table. Each of these queries would have to be called independently through code. In SQL Server, you can run a single stored procedure that creates a table in the tempdb database, appends the necessary data from each source table to the new temporary table, and then returns the data in the temporary table to the client. The temporary table only lasts as long as the data is in use. Once the report is closed, the table is deleted. The main advantage to using a stored procedure under these circumstances is that you only need to call one stored procedure. This greatly simplifies the coding on the client-side and reduces the number of permanent objects

required in the database. Using SQL pass-through queries, Access can call the stored procedure and the pass-through query itself can be bound by the report.

When planning the migration of queries to SQL Server, you should watch for queries that use Access or user defined functions in their definition. Visual Basic functions that are used in queries cannot be directly migrated to SQL Server. Instead, you should try to find equivalent functions in SQL Server, create stored procedures, or use SQL statements that implement the same functionality. For example, Access domain functions, such as DMax and DMin, do not have equivalents in SQL Server. Results similar to the results provided by these functions have to be derived from SQL statements in a view or with stored procedures that create the same functionality. Some aggregate functions do have equivalents in SQL Server. The Access Jet functions Max and Min can be replaced with SQL Server MAX and MIN functions where necessary, however, you should check the SQL Server documentation to ensure that their implementation achieves the results you need.

User defined VBA functions cannot be directly migrated to SQL Server. In many cases, the functionality they implement cannot be directly duplicated. In such cases, it may be better to retrieve the necessary information from SQL Server, and then use the VBA functions against the data once it has been passed back to Access. Caution should be exercised with such functions, however, as working with data in this manner can be quite cumbersome and can lead to bottlenecks in the application. If you call the function directly in a query based on a linked table and you use the function itself to filter the resulting data, the Jet engine will assume that none of the processing of the query can be run by SQL Server, and it will retrieve the entire contents of the table before running your function to filter the data. This means that the entire table is returned to the client. Network load is increased, and the performance of your application suffers.

In some cases, you can create equivalent functionality in Transact-SQL stored procedures. Running stored procedures on the server instead of using VBA functions can help you increase the response time of the application because any processing that takes place on the server before the application receives the data increases the speed at which the data you require is summarized. SQL Server is much faster at handling data requests than Access is, and if you can make the server do the work, you reduce the time it takes for the data to be presented to the user.

One function of Access SQL that is not supported in SQL Server's version of SQL is cross-tab queries. A cross-tab query derives its columns not from the columns in the tables it takes its data from, but creates a column for every unique value in one of the table's columns. The SQL statements that you use in Access to create these queries are not valid for SQL Server. Creating a cross-tab query in Access from SQL Server data generally means that you must reduce the records returned by SQL Server to the absolute minimum before sending them to Access, and then create a local cross-tab query to create and summarize the data in the desired format.

Forms

When evaluating a form for migration, the record source should be analyzed. If you base all of your forms on queries, you may have already completed some of the analysis for this migration in your query analysis. If your forms are based on SQL statements or tables, you should evaluate what data is being retrieved and how that data is retrieved.

In Access 2000, you have much more versatility with form data sources than you do in Access 97. Using a database project or ADO code in Access 2000, you can base a form's data on a table, a view, a stored procedure, or a SQL statement. In Access 97, your best choice on bound forms is to use views or tables linked in through ODBC when working with remote data servers. Forms bound to stored procedures are read-only because a SQL pass-through query must be used to run the stored procedure (and pass-thru queries are always read-only). Choosing the appropriate object to supply the data to your form can be very complex. When you make a choice as to which object type you will use, you should keep a few concepts in mind.

If a form is always opened with supplied criteria in Access 2000, you will probably want to use a stored procedure because stored procedures can accept arguments, whereas views cannot. For example, if you normally open a Customer Details form with only one customer shown at a time, you can use a stored procedure that takes a Customer ID and returns only the supplied customer. Consider the following stored procedure:

```
Create Procedure spGetCustomer (@intID int)
/* retrieves only one customer id */
As
   BEGIN
        SELECT * FROM tblCustomer WHERE CustomerID = @intID
        RETURN @@error
    END
```

For now, it is not really necessary to understand how to write a store procedure or how this particular stored procedure works. You only need to know how it is called. To run this stored procedure in SQL and have it retrieve information for a customer with CustomerID 12, you would use the command *Exec spGetCustomer 12*. There is a benefit to using a stored procedure that is written this way. Supplying the procedure with a Null value as the parameter returns an empty recordset to which data can be added. Therefore, setting a form's record source to *Exec spGetCustomer NULL* returns a new blank record that a form can use as a base to add more data. Because the amount of data retrieved is small, network traffic is reduced.

