#### **Programming OLE DB SQL Server Applications**

OLE DB is a low-level, COM API that is used for accessing data. OLE DB is recommended for developing tools, utilities, or low-level components that need high performance. The OLE DB Provider for SQL Server (SQLOLEDB) is a native, high performance provider that accesses the SQL Server TDS protocol directly.

SQLOLEDB exposes interfaces to consumers wanting access to data on one or more computers running an instance of Microsoft® SQL Server<sup>™</sup> 2000 or SQL Server version 7.0 or earlier.

When developing an OLE DB consumer, select a provider developed for the data source to consume. Use SQLOLEDB to develop an optimized OLE DB consumer for SQL Server databases.

SQLOLEDB is an OLE DB version 2.0–compliant provider.

SQLOLEDB passes the command statements (such as SQL-92 and Transact-SQL) through to the server. The server rejects invalid commands.

# Getting Started with the OLE DB Provider for SQL Server

The topics in this section describe how to use Microsoft OLE DB Provider for SQL Server to communicate with Microsoft® SQL Server<sup>TM</sup> 2000.

# **OLE DB Syntax Conventions**

Convention	Used for
UPPERCASE	Transact-SQL functions and statements, and C macro
	names.
Monospace	Sample commands and program code.
Italic	Function parameter names and information that the user or the application must provide.
Bold	Function names, parameter keywords, and other syntax that must be typed exactly as shown.

# System Requirements for the OLE DB Provider for SQL Server

To access data in Microsoft® SQL Server™ 2000, you must have the following software installed:

- Microsoft OLE DB Provider for SQL Server (SQLOLEDB).
- An instance of SQL Server.
- Network software.

OLE DB consumer development requires Microsoft Visual C++® version 5.0.

#### **SQLOLEDB Requirements**

SQLOLEDB requires one of the following:

- Microsoft Windows® 95 or Windows 98 operating system on Intel® computers.
- Microsoft Windows 2000 or Microsoft Windows NT® 4.0 operating system on Intel computers.

#### **SQL Server Requirements**

To use SQLOLEDB to access data in SQL Server databases, you must have an instance of SQL Server 2000 or SQL Server version 6.5 or later installed; the catalog stored procedures must also be installed.

#### **Network Software Requirements**

SQLOLEDB communicates with network software through the SQL Server Net-Library interface, which requires a Net-Library dynamic-link library (DLL). The Microsoft OLE DB Provider for SQL Server 2000 requires SQL Server 2000 Net-Library .dll files, which are installed when you run the client portion of SQL Server 2000 Setup.

#### **See Also**

**Configuring Client Network Connections** 

Hardware and Software Requirements for Installing SQL Server

**Client Net-Libraries and Network Protocols** 

<u>Upgrading the Catalog Stored Procedures (OLE DB)</u>

#### **Installing the OLE DB Provider for SQL Server**

The Microsoft OLE DB Provider for SQL Server (SQLOLEDB) is installed automatically when you install Microsoft® SQL Server<sup>TM</sup> 2000. In a custom installation, the provider is installed when any of the following components are selected:

- Server Components
- Management Tools
- Client Connectivity

#### **SQLOLEDB** Files

Files necessary to develop SQLOLEDB consumers are installed when the appropriate option is selected during a custom installation; several SQLOLEDB sample applications are included. The samples implement SQLOLEDB consumers in C++.

Directory	File	Description
Program files\Common	Sqloledb.dll	Dynamic-link library that
files\System\Ole db		implements the
		SQLOLEDB provider.
Program Files\Microsoft SQL	Oledb.h	OLE DB SDK header file
Server\80\Tools\DevTools\Include		for OLE DB providers and
		consumers.
Program Files\Microsoft SQL	Sqloledb.h	Header file used for
Server\80\Tools\DevTools\Include		developing SQLOLEDB
		consumers.
Program Files\Microsoft SQL	Oledb.lib	Library file used for
Server\80\Tools\Dev Tools\Lib		developing SQLOLEDB
		consumers.

#### **OLE DB SDK**

The primary source of information for OLE DB is the OLE DB Software Development Kit (SDK), which can be downloaded from Microsoft Web site.

The OLE DB SDK is not installed with SQL Server 2000. To develop OLE DB applications, you need the OLE DB SDK from Microsoft Web site.

