Using SQL Databases with Dexterity – Part Two

Using SQL Views with Dexterity

Views are a logical way of looking at the physical data located in SQL tables. In fact, to a user and to Dexterity, a view looks exactly like a table. However, a view does not actually represent a physical table on the SQL Server. Instead, it is merely a window into the data. Because it is not a true physical table, you can use a view to limit the amount of data a user can see. For example, you can choose to allow users access only to selected columns from a SQL table. Views can assist in building complex reports that cannot currently be generated by the Report Writer or can filter data that is included on reports.

Creating a View in SQL

Using the following syntax, you can create a view that will restrict access to an existing SQL table to only the column names that are included in the SQL statements.

```
CREATE VIEW view_name
SELECT column_name, column_name, …
FROM table_name_A, table_name_B
WHERE table_name_A.column_name = table_name_B.column_name
```

*Like the creation of SQL tables, when creating SQL views, it is necessary to be logged into SQL Server as the database owner (dbo) of the SQL database you are working with. Generally this means that you will want to log into Microsoft SQL Server as either **sa** or **DYNSA** before executing the code that creates your view. This is because **sa** and **DYNSA** are the database owners for Dynamics-related databases.*

In order to use a view with Dexterity, when you create the view it must include the DEX_ROW_ID column from the original SQL table. This is in addition to any other columns that you will want to access. For example, the following SQL code will create a view that allows access to the Customer Number and Customer Name fields in the Dynamics RM Customer MSTR table along with the Customer Balance field from the Customer Master Summary table. Note that it also contains the DEX_ROW_ID so that it can be accessed from Dexterity.

```
CREATE VIEW CstrVW
AS SELECT RM00101.DEX_ROW_ID, RM00101.CUSTNMBR, CUSTNAME, CUSTBLNC
FROM RM00101, RM00103
WHERE RM00101.CUSTNMBR = RM00103.CUSTNMBR
```

Creating a Table Definition in Dexterity for a SQL View

In order to access an existing SQL view in your Dexterity application, a corresponding table must be defined using the Table Definition window. Any value can be entered for the Table Name and Display Name, but the Physical Name must be equivalent to the View Name that was used in SQL. In addition, the appropriate fields must be inserted to represent each column in the view. However, the DEX_ROW_ID does not need to be inserted because it will be included automatically by Dexterity. Finally, at least one key must be defined for the table. This key should be marked as the Primary key. The following is an example of a table and key definition that could be defined to access the customer view that was described previously:

Table Name:	RM_Customer_MSTR_VIEW
Display Name:	RM Customer MSTR VIEW
Physical Name:	CstrVW
Table Fields:	Customer Number
	Customer Name
	Customer Balance
Key Name:	RM_Customer_MSTR_VIEW_Key1
Key Segments:	Customer Number
SQL Key Options:	Primary
	Create Index
	Unique

Creating Stored Procedures for the View

There are stored procedures that can be used to make the communication with a view more efficient. These are the similar to the stored procedures that were discussed previously in the section called Creating SQL tables with Dexterity. A function library called Table_CreateProcedures() can be used to automatically generate all the stored procedures that can be used to communicate with a view. Sample code that includes this function library is shown in the following example. Note that the PRODID constant that is used as represents the Product ID of the third party program that you are writing.

```
local integer result.

result = Table_CreateProcedures(PRODID, table RM_Customer_MSTR_VIEW)
if result <> 0 then
       error "Stored procedures were not created successfully.".
end if.
```

The stored procedures for the view would generally be created at the same time that the third party tables are created for your application. Therefore, you would most likely want to include this code in the same global procedure that was used to create your tables. This procedure is generally attached as an installation script for product.

Granting Access to a SQL View and Associated Stored Procedures

Similar to SQL tables, the user who creates a view is the only person who can access it. In order for other users to access the view, appropriate permissions must be granted to both the view and its associated stored procedures. These permissions must be set for each of the other users on the system. They can be granted in the same way that permissions can be granted to tables. Refer to the information provided earlier for granting access to SQL tables.

Using the View

Once a table has been defined for the view, Dexterity can be used to read from this view by accessing the new table in your sanScript code or by including it on reports.

Writing back to a view through Dexterity is much more challenging than reading from a view. Therefore, is generally not recommended. Instead it is suggested to write back directly to the original SQL table that the view is associated with.

Using Pass-through SQL with Dexterity

Dexterity is designed so that it is not necessary to know SQL when creating your applications. In other words, Dexterity creates all the SQL statements that are necessary to execute your sanScript code in conjunction with a SQL database. However, there may be times when you would like to use SQL to perform operations that are not supported by sanScript. That is when pass-through SQL can be used to execute SQL statements directly from your Dexterity code.

Pass-through SQL is an alternative to using stored procedures. Generally Pass-through SQL is used when there is only one or two SQL statements that need to be executed. If there are more than that, a stored procedure should generally be used.

Another valuable use of pass-through SQL is to be able to create a stored procedure through Dexterity. For example, pass-through SQL could be used to create a stored procedure that will set proper permissions for third party tables. Therefore, not only can a stored procedure be executed through Dexterity, it can also be created from Dexterity. For more information on using pass-through SQL to create this type of stored procedure in Dexterity, refer to one of the last sections of this document called "Granting Table Permissions with Pass-through SQL".

Creating a Pass-through SQL Connection

The first step in using pass-through SQL within Dexterity is to create a pass-through SQL connection. This can be done by using the **SQL_Connect()** function. The login information for the current SQL login will be used for the new connection, and the connection will only be utilized for executing SQL statements from sanScript. Other connections will be used by Dexterity to access SQL data. The pass-through SQL connection will be active until you use the **SQL_Terminate()** function to terminate it.