If your form is used only to display data and does not directly modify it, it is best to use a stored procedure. However, if you filter the data in the form regularly and different columns are used to filter the data each time that the stored procedure is called, you may have to create multiple stored procedures to handle the different situations.

If you are using Access 97 with ODBC, your choices are reduced. The only updateable connections to SQL Server are linked tables or views. Because of this limitation, any form that updates data must be linked to one of these objects. However, it is almost always better to use a view, especially if your form does not show all of the fields in a table. Using a view to retrieve only the necessary fields can reduce the amount of network traffic and reduce the memory requirements on the client because less data is handled.

Controls on forms that display data in lists (such as list boxes or combo boxes) provide opportunities to make performance improvements. The list of items in these objects does not often change. When the list is static and small, it is best to use a nonupdateable stored procedure (called through a pass-through query if using ODBC) to return the required data. Creating a stored procedure to retrieve the values in the list allows you to transfer some of the processing of data to the server, and using a nonupdateable recordset allows Access to avoid some of the overhead that updateable recordsets require. If a list is long, you may want to return only some of the top values from the database unless the user requests more. For example, you could have a drop-down list of customers that, by default, lists only the top 10 values and a value called "More…". When a user selects the "More…" item, you could run code that requeries the stored procedure and tells it not to limit the list. The following stored procedure would accomplish just this in the Northwind sample database included with SQL Server 7:

```
CREATE PROCEDURE spGetCustomers (@bitLimit bit)
AS
BEGIN
    IF @bitLimit = 0
        SELECT CustomerID, CompanyName FROM Customers
        ORDER BY CustomerID
    ELSE
        SELECT TOP 10 CustomerID, CompanyName FROM Customers
        UNION SELECT 'ZZZZZ', 'More...'
        ORDER BY CustomerID
END
```

Calling this stored procedure with 1 as the parameter value tells the stored procedure to return the top 10 with a "More…" item added to the end of the list with a SQL UNION statement. Calling the same stored procedure with 0 as the parameter value returns all values in the list. The only detail you must ensure is

that the "More…" is displayed last in the list. This can be done by creating a false sort in the stored procedure on the server (as was done in the previous code) or in code by dynamically adding an entry after the data has been returned to the client. When the user selects the "More…" item from the list, you can force the list to change to a full list of customers, as in the following code on a combo box:

```
Private Sub cboCustomers_Change()
    If Me!cboCustomers = "More..." Then
        'Change the RowSource to another pass-through
        'that lists all customers
        Me!cboCustomers.RowSource = "qptCustomersAll"
        Me!cboCustomers.Requery
        Me!cboCustomers.Dropdown
    End If
End Sub
```

Normally, the RowSource property for this combo box is a pass-through query that calls spGetCustomers with the bit parameter set to 1. When the user selects the item "More…" from the list in the combo box, the Change event fires, and the source of the data in the list changes. The combo box is then requeried, so that the new data is shown, and the list is dropped-down, so that the user knows that they can now select a different item.

Code behind Forms

Event-driven code behind forms should be thoroughly analyzed when planning a migration. Much of the functionality that needs to be programmed into forms to ensure data integrity can be moved to SQL Server. The most common migration technique is to move functionality in the Before Update and After Update events to SQL Server triggers. For example, if you have a need to update the quantity of a product on hand in the Inventory table whenever an item is ordered and quantities are entered into the OrderDetails table, in Access, you would accomplish this through code or macros in form Update events. A form will call the necessary code to update the quantity on hand and ensure that the totals in the Inventory are correct. This kind of update has one major failing, which is when a user updates the data directly in a table, the code in the form will not be executed. Because the update to the Inventory information is only fired when the data is changed in the form, entering data directly into the table would break the consistency of the data and could cause problems in reporting or in the ability to deliver the product to a client because the quantity of product on hand would not be correct. In SQL Server, adding a trigger to the OrderDetails table that fired an update to the Inventory table whenever the quantity of a product order changed would force any update to the data to trigger the update on the quantity on hand in the Inventory table. In this case, it does not matter where the OrderDetails table is updated, in a form, in code,

or directly in the table, the trigger will always run. You can also use triggers to prevent updates that do not satisfy your own criteria, thereby enforcing business rules for the application. Look for code in your forms that performs these kinds of actions and plan to move the functionality to SQL Server.