#### **See Also**

Overview of Installing SQL Server 2000

#### **Upgrading the Catalog Stored Procedures (OLE DB)**

The Microsoft OLE DB Provider for SQL Server (SQLOLEDB) uses a set of system stored procedures known as catalog stored procedures to obtain information from the system catalog. Microsoft® SQL Server™ 2000 installs the catalog stored procedures automatically when you install or upgrade an instance of SQL Server. The Instcat.sql file included with this provider includes updates to the catalog stored procedures. If this version of SQLOLEDB will be used with SQL Server version 6.5, the SQL Server system administrator must upgrade the catalog stored procedures on the earlier instance of SQL Server by running Instcat.sql. Upgrading the catalog stored procedures does not affect the operations of SQL Server clients.

To upgrade the catalog stored procedures

## **Creating an OLE DB Application**

Creating an OLE DB application involves these steps:

- 1. Establishing a connection to a data source.
- 2. Executing a command.
- 3. Processing the results.

#### **Establishing a Connection to a Data Source**

To access the Microsoft OLE DB Provider for SQL Server (SQLOLEDB), the consumer must first create an instance of a data source object by calling the **CoCreateInstance** method. A unique class identifier (CLSID) identifies each OLE DB provider. For SQLOLEDB, the class identifier is CLSID\_SQLOLEDB.

The data source object exposes the **IDBProperties** interface, which the consumer uses to provide basic authentication information such as server name, database name, user ID, and password. The **IDBProperties::SetProperties** method is called to set these properties.

If there are multiple instances of Microsoft® SQL Server™ running on the computer, the server name is specified as ServerName\\InstanceName (the escape sequence \\ is used for the backslash).

The data source object also exposes the **IDBInitialize** interface. After the properties are set, connection to the data source is established by calling the **IDBInitialize**::**Initialize** method. For example:

```
CoCreateInstance(CLSID_SQLOLEDB,
NULL,
CLSCTX_INPROC_SERVER,
IID_IDBInitialize,
(void **) &pIDBInitialize)
```

This call to **CoCreateInstance** creates a single object of the class associated with CLSID\_SQLOLEDB (CSLID associated with the data and code that will be used to create the object). IID\_IDBInitialize is a reference to the identifier of the interface (**IDBInitialize**) to be used to communicate with the object.

This is a sample function that initializes and establishes a connection to the data source:

```
void InitializeAndEstablishConnection()
{
  //Initialize the COM library.
```

```
CoInitialize(NULL);
//Obtain access to the SQLOLEDB provider.
hr = CoCreateInstance(CLSID_SQLOLEDB,
            NULL,
            CLSCTX_INPROC_SERVER,
            IID IDBInitialize,
            (void **) &pIDBInitialize);
/*
Initialize the property values needed
to establish the connection.
*/
for(i = 0; i < 4; i++)
  VariantInit(&InitProperties[i].vValue);
//Server name.
InitProperties[0].dwPropertyID = DBPROP_INIT_DATASOURCE
InitProperties[0].vValue.vt
                           = VT BSTR;
InitProperties[0].vValue.bstrVal=
             SysAllocString(L"Server");
InitProperties[0].dwOptions = DBPROPOPTIONS_REQUIRED;
                         = DB_NULLID;
InitProperties[0].colid
//Database.
InitProperties[1].dwPropertyID = DBPROP INIT CATALOG;
InitProperties[1].vValue.vt
                           = VT BSTR;
InitProperties[1].vValue.bstrVal= SysAllocString(L"database");
InitProperties[1].dwOptions = DBPROPOPTIONS_REQUIRED;
InitProperties[1].colid
                         = DB NULLID;
//Username (login).
InitProperties[2].dwPropertyID = DBPROP_AUTH_USERID;
InitProperties[2].vValue.vt
                           = VT BSTR;
InitProperties[2].vValue.bstrVal= SysAllocString(L"sa");
                            = DBPROPOPTIONS REQUIRED;
InitProperties[2].dwOptions
InitProperties[2].colid
                         = DB NULLID;
//Password.
```

```
InitProperties[3].dwPropertyID = DBPROP_AUTH_PASSWORD;
InitProperties[3].vValue.vt
                            = VT BSTR;
InitProperties[3].vValue.bstrVal= SysAllocString(L"");
                             = DBPROPOPTIONS_REQUIRED;
InitProperties[3].dwOptions
InitProperties[3].colid
                          = DB NULLID;
/*
Construct the DBPROPSET structure(rgInitPropSet). The
DBPROPSET structure is used to pass an array of DBPROP
structures (InitProperties) to the SetProperties method.
*/
rgInitPropSet[0].guidPropertySet = DBPROPSET_DBINIT;
rgInitPropSet[0].cProperties
                             = 4:
rgInitPropSet[0].rgProperties = InitProperties;
//Set initialization properties.
hr = pIDBInitialize->QueryInterface(IID_IDBProperties,
                  (void **)&pIDBProperties);
hr = pIDBProperties->SetProperties(1, rgInitPropSet);
pIDBProperties->Release();
//Now establish the connection to the data source.
pIDBInitialize->Initialize()
```

#### **Executing a Command**

After the connection to a data source is established, the consumer calls the **IDBCreateSession::CreateSession** method to create a session. The session acts as a command, rowset, or transaction factory.