Since creating a pass-through SQL connection can be time consuming, once a connection is open, it is generally recommended to leave it open until you have finished executing all of the SQL statements that you need. You can terminate the connection once all SQL statements are complete.

Executing SQL statements

Any SQL statements that you would like to execute must be placed in a text field on one of your Dexterity windows. **The SQL_Execute()** function will be used to execute the statements contained in the field and after the statements have been executed, the **SQL_GetError()** function can be used to determine if any errors were encountered.

Generally the text field that contains the SQL statements will be marked as invisible. You would only want it to be seen if you would like your users to have the ability to enter their own SQL statements.

Working with Result Sets

The SQL statements that you execute may generate one or more result sets and usually you will want to retrieve values from these result sets. The following function libraries can be used to do this:

- **SQL_GetNextResults()** – makes the next results set active.
- **SQL_FetchNext()** – moves to the next row in the current results set.
- **SQL_GetNumCols()** – returns the number of columns in the results set.
- **SQL_GetData()** – retrieves a specific column from the current row in the results set.
- **SQL_Clear()** – clears any results sets and error information to prepare the pass-through connection to be used again.

Once you have used SQL_FetchNext() to move to another row, you cannot return to a previous row. In addition, there is no way to know how many rows are in a results set. To read all rows, use SQL_FetchNext() until it returns an error 31 which indicates there are no more rows in the results set.

Pass-through SQL Example

Here is an example of using pass-through SQL to create a view in the TWO database:

```
local long SQL_connection, status.
local text SQL_Stmnts. {global text field definition}

{SQL error information}
local long GPS_error_num, SQL_error_num.
local string SQL_error_str, ODBC_error_str.

{Connect to the SQL data source.}
status = SQL_Connect(SQL_connection).
if status = 0 then

   {Build the SQL statements. Store them in a local variable.}
   SQL_Stmnts = "use TWO".
   {Execute the SQL statements.}
   status = SQL_Execute(SQL_connection, SQL_Stmnts).
   if status <> 0 then
      error "An error occurred executing the SQL statements.".
      {Retrieve the specific error information.}
      status = SQL_GetError(SQL_connection, GPS_error_num,
         SQL_error_num, SQL_error_str, ODBC_error_str).
      if status = 0 then
         warning "GPS Error: " + str(GPS_error_num).
         warning "SQL Error: " + str(SQL_error_num) + SQL_error_str.
         warning "ODBC Error: " + ODBC_error_str.
      else
         error "Unable to retrieve SQL error information.".
      end if.
   end if.
```

```
{Build the SQL statements. Store them in a local variable.}
   SQL_Stmnts = "create view SOPTRX as select * from TWO.dbo.SOP10200".
   {Execute the SQL statements.}
   status = SQL_Execute(SQL_connection, SQL_Stmnts).
   if status <> 0 then
      error "An error occurred executing the SQL statements.".
      {Retrieve the specific error information.}
      status = SQL_GetError(SQL_connection, GPS_error_num,
         SQL_error_num, SQL_error_str, ODBC_error_str).
      if status = 0 then
         warning "GPS Error: " + str(GPS_error_num).
         warning "SQL Error: " + str(SQL_error_num) + SQL_error_str.
         warning "ODBC Error: " + ODBC_error_str.
      else
         error "Unable to retrieve SQL error information.".
      end if.
   end if.

end if.
```

Working with Reports

Report Writer Query

If possible, when the Report Writer generates a report it will take advantage of a Dexterity feature called Report Writer Query. This functionality takes advantage of server side joins to drastically improve performance. Basically, whenever possible, at the time a report is generated, Dexterity will automatically create a query and thus a SQL string that represents the entire report. This contrasts the previous functionality where the Report Writer would make individual table requests in order to generate the results. This new method will automatically be used by reports under the following circumstances:

- The sort that is used for the report includes less than 7 columns or segments.
- The Ignore Case Option is not selected for any of the columns or segments included in the sort for the report.
- The combined row length of the report is less than 1963 bytes.
- The report uses all SQL tables and does not include a local temporary table.

If these conditions are not met the original functionality of the Report Writer will be used. Individual database manager requests will be made to generate the report.

Multi-Login Report Option

If your application gives users the ability to login to multiple SQL servers simultaneously, when you create a report in Dexterity you may need to take advantage of the Printing Option called Multi-Login Report. This option should be selected when multiple tables are used on a report and those tables exist on different servers. By marking this option it will prevent the Report Writer from generating a query when it gathers data for the report. It may reduce the performance of the report, but is required for the report to print correctly. This is because when the tables are located on different servers, the query that the Report Writer generates will always fail and the report will not print. Therefore, in order for the report to print, the Multi-Login Report option must be selected so that a query is not generated when the data is collected for the report.

Skip Blank Records

The Skip Blank records option on the Report Definition windows specifies where data should be included on a report if it does not have a corresponding record in a related table. For example, if table A is related to table B and Skip Blank Records is selected, when a record is encountered in table A that does not have a matching record in table B, the record will not be included on the report. In relational terminology this is called a natural join. On the other hand, if the option is not marked and a corresponding record does not exist in table B, the record will still be included on the report. In relational terminology this is called an outer join. When working with the Report Writer, natural joins are most desirable. Therefore, whenever possible you will want to select the Skip Blank Records option. This will provide the most efficient results.

SQL Debugging Techniques

DEX.INI Settings

SQLLogSQLStmt=TRUE
Including this setting in your DEX.INI file will prompt all sanScript table operations to be written to a text file called DEXSQL.LOG. If any of your code is not executing as you would expect, you can use the information that is printed to this file to help you troubleshoot the situation. A common debugging technique is to copy any lines in the DEXSQL.LOG file that are not working correctly, and using another application such as ISQL/w, try executing those same statements to determine if the results are exclusive to the Dexterity environment.