In planning the migration of your forms, you should also look for any code that directly accesses the database or uses DAO or ADO to update data. The functionality that these procedures implement can often be successfully migrated to SQL Server. For example, you may have a master/detail form where the items in the subform cannot exist without the master form data. In this case, you want all the subrecords deleted if the parent record is deleted. Although you can accomplish this functionality in Access through cascading deletes, it is often run through form events instead of through cascades. This is because enabling automatic cascading deletes is risky. You want to ensure that this type of cascade only occurs from the form in question. However, when you do this from your Access forms for SQL Server data, you should encapsulate the detail data deletion into a stored procedure that takes the master primary key information as a parameter. This reduces the amount of network traffic because all of the records are deleted with a single command. This allows SQL Server to manage most of the deletion and enforces any rules against the data that are necessary without having to implement the same functionality in two places. It also makes the system more secure by forcing deletions to occur in a controlled manner.

Reports

Reports are probably the easiest item in a database to analyze for migration because there is very little that needs to be done. The main portion of a report that needs to be analyzed is the report record source. It is almost always easiest to migrate the record source definition to a stored procedure that returns records. There is only a small amount of analysis that is really necessary. The main feature to watch for is if your reports are often opened using a specific criteria. For example, if you want to print a customer record, the main filter you need to apply against the data is a customer ID. Creating a stored procedure that takes the CustomerID as its sole parameter can speed the previewing and printing of a report because SQL Server will precompile the stored procedure such that it retrieves the data in the fastest possible manner. It also allows you to use complex SQL functions, such as nested queries or multiple SQL statements. The record source should be analyzed in much the same way as queries are and the same caveats should be kept in mind.

Macros

It is common for Access applications to use macros that update data. Because of this, you need to analyze all of your macros and determine what they do. Look for

any actions that interface with data, especially those that modify data. The most important actions executed in a macro to watch for are ApplyFilter, FindRecord, OpenQuery, OpenTable, and RunSQL because they are all used against data. Any of these actions in a macro work with or modify data in such a way that their functionality may degrade the performance of an Access or a SQL Server application, so each action that takes place should be carefully analyzed. The most important one, RunSQL, directly executes a SQL statement against a database and should be analyzed using the same criteria as a query. Moving macros to stored procedures can often accomplish the same task with less work, especially when the macro contains multiple steps to accomplish a complex task. Stored procedures can include as many commands as necessary, and they will benefit from the increased performance of SQL Server. You will also be reducing the network traffic because you will only make one call to the stored procedure as opposed to running multiple functions on the data through macros.

Modules

Much of the processing that takes place in database modules is designed to perform detailed functions against data from a database. The most critical code to look for is code that performs any data processing using DAO or ADO, especially those routines that use multiple tables and/or recordsets to achieve a business goal. The type of SQL Server object you will use to replace this code depends on what the purpose of the routine is, how many database objects it accesses, and whether or not it needs to modify data.

Developers often create VBA modules that retrieve data from the database, and then navigate that data to update other tables or perform some other complex function against the data. This record processing code often takes the form of the following pseudocode:

```
Set rst = cnn.Execute(“Select ….”)
While not rst.EOF
    If rst(“f1”) = “some value” Then
        rst(“f2”) = “some other value”
    End If
    rst.MoveNext
Loop
```

Because this type of procedure is mostly data oriented, it should be moved to SQL Server by creating stored procedures that achieve the same functionality. You may not have access to some functions that VBA provides, but there is almost always a way to achieve similar functionality in Transact-SQL. To this end, you should familiarize yourself with the various Transact-SQL functions that are available by looking at the Books Online. As part of creating the stored procedure, you should

consider eliminating this record-by-record processing. Even though you can navigate individual records by defining special Transact-SQL variables called cursors, it is not the best solution. Typically, you get better performance from bulk update commands than you do from cursors. You also get less code, which can reduce the number of bugs as well as your maintenance costs. You should attempt, as best you can, to use pure SQL in your functions. Many of the functions developed in VBA that navigate individual records can be rewritten to use pure SQL with a bit of time and effort.

You should look at the code and determine whether it will be able to stay as it is in the migration or if changes will be necessary. Some of the most common places where changes are required are in the connection information to the database and the type of recordsets that are used. You should be conscious of whether the recordsets you use are being updated or if they are just read-only data used for other purposes. If you do not need to update the data, tell the server that this is the case by using a read-only recordset. The server will understand that it does not need to track any changes to the data that you make, and that you will not be returning the data at any time. This reduces network traffic and server load. This technique should be used wherever possible. Also, you cannot open table type recordsets on ODBC or OLE DB data sources because these are only supported for Jet tables.