To work directly with individual tables or indexes, the consumer requests the **IOpenRowset** interface. The **IOpenRowset**::**OpenRowset** method opens and returns a rowset that includes all rows from a single base table or index.

To execute a command (such as SELECT \* FROM Authors), the consumer requests the **IDBCreateCommand** interface. The consumer can execute the **IDBCreateCommand**::CreateCommand method to create a command object and request for the **ICommandText** interface. The

**ICommandText::SetCommandText** method is used to specify the command that is to be executed.

The **Execute** command is used to execute the command. The command can be any SQL statement, procedure name, and so on. Not all commands produce a result set (rowset) object. Commands such as SELECT \* FROM **authors** produce a result set.

#### **OLE DB Extensions for XML**

The **ICommandText**::**SetCommandText** and **ICommand**::**Execute** statements can be used to set XML documents as command text, execute the command, and retrieve the result as a stream, which can then be used in further processing, such as passing the XML to the Document Object Model (DOM).

Templates are valid XML documents that contain one or more SQL command tags. These XML templates can be passed to

**ICommandText::SetCommandText**. When XML templates are set as command text using **ICommandText::SetCommandText**, the consumer must pass **DBGUID\_MSSQLXML** as the globally unique identifier (GUID) of the command syntax. This GUID indicates that the command text is an XML template.

The consumer must call **ICommand::Execute** to execute XML templates. To

obtain XML documents as a result set, **riid** must be set to **IStream**.

#### **Processing Results**

If a rowset object is produced by either the execution of a command or the generation of a rowset object directly from the provider, the consumer needs to retrieve and access data in the rowset.

Rowsets are central objects that enable all OLE DB data providers to expose data in tabular form. Conceptually, a rowset is a set of rows in which each row has column data. A rowset object exposes interfaces such as **IRowset** (contains methods for fetching rows from the rowset sequentially), **IAccessor** (permits the definition of a group of column bindings describing the way tabular data is bound to consumer program variables), **IColumnInfo** (provides information about columns in the rowset), and **IRowsetInfo** (provides information about rowset).

A consumer can call the **IRowset::GetData** method to retrieve a row of data from the rowset into a buffer. Before **GetData** is called, the consumer describes the buffer using a set of DBBINDING structures. Each binding describes how a column in a rowset is stored in a consumer buffer and contains information such as:

- Ordinal of the column (or parameter) to which the binding applies.
- What is bound (data value, length of the data, and its binding status).
- What is offset in the buffer to each of these parts.
- Length and type of the data values as they exist in the consumer buffer.

When getting the data, the provider uses information in each binding to determine where and how to retrieve data from the consumer buffer. When setting data in the consumer buffer, the provider uses information in each binding to determine where and how to return data in the consumer's buffer.

After the DBBINDING structures are specified, an accessor is created (**IAccessor::CreateAccessor**). An accessor is a collection of bindings and is

used to get or set the data in the consumer buffer.

#### **Compiling OLE DB Applications**

OLE DB applications must include Oledb.h, Sqloledb.h, and Oledberr.h (if using error constants defined in this file) files. Most applications use wide character strings to make OLE DB function calls. If applications are using TCHAR variables, the application must include #define UNICODE in the application. It converts the TCHAR variables to wide character strings. OLE DB applications must be linked with the Oledb.lib file. In a custom installation of Microsoft® SQL Server™ 2000, the header files are installed in the C:\Program Files\Microsoft SQL Server\80\Tools\Dev Tools\Include directory and the library files are installed in the C:\Program Files\Microsoft SQL Server\80\Tools\Dev Tools\Lib directory. The SQL Server Include and Lib directories are located in the INCLUDE and LIB path on the compiler.