SQLLogODBCMessages=TRUE
Including this setting in your DEX.INI file will prompt all ODBC messages returned by the ODBC driver to be written to a text file called DEXSQL.LOG. Note that this is the same file used for logging SQL statements. When both statements are set to TRUE both ODBC and SQL messages printed to the same DEXSQL.LOG file. This file is created in the same directory as the application dictionary. This information can be helpful when trying to uncover more information about the errors that are occurring in your application.

Any ODBC messages that are displayed in the DEXSQL.LOG will always be preceded by the word [Microsoft].

DebugRW=1
Including this setting in your DEX.INI file will prompt the results of a Report Writer query to be written to a text file called DEBUGRW.TXT. From the information that is provided in the text file you can determine if a query was successful or not.

To help in trouble shooting problems related to the generation of reports, before printing a report you may choose to mark the Multi-Login Report option on the Report Definition window. This will force the system to perform individual table operations instead of creating a query. If you have the SQLLogSQLStmt = TRUE setting included in your DEX.INI. The individual select statements are then included in your DEXSQL.LOG file and can be analyzed to uncover any potential problems.

SQLQueryTimeout=0
Including this setting in your DEX.INI file will control the period of time Dexterity will wait for a SQL query to execute. When set to 0 it will wait indefinitely. A valid range of values for this setting are 0 to 9999. If this setting is not included, the default value is 300 seconds.

SQLProcsTimeout=0
Including this setting in your DEX.INI file will control the period of time Dexterity will wait for a SQL stored procedure to execute. When set to 0 it will wait indefinitely. A valid range of values for this setting are 0 to 9999. If this setting is not included, the default value is 300 seconds.

SQLRptsTimeout=0
Including this setting in your DEX.INI file will control the period of time Dexterity will wait for a report to generate. When set to 0 it will wait indefinitely. A valid range of values for this setting are 0 to 9999. If this setting is not included the default value is 300 seconds.

FHCheckRanges=TRUE
Including this option in your DEX.INI file will allow you to determine what types of ranges are being used within your application. This information will be written to a log file called FHRANGE.LOG.

Other Debugging Tools

SQL Trace

SQL Trace is a utility that is included with SQL Server. It allows you to monitor the activity for a given user. Using this tool you can analyze user activity, produce audit trails, and observe SQL statements generated by your application. With the Include Performance Information checkbox selected, SQL Trace will also report the amount of time it takes to execute SQL statements on a per user basis. This information can then be used to help find potential problems in your application code.

SQL Error Log

The error log is a text file that contains audit and error information generated by SQL Server. The file is called ERRORLOG and is located in the SQL Server root directory in a folder called \LOG. You can view the information in the error log by selecting Error Log from the Server menu within Enterprise Manager or by using any text editor. Scan the error log for any error messages that may have been caused by your application.

A new error log is created any time you stop and restart SQL Server. SQL Server will archive the old error logs by saving the previous six error log files using the filename ERRORLOG.x where x is 1 to 6.

Converting an Existing Application for use with SQL

Generally, an application created for Btrieve or c-tree Plus databases can also be used with a SQL database with virtually no development modifications. All Dexterity commands and functions are designed to work the same with the SQL database manager as with Btrieve and c-tree database managers. Therefore, a developer is not required to write SQL code in order to make their applications compatible with SQL data. Dexterity automatically generates all of the SQL code that is required to communicate with SQL Server.

SQL Compatibility Utility

There are a few possible compatibility concerns that you may encounter when converting your existing application for use with SQL for the first time. A SQL Compatibility Utility is provided in Dexterity Utilities to help you identify these concerns. This utility will verify certain elements of a dictionary to see if they will be acceptable in a SQL environment. The verifications include the following:

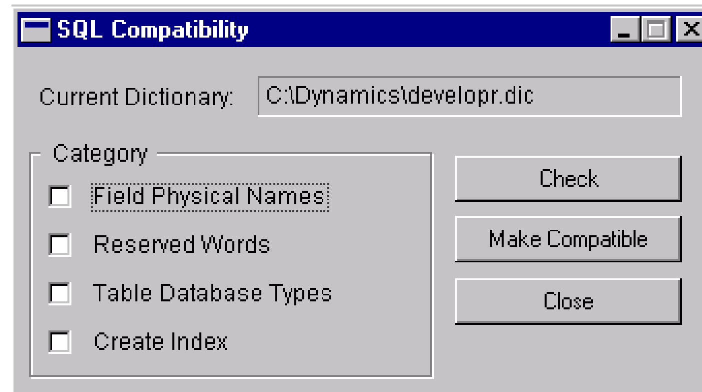
- Making sure that the SQL table limits have not been exceeded. For example, it tests the number of fields in a table definition, the record size of each table, the number of keys for each table and the number of segments in each key.
- Verifying that each table field has a valid physical name that does not include spaces.
- Determining that the physical names of table fields do not use SQL reserved words.
- Checking to see if any of the tables have been marked with a database type of c-tree or Btrieve instead of SQL or Default.
- Making sure that the SQL key options are valid for each key of every table.

In Dexterity Utilities, after your dictionary is opened from the File menu as an Editable Dictionary, the SQL Compatibility window can be accessed from the Utilities button. From this window you can click on the Check button to print a report that will list any potential problems that are identified. This includes any SQL limits that have been exceeded in your dictionary along with incompatibilities with any of the other verifications that can be preformed. You can choose which specific verifications you would like to include by selecting the appropriate Categories.