SQL strings that appear in code are a very important aspect of the migration. You should avoid SQL strings used in code as much as possible. Instead, use views or stored procedures to retrieve your data. Although this increases the number of objects in your database, it is a much better method of retrieving data for a few reasons. Saving stored procedures on the server, rather than passing SQL statements allows the server to precompile the T-SQL code before you call it. SQL Server compiles the stored procedure with an execution plan that it thinks is the most efficient plan to perform the necessary functions against the data when the object is saved. This allows SQL Server to execute the procedure without having to compile each SQL string as it is passed. As a result, the stored procedure returns the data faster than a SQL string could. The same is true of SQL Server Views. An added benefit is that saving the SQL as a stored procedure or view on the server makes the SQL more maintainable. If someone changes the name of a field or some other information on the server, it is much easier to determine what code and queries are affected by that change if the code and queries are stored on the server. The SQL Server administration utilities include a tool to automatically determine the dependencies of an object. If your SQL string is stored in an Access database and not on the server in a stored procedure or view, SQL Server has no way of notifying you of the possible repercussions of a change in the database design. In addition, rewriting code to move functionality to the server sends the more complex functions to the server and saves your application from having to call on the memory and processing power of the local client.

When analyzing your code, you should also keep in mind that SQL Server stored procedures can encompass many operations. Unlike Access queries, you can define multiple statements for a single stored procedure. Much of the code that is written for Access applications is designed to work around the fact that Access cannot handle multiple statement queries. When migrating to SQL Server, you may be able to completely eliminate some VBA procedures from your database and rewrite them as stored procedures. T-SQL has much of the procedural functionality that you need to accomplish the same tasks as your VBA code, so you should take advantage of this functionality.

General Considerations

When you analyze the objects in your database, there are a number of general considerations you should take into account. First, the more objects you can move off the client, the better the performance of the application will most likely be. Mixing objects between the server and the client can cause a lot of unnecessary overhead in the application. Mixing causes greater network traffic as data must be exchanged between these objects more often. Also, you are forcing two systems to share the work on the data when in truth, one of them is much better at it. Second, you should check the importance of the objects in the new application. Do you really need all of the objects in the new application? Do they add value to the application? Are they designed to accomplish a task in the most efficient manner? Is there a better way to implement them? Sometimes, these questions can only be answered by testing the application under different situations. Third, you should take into account that if concurrent user problems are forcing you to upsize, you may find that your current design is aggravating the problem and, as a result, moving to SQL Server by itself won't solve the problem. You should consider how a large number of users will affect your application and if the design can be improved upon, so that these users can be handled. Finally, if you are upsizing an application that ran on the client's computer, you should take into account that your application will now need constant access to a network. Failures in the network mean that your application cannot function, and your users will become very unhappy. Having good network support can mean the difference between an application that is productive and one that is wasting space on a user's hard drive.

Documentation

In any migration plan, there can never be too much documentation. Documenting your database design and migration procedures not only allows you to keep a record of the migration, it also helps you think through the various aspects of the analysis. You should begin your documentation with a general description of the migration including why you are planning to move the database to SQL Server.

This allows you to explain to others and to yourself why the migration is happening. You should include information, such as what problems currently exist, how SQL Server will solve those problems, costs involved in the old and new application, maintenance considerations, and any other information that you think is pertinent. You should then document the physical name and location of the database before and after the migration, how large the database is, how large it will be initially in SQL Server, and what application or scripts will be used to migrate the data from one database to the other. Finally, you should include thorough documentation on the migration plan for each object in the database and the purpose of each object. This will be the object documentation that you will return to as you redevelop portions of the application to take advantage of the new environment and avoid future problems. This documentation should include all of the following information and any other information you deem necessary.

Tables

- Table name and purpose
- Field names
- Field source and destination data types
- Defaults and rules
- Triggers that will be needed
- Relationships and how they will be enforced
- Indexes and their characteristics

Queries

- Query name and purpose
- Destination object type and name
- Considerations for VBA code calls
- Relationship to other objects

Forms

- Form name and purpose
- Record source for the form and its destination
- List box and combo box source data and destination object and name
- Code behind form purpose and destination if any
- Triggers that can be created from code

Reports

- Report name and purpose
- Record source for the report and its destination

Macros

- Macro name and purpose
- Migration potential and destination object

Modules

- Module name and purpose
- Description of each procedure in module
- Migration destination of procedures if any
- Areas where code needs to change for access to another system

This documentation should be used to understand how the migration work will be done and what improvements will result.

Once the documentation is complete, you should be ready to begin the migration to SQL Server.