The latest versions of these files can be downloaded with the latest Microsoft Data Access SDK from Microsoft Web site. If you have downloaded a version of the Microsoft Data Access SDK and the dates are later than the dates for SQL Server 2000, place the MSDA directories before the SQL Server 2000 directories. For example:

LIB=c:\msdasdk\oledb\lib;c:\Program Files\Microsoft SQL Server\80\' c:\msdev\mfc\lib

INCLUDE=c:\msdasdk\oledb\include;c:\Program Files\Microsoft SQL
c:\msdev\include;c:\msdev\mfc\include

#### **About OLE DB Properties**

Consumers set property values to request specific object behavior. For example, consumers use properties to specify the interfaces to be exposed by a rowset. Consumers get the property values to determine the capabilities of an object such as rowset, session, or a data source object.

Each property has a value, type, description, and read/write attribute, and for rowset properties, an indicator of whether it can be applied on a column-by-column basis.

A property is identified by a GUID and an integer representing the property ID. A property set is a set of all properties that share the same GUID. In addition to the predefined OLE DB property sets, SQLOLEDB implements provider-specific property sets and properties in them. Each property belongs to one or more property groups. A property group is the group of all properties that apply to a particular object. Some property groups include the initialization property group, data source property group, session property group, rowset property group, table property group, column property group, and so on. There are properties in each of these property groups.

Setting property values involves:

- 1. Determining the properties for which to set values.
- 2. Determining the property sets that contain the identified properties.
- 3. Allocating an array of DBPROPSET structures, one for each identified property set.
- 4. Allocating an array of DBPROP structures for each property set. The number of elements in each array is the number of properties (identified in Step 1) that belong to that property set.
- 5. Filling in the DBPROP structure for each property.

- 6. Filling in information (property set GUID, count of number of elements, and a pointer to the corresponding DBPROP array) in the DBPROPSET structure for each property set.
- 7. Calling a method to set properties and passing the count and the array of DBPROPSET structures.

#### **Data Source Objects**

OLE DB uses the term data source for the set of OLE DB interfaces used to establish a link to a data store, such as Microsoft® SQL Server<sup>TM</sup> 2000. Creating an instance of the data source object of the provider is the first task of an OLE DB consumer.

Every OLE DB provider declares a class identifier (CLSID) for itself. The CLSID for SQLOLEDB is the C/C++ GUID CLSID\_SQLOLEDB. With the CLSID, the consumer uses the OLE **CoCreateInstance** function to manufacture an instance of the data source object.

SQLOLEDB is an in-process server. Instances of SQLOLEDB objects are created using the CLSCTX\_INPROC\_SERVER macro to indicate the executable context.

The SQLOLEDB data source object exposes the OLE DB initialization interfaces that allow the consumer to connect to existing SQL Server databases.

Every connection made through SQLOLEDB sets these options automatically:

- SET ANSI\_WARNINGS ON
- SET ANSI\_NULLS ON
- SET ANSI\_PADDING ON
- SET ANSI\_NULL\_DFLT\_ON ON
- SET QUOTED\_IDENTIFIER ON
- SET CONCAT\_OF\_NULL\_YIELDS\_NULL ON

This example uses the class identifier macro to create a SQLOLEDB data source object and get a reference to its **IDBInitialize** interface.

```
IDBInitialize* pIDBInitialize;
HRESULT hr;

hr = CoCreateInstance(CLSID_SQLOLEDB, NULL, CLSCTX_INPR IID_IDBInitialize, (void**) &pIDBInitialize);

if (SUCCEEDED(hr))
{
    // Perform necessary processing with the interface.
    pIDBInitialize->Uninitialize();
    pIDBInitialize->Release();
}
else
{
    // Display error from CoCreateInstance.
}
```

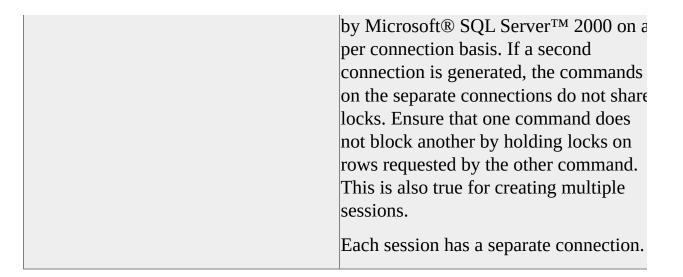
With successful creation of an instance of a SQLOLEDB data source object, the consumer application can continue by initializing the data source and creating sessions. OLE DB sessions present the interfaces that allow data access and manipulation.

SQLOLEDB makes its first connection to a specified instance of SQL Server 2000 as part of a successful data source initialization. The connection is maintained as long as a reference is maintained on any data source initialization interface, or until the **IDBInitialize::Uninitialize** method is called.