You should use the SQL Compatibility utility with your ***complete*** *development dictionary. If it is used with the extracted dictionary, the SQL limits may not be evaluated properly if Dynamics fields are included in your third party tables.*

One of the most obvious challenges developers face when moving applications from an ISAM database to a SQL database is the maximum record size. The maximum record size for a SQL table is quite a bit smaller (1,962 bytes) then the maximum record size in an ISAM table (Btrieve 4,088 bytes, c-tree 32,767 bytes). If you do have to reduce the record size of your tables more information on table normalization can be found in most database development books.

An example of the SQL Compatibility window is shown in the following illustration:



From the SQL Compatibility window you can also choose to automatically correct any incompatibilities by choosing the button called Make Compatible. This button will make the following changes according to the verifications that you have specified:

- Any table field that does not have a physical name will be given one. The physical name will be created in all uppercase using the first 30 characters of the field name.
- Any table and table field that has a physical name that uses a SQL reserved word will have an integer appended to the name. In addition, if necessary, the physical name will be truncated.
- Any tables that have database type marked as c-tree or Btrieve will be modified to have Default selected as the database type.
- The Create Index option will be marked for each key of every table in the dictionary. The Create Index -Unique SQL key option will also be marked based upon the current setting of the Duplicates key option. If Duplicates is not marked, the Unique key option will be marked. If Duplicates is marked, the Unique key option will be unmarked.

Any SQL Server table limits that have been reached must be adjusted manually using Dexterity. The Make Compatible option will not automatically modify the dictionary to address this problem.

You can use the Make Compatible feature to make adjustments to conflicting resources, but keep in mind you that you generally have more control and flexibility making those changes manually with Dexterity. In addition, since you are using a complete development dictionary when you utilize the SQL Compatibility utility, there may be Dynamics resources that are included on your SQL Compatibility report. Any references to Dynamics resources can be ignored. However in that situation, the Make Compatible button should definitely not be used because you would not want to make changes to any of the Dynamics' core resources.

The following is an example of the SQL Compatibility report that is printed when the Check button is selected:

```
                              SQL Compatibility Report
SQL Limits:

      The row size of Table IV_Item_MSTR is too large. Limit is 1962

Field Physical Names:

      Field Item Number does not have a physical name.
      Field Description does not have a physical name.
      Field Price does not have a physical name.
      Field Item Type does not have a physical name.

Reserved Words:

      The physical name, CREATE, for field Created Date is a reserved
      word.

Table Database Types:

      Table IV_Item_MSTR, with physical name IV00100, has database type
      ctree.

Create Key Index:

      Create index not turned on for key IV_Item_MSTR_Key2 of table
      IV_Item_MSTR
```

The following is an example of the SQL Compatibility report that is printed when the Make Compatible button is selected:

```
                         SQL Compatibility Report
SQL Limits:

      The row size of Table IV_Item_MSTR is too large. Limit is 1962

Field Physical Names:

      Physical name ITEM NUMBER added for field Item Number.
      Physical name DESCRIPTION added for field Description.
      Physical name PRICE added for field Price.
      Physical name ITEM TYPE added for field Item Type.

Reserved Words:

      Physical name CREATE for field Created Date renamed to CREATE1.

Table Database Types:

      Changed database type for table IV_Item_MSTR to Default.

Create Key Index:

      Create index turned on for key IV_Item_MSTR_Key2 of table
      IV_Item_MSTR
```

Accessing Existing SQL Data

If there is existing SQL data that you would like to access with your Dexterity application, you will need to be sure to define your dictionary's resources properly in order to successfully access those tables. In addition, there is also a change that will need to be made to the existing SQL table that you will be working with. The following gives you additional detail about these specific issues:

Data Types

Data types must be defined in Dexterity for each unique column in the existing SQL table you would like to access. The data type needs to use a control type that corresponds directly to the SQL data types that have been defined for the columns. For example, if a column in an existing table has been defined as a SQL integer, it must be created as a long integer in Dexterity. The following table lists the Dexterity control types along with their corresponding SQL data types:

Dexterity Control Type	SQL Server Data Type
Boolean	tinyint
Check Box	tinyint
Combo Box	char
Composite	Each component of the composite is stored in its own column in the SQL table. The name of each column is the physical name of the composite field followed by an underscore and the number of the component.
Currency	numeric(19,5)
Date	datetime
Drop-down List	smallint
Horizontal List Box	smallint
Integer	smallint
List Box	smallint
Long integer	int
Multi-select List Box	binary(4)
Non-native List Box	smallint
Picture	image
Radio Group	smallint
String	char
Text	text
Time	datetime
Visual Switch	smallint

When defining the data type in Dexterity, if there are multiple control types that are appropriate, choose the control type that corresponds most closely to how information will be displayed within your application.

Special note must be taken when assigning the keyable length to string and text data types. The keyable length that is used for SQL char fields is always the same as its storage size. However in Dexterity, the storage sizes for string and text fields do not directly correspond with their keyable length. That is because length bytes are added to these types of the fields within Dexterity. To represent the length of each field, one byte is added to string fields and two bytes are added to text fields. In addition, by default, these fields may also include another byte to ensure an even-number storage size. When defining the keyable length for existing data, you will want to make sure that you assign the keyable length in Dexterity to be at least equivalent to the storage size of the column in SQL. If not, the existing data may be truncated when displayed in your application.

When setting date fields in Dexterity, the time portion of the SQL datetime field is automatically set to 12:00:00 AM. Similarly when setting time fields, the date portion of the SQL datetime field is automatically set to 1/1/1900. Therefore, when creating SQL data to import into a Dexterity application, make sure to set the time portion of a date field and the date portion of a time field to 12:00:00 AM and 1/1/1900 respectively. This will ensure that the data will be handled properly by Dexterity.

Fields

Fields must be defined in Dexterity for each unique column in the existing SQL table. The field's physical name should be equivalent to the column name in SQL. However, since Dexterity does not allow for underscores in a field's physical name, if an existing SQL column contains an underscore you will need to replace it with a space when you enter the physical name in Dexterity. It will then automatically be converted an underscore when the field is accessed by your application.

Tables

You will need to create a table definition in Dexterity for each SQL table you would like to access. Use the existing SQL table name as the physical name of the table when you define the table within Dexterity. In addition, make sure to include a field in the Dexterity table definition for every column in the SQL table.

Changes to SQL Table

You will also need to make sure that the physical SQL table that you want to access contains a DEX_ROW_ID column. This is necessary in order for active locking to perform correctly. If this column does not already exist in the SQL table you want to access, it needs to be created in SQL using the following characteristics:

Column Name	Type	Length	Nulls	Identity Column
DEX_ROW_ID	Integer	4	0	YES

Appendix A - Granting Table Permissions with Pass-through SQL

Following is an example of how pass-through SQL can be used to create and execute a stored procedure that will grant permissions to a specified table. The stored procedure that is created is similar to a sample procedure shown earlier called amAutoGrant. However, in this example, not only is the stored procedure executed, it is also created.

```
{-----------------------------------------------------------------------
Use pass through SQL to grant permission to one of your third party
tables, shown in the example as table_physical_name.  This script should
be implemented as a window or form level event, such as on a change
script on a push button. In addition, the window needs to have a Text
field named 'SQL_Statements' that contains the pass-through SQL code.
-----------------------------------------------------------------------}

local long SQL_Connection, exe_status, l_error, clear_status.
local long GPS_error_number, SQL_error_number.
local string SQL_error_string, ODBC_error_string, table_physical_name.

table_physical_name = "IV_Item_Additional_Info_MSTR".

{--Connect to the SQL data source.--}
exe_status = SQL_Connect(SQL_Connection).
if exe_status = 0 then

      {--Build SQL Statements.  This could also have been done using
       a single pass through statement utilizing the SQL execute command
       but has been broken apart for a more adaptable sample.--}

      {--Drop the stored procedure amAutoGrant and table autotemp if they
       already exist. The char(13) = CR, and is only needed to neatly
       display the SQL statements in the text field. The semicolon ';' is
       the same as the 'go' statement used in previous stored procedure
       examples.--}

      'SQL_Statements' = "if exists (select * from sysobjects where id =
            object_id('amAutoGrant')) " + char(13) + "drop procedure " +
            "amAutoGrant " + char(13)+ "; " + char(13) + "if exists
            (select * from sysobjects where id = object_id('autotemp'))".
      'SQL_Statements' = SQL_Statements + "drop table autotemp " +
            char(13)+ "; ".

      exe_status = SQL_Execute(SQL_Connection, field 'SQL_Statements') .
      clear_status = SQL_Clear(SQL_Connection).
      if clear_status <> 0 then
            error "An error occurred clearing the pass-through SQL
            connection (A).".
            exe_status=clear_status.
      end if.
```

```
{-- Create table autotemp.--}
if exe_status =0 then
      'SQL_Statements' = "create table autotemp(name char(150)) " +
            char(13)+ "; "+char(13).
      exe_status = SQL_Execute(SQL_Connection, field
            'SQL_Statements') .
      clear_status = SQL_Clear(SQL_Connection).

      if clear_status <> 0 then
            error "An error occurred clearing the pass-through SQL
                  connection (B).".
            exe_status=clear_status.
      end if.
end if.

{-- Create amAutoGrant stored procedure.--}
if exe_status= 0 then

      {--Build SQL Statements--}
      SQL_Statements= "create procedure amAutoGrant @tablename
            char(150) output as set nocount on ".
      SQL_Statements = SQL_Statements + "DECLARE @command
            varchar(255) SELECT @command = ".
      SQL_Statements = SQL_Statements + "'grant
            SELECT,INSERT,DELETE,UPDATE on '+rtrim(@tablename)+'
            to DYNGRP' ".
      SQL_Statements = SQL_Statements + "EXEC (@command) DELETE
            FROM autotemp SELECT @command = ".
      SQL_Statements = SQL_Statements + QUOTE + "insert into
            autotemp select name from sysobjects where name like
            'zDP_" + QUOTE.
      SQL_Statements = SQL_Statements + "+rtrim(@tablename) + " +
            QUOTE + "%'" + QUOTE + "EXEC (@command) DECLARE
            TheCursor CURSOR FOR ".
      SQL_Statements = SQL_Statements + "select 'grant EXECUTE on
            '+ rtrim(name) + " + QUOTE + " to DYNGRP" + QUOTE + "
            from autotemp ".
      SQL_Statements = SQL_Statements + "OPEN TheCursor WHILE
            @@FETCH_STATUS = @@FETCH_STATUS BEGIN FETCH NEXT FROM
            TheCursor INTO @command ".
      SQL_Statements = SQL_Statements + "IF @@FETCH_STATUS = -2
            CONTINUE IF @@FETCH_STATUS = -1 BREAK EXEC (@command)
            END ".
      SQL_Statements = SQL_Statements + "CLOSE TheCursor DEALLOCATE
            TheCursor ; SET NOCOUNT off ; ".
```

```
		{--Execute SQL Statements--}
		exe_status = SQL_Execute(SQL_Connection, field
			'SQL_Statements').
		if exe_status =0 then
			clear_status = SQL_Clear(SQL_Connection).
			if clear_status <> 0 then
				error "An error occurred clearing the
					pass-through SQL connection (C).".
				exe_status=clear_status.
			end if.
		end if.

	end if.

	{-- set permissions on the amAutoGrant procedure.--}
	if exe_status =0 then
		'SQL_Statements' = "grant execute on TWO.dbo.amAutoGrant
			to DYNGRP".
		exe_status = SQL_Execute(SQL_Connection, field
			'SQL_Statements') .
		if exe_status = 0 then
			clear_status = SQL_Clear(SQL_Connection).
			if clear_status <> 0 then
				error "An error occurred clearing the
					pass-through SQL connection (D).".
				exe_status=clear_status.
			end if.
		end if.
	end if.

	if exe_status = 0 then
		'SQL_Statements' = "SQL stored procedure successfully
			created.".

		 {--Run amAutoGrant stored procedure. At this point the
		    amAutoGrant stored procedure can be run for any number
		    of tables, simply set the table_physical_name and call
		    amAutoGrant again.--}
		call amAutoGrant, l_error, table_physical_name.
		SQL_Statements= SQL_Statements + char(13) + "DYNGRP
			permissions granted to " + table_physical_name + ".".
		SQL_Statements= SQL_Statements + char(13) + "Done.".

	else {--Handle errors...--}
		warning "An error occurred creating the pass-through SQL
			 connection: " + str(exe_status).
		SQL_Statements= SQL_Statements + char(13) + "An error
			occurred creating the pass-through SQL connection: " +
			str(exe_status).
		exe_status = SQL_GetError(SQL_Connection, GPS_error_number,
			SQL_error_number, SQL_error_string, ODBC_error_string).
```

```
            if exe_status = 0 then
                  SQL_Statements= SQL_Statements + char(13) + "GPS Error:
                        " + str(GPS_error_number).
                  SQL_Statements= SQL_Statements + char(13) + "SQL Error:
                        " + str(SQL_error_number) + " " +
                        SQL_error_string.
                  SQL_Statements= SQL_Statements + char(13) + "ODBC
                        Error: " + ODBC_error_string.
                  SQL_Statements= SQL_Statements + char(13) + "Done.".
            else
                  error "Unable to retrieve SQL error information.".
            end if.
      end if.
end if.
```

Additional notes on trouble-shooting Pass-through SQL issues:

- Use the DEX.INI entries SQLLogSQLStmt=TRUE, and SQLLogODBCMessages=TRUE to create a DEXSQL.LOG file.
- The Trace settings option in the ODBC Data Source Administrator or the utility ODBC Test from the Microsoft ODBC SDK can be helpful. The Microsoft ODBC SDK is available from Microsoft Corporation.
- Dexterity uses the series pathname to determine the database that the stored procedure will be created in. Therefore, your prototype procedure that is used to run the stored procedure (amAutoGrant) should be in the same series as the above script.

Appendix B - Other SQL Topics for Stand-Alone Dexterity Applications

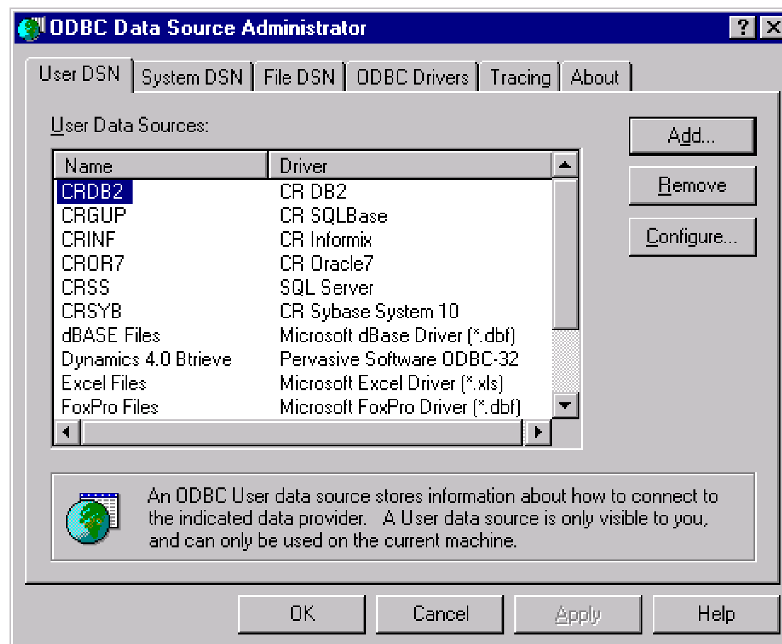
ODBC Data Sources

In order to use Dexterity to access SQL data, Microsoft SQL Server 6.5 must be installed and properly configured. In addition, because Dexterity communicates with SQL Server through the use of Open Database Connectivity (ODBC), a SQL ODBC data source must also be setup in order for Dexterity to be used to connect to SQL data.

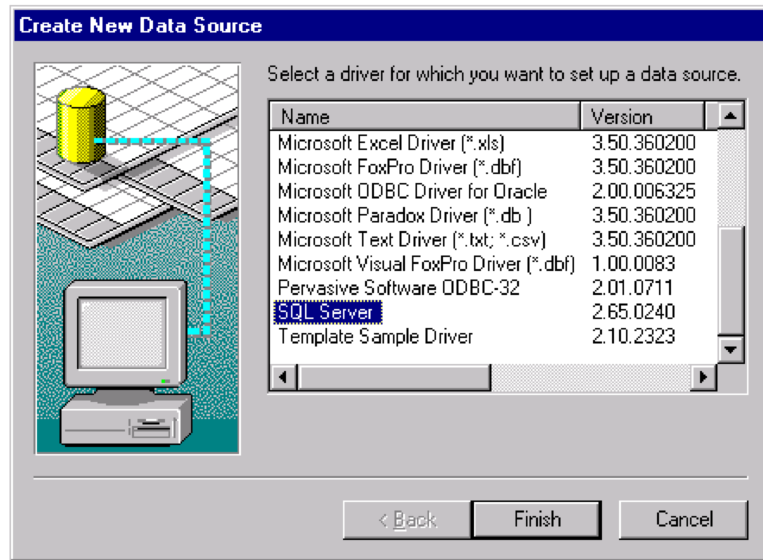
When working with an application that integrates with Dynamics C/S+ for SQL, Microsoft SQL Server and a SQL ODBC data source are installed and configured as part of the steps to setup the Dynamics application. However, if you are building your own stand-alone application, you will need to perform the following steps:

- **Install Microsoft SQL Server**
  For the current release of Dexterity, you will need to install Microsoft SQL Server 6.5 along with Service Pack 1 for SQL Server 6.5. In addition, during the installation process be sure to select either *Binary Order* or *Dictionary Order, Case-insensitive*. This will ensure that sorting will be performed properly within your application.

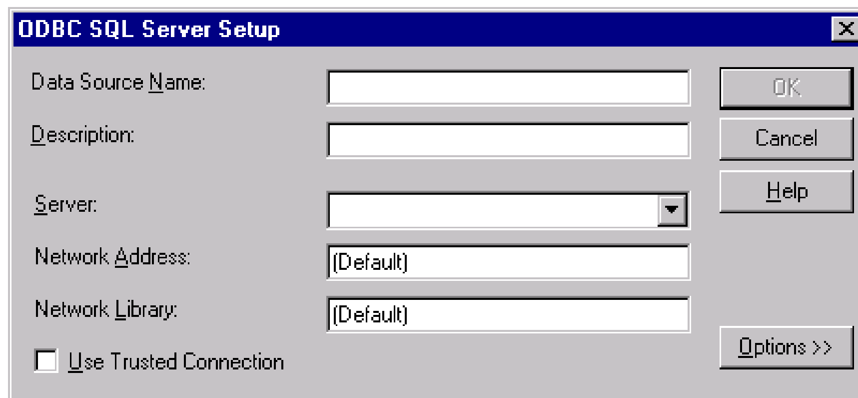
- **Configure a SQL ODBC Data Source**
  Launch the ODBC Administrator by double-clicking on the ODBC icon in the Windows Control Panel. Select the User DSN tab and then choose Add.



Select SQL Server from the list and choose Finish.



The ODBC SQL Server Setup window will appear. Enter a unique name for the data source and a brief description. Choose the Server you will be using and then click on the Options Button.



Unmark the option to Generate Stored Procedure for Prepared Statement and the option to Convert OEM to ANSI characters. This is necessary in order for your application to work correctly. Then choose OK.

ODBC SQL Server Setup

Data Source Name: Dexterity Sample Data
Description: Sample Data for Dexterity App
Server: GPSALPHA2
Network Address: (Default)
Network Library: (Default)
Use Trusted Connection

OK
Cancel
Help
Profiling...
Options >>

Login
Database Name:
Language Name: (Default)
Generate Stored Procedure for Prepared Statement
Use ANSI Quoted Identifiers
Use ANSI Nulls, Padding and Warnings

Translation
Select...
Convert OEM to ANSI characters

If Generate Stored Procedure for Prepared Statement is marked and your application includes fields that are not required, you will not be able to save records in which those non-required fields are empty. If Convert OEM to ANSI characters is marked, the functionality to encrypt fields within your application will not work properly.

SQL Logins

Before any users can access SQL data with your application, they must log into a SQL data source. When working with an application that integrates with Dynamics, this operation is handled by the accounting application. However, when you are building your own stand-alone application that does not integrate with Dynamics, you will need to deal with SQL Logins within your own program.

Although a Login window will automatically display when a user first attempts to access data, it is generally recommended that you control the timing by requiring users to log in as soon as your application is started. This can be done by opening the Dexterity Login window using a function called Login_OpenDexDialog().

You can also incorporate the login into one of your own windows by using other functions from the Login function library such as LoginIntoDataSource(). This would then allow you to combine the process of logging into your application with the procedure to log into a data source.

Record Locking

Dexterity takes advantage of two specific tables to implement record locking within an application. It uses a DEX_SESSION table to determine which client machines are logged into a specific SQL database from your Dexterity application and a DEX_LOCK table to determine if any of the users have actively locked a record within one of your SQL tables. These two tables will be used automatically as records are accessed from within your application, but they will not always be created automatically.

Creating the DEX_SESSION and the DEX_LOCK Tables

Both the DEX_SESSION and the DEX_LOCK tables exist within the standard SQL Server database called tempdb.. Because they are temporary tables, a Dexterity application will need to determine if the tables exist and create them if they do not exist. If you are integrating with Dynamics C/S+ for SQL, these tables will be created automatically by the accounting application. However, if you are building your own stand-alone application, you must be sure that your application handles the creation of these tables. Typically, this can be done by including a stored procedure within your application's SQL Server startup. It should be similar to the following ISQL example code.

If your application does not implement active locking, you can exclude the portion of the following code that pertains to the DEX_LOCK table. However, you will still need to include the stored procedure for the creation of the DEX_SESSION table since that table is used to allow both active and passive locking to function correctly.

```
use master
go

create procedure Build_Locks
as
exec ('create table tempdb..DEX_LOCK (session_id int, row_id int,
table_path_Name char(100))')
exec ('create unique index PK_DEX_LOCK on tempdb..DEX_LOCK(row_id,
table_path_name)')
exec ('create table tempdb..DEX_SESSION (session_id int identity,
sqlsvr_spid_id smallint)')
exec ('create unique index PK_DEX_SESSION on tempdb..DEX_SESSION(session_id)')
exec ('use tempdb grant insert, update, select, delete on DEX_LOCK to public')
exec (' use tempdb grant insert, update, select, delete on DEX_SESSION to
public')
return
go

sp_makestartup Build_Locks
go
```

Clearing Stranded Locks

When a session is concluded normally, such as when your Dexterity application is closed properly, Dexterity will always check the DEX_LOCK table and remove all records containing that session ID. Therefore, removing any locks associated with the session. However, when a session is concluded abnormally, such as when the computer's power supply is interrupted, Dexterity does not automatically clear all locks associated with that session ID. In that situation, locks will remain stranded. If you are integrating with Dynamics C/S+ for SQL these stranded locks will be cleared automatically when the application is restarted. However, if you are building a stand-alone application, your program must include a means to handle stranded locks. This could be done by including the following code in a stored procedure that is called from your Dexterity login script.

```
create procedure Release_Strandeds
as
declare @DEX_LOCK_ID char(15)
declare @iSession_ID int
declare T_cursor CURSOR for select session_id from tempdb..DEX_LOCK
set nocount on
OPEN T_cursor
FETCH NEXT FROM T_cursor INTO @iSession_ID
WHILE (@@FETCH_STATUS <> -1)
begin
   select @DEX_LOCK_ID = stuff( '##DL0000000000', 14 -
      datalength(rtrim(ltrim((convert(char(10),@iSession_ID)))))+1,
      datalength(rtrim(ltrim((convert(char(10),@iSession_ID))))),
      convert(char(10), @iSession_ID))

      if not exists ( select name
         from
                tempdb..sysobjects
         where
      tempdb..sysobjects.name = @DEX_LOCK_ID)
```

```
        begin
           /*This is a stranded lock, so clear it.*/
           delete from tempdb..DEX_LOCK where
            tempdb..DEX_LOCK.session_id = @iSession_ID
        end
        FETCH NEXT FROM T_cursor INTO @iSession_ID
end
DEALLOCATE T_cursor
return
go
```

SQL Pathnames

SQLPath Procedure

Each time a SQL table is opened by a Dexterity application, the series that has been assigned to the table is examined. If the series is not Pathname or Design Document, a procedure called SQLPath is automatically called. This procedure determines where each of the SQL tables is located. The SQLPath procedure is passed parameters about the table that is being accessed and it will find the appropriate path to use to find the table.

If no SQLPath procedure exists in the application, the system will look for the table in the default database for the current user.

If you are integrating with Dynamics C/S+ for SQL, this procedure already exists and will be used by your application as well as Dynamics. However, if you are building a stand-alone application, even though it will be called automatically by Dexterity, you will still need to create the procedure. The script must be called SQLPath, and it must be created with the following parameters:

- *dict_id* – An in parameter that is an integer specifying the ID for the current dictionary.
- *table_series* – An in parameter that is an integer specifying the table series for the current table.
- *table_group* – An in parameter that is an integer corresponding to the table group that the table is included in.
- *data_path* – An out parameter that is a string specifying the location of the data table. The path is in the following format with the server name and owner name being optional: `:server_name:database_name/owner_name/`

SQL Pathnames Table

The procedure that you create will generally read the locations of the SQL table from a SQL Pathnames file. This is a table that you must also create for any stand-alone applications you write. It will store information such as the table series or group and the appropriate path to the table. This table **must** be assigned to the Pathname series or the SQLPath procedure will not run properly. Generally it will be created as a c-tree table so it can be stored locally on each machine next to the application's dictionary. It typically contains an integer field that stores the Table Group ID, an integer field that stores the Table Series, and a string field that stores the path to the data. The SQL Path table may also include a Company ID field if multiple companies may be accessed from your application.

The Table Group ID for all of the tables in your application can be obtained by using the Resource_StartSeriesFill() and Resource_GetInfo() functions.

The Table Series values include the following:

1 – Financial	*7 – System*	*12 – Pathname*
2 – Sales	*8 – Company*	*13 – Design Document*
3 – Purchasing	*9 – Online Documentation*	*14 – Dexterity*
4 – Inventory	*10 – 3rd Party*	*15 – Dexterity System*
5 – Payroll	*11 – Payroll Tax*	*16 – Report Writer*
6 – Project		

SQLScriptPath Procedure

Each time a stored procedure is called by a Dexterity application, a procedure called SQLScriptPath is executed automatically to determine the exact path to the stored procedure. This procedure is similar to the SQLPath procedure that is used to determine the location of SQL tables. If you are integrating with Dynamics C/S+ for SQL, this SQLScriptPath procedure already exists and will be used by your application as well as Dynamics. However, if you are building a stand-alone application, even though it will be called automatically by Dexterity, you will still need to create the procedure. The script must be called SQLScriptPath, and it must be created with the following parameters:

- *product_id* – An in parameter that is an integer specifying the ID for the current dictionary.

- *core_id* – An in parameter that is an integer specifying the ID for the core associated with the stored procedure's prototype procedure.

The Core values include the following:

1 – Financial	*3– Purchasing*	*5 - Payroll*	*7 – System*
2 – Sales	*4 – Inventory*	*6 - Project*	

- *procedure_id* – An in parameter that is an integer corresponding to the resource ID of the stored procedure's prototype procedure.

- *data_path* – An out parameter that is a string specifying the location of the stored procedure. The path is in the following format with the server name and owner name being optional: `:server_name:database_name/owner_name/`