## **Data Source Properties**

SQLOLEDB implements data source properties as follows.

Property ID	Description	
DBPROP_CURRENTCATALOG	R/W: Read/write	
	Default: None	
	Description: The value of	
	DBPROP_CURRENTCATALOG	
	reports the current database for a	
	SQLOLEDB session. Setting the	
	property value has the identical effect as	
	setting the current database by using the	
	Transact-SQL USE <i>database</i> statement.	
DBPROP_MULTIPLECONNECTIONS		
	Default: VARIANT_TRUE	
	Description: If the connection is running	
	a command that does not produce a	
	rowset, or produces a rowset that is not	
	a server cursor and you execute another	
	command, a new connection will be	
	created to execute the new command if	
	DBPROP_MULTIPLECONNECTIONS is VARIANT_TRUE.	
	IS VARIANT_TRUE.	
	SQLOLEDB will not create another	
	connection if	
	DBPROP_MULTIPLECONNECTION	
	is VARIANT_FALSE or if a transaction	
	is active on the connection.	
	SQLOLEDB returns	
	DB_E_OBJECTOPEN if	
	DBPROP_MULTIPLECONNECTIONS	
	is VARIANT_FALSE and returns  E. EALL if there is an active transaction	
	E_FAIL if there is an active transaction.	
	Transactions and locking are managed	



In the provider-specific property set DBPROPSET\_SQLSERVERDATASOURCE, SQLOLEDB defines the following additional data source property.

Read/write t: VARIANT_FALSE
_
ption: To bulk copy,
OP_ENABLEFASTLOAD property
o VARIANT_TRUE. With this
ty set on the data source, the newly
l session allows consumer access to
owsetFastLoad interface.
property is set to
ANT_FALSE, IRowsetFastLoad
ce is available through
Rowset::OpenRowset by requesting
RowsetFastLoad interface or by
SSPROP_IRowsetFastLoad to
ANT_TRUE.

## **Data Source Information Properties**

In the provider-specific property set DBPROPSET\_SQLSERVERDATASOURCEINFO, SQLOLEDB defines the following data source information properties.

Property ID	Description
SSPROP_CHARACTERSET	Type: VT_BSTR
	R/W: R
	Default: NULL
	Description: The character
	set in the server. Apply to
	only Microsoft® SQL
	Server <sup>TM</sup> version 7.0 and
	earlier.
SSPROP_CURRENTCOLLATION	Type: VT_BSTR
	R/W: R
	Default: NULL
	Description: The current
	database collation name.
	Apply to only SQL Server
	2000.
SSPROP_SORTORDER	Type: VT_BSTR
	R/W: R
	Default: NULL
	Description: The sort order in
	the server. Apply to only
	SQL Server 7.0 and earlier.
SSPROP_UNICODELCID	Type: VT_I4
	R/W: Read
	Description: Unicode locale
	ID.
	This is the locale used for
	Unicode data sorting. The
	value of this property is 0 for
	1 1 0

	Microsoft SQL Server version 6.5.
SSPROP_UNICODECOMPARISONSTYLE	Type: VT_I4 R/W: Read Description: Unicode comparison style. The sorting options used for Unicode data sorting. The value of this property is 0 for SQL Server 6.5.

In the provider-specific property set DBPROPSET\_SQLSERVERSTREAM, SQLOLEDB defines the following additional properties.

Property ID	Description
SSPROP_STREAM_BASEPATH	Type: VT_BSTR R/W: Read/Write
	Description: Is used for resolving relamapping schema or external schema template.
SSPROP_STREAM_CONTENTTYPE	Type: VT_BSTR R/W: Read Only Description: If XSL is applied to the
	type property on <xsl:output> in the 2 the value of this property.</xsl:output>
SSPROP_STREAM_FLAGS	Type: dword R/W: Read/Write Description: Following values can be property (multiple values can be ORe
	STREAM_FLAGS_DISALLOW_UNO URL reference to any files is allowed for example, in a template you can symapping schema files. When STREAM_FLAGS_DISALLOW_UI

	the property, no URL references to the in the templates. URL references to files can slow down and it is also a security risk because it you may not be sure about the file contribution.
	STREAM_FLAGS_DISALLOW_A No absolute path to files is allowed. I relative to the template in which the f Absolute paths such as references to e security risk. Therefore, STREAM_FLAGS_DISALLOW_AI set to disallow absolute paths.
	STREAM_FLAGS_DISALLOW_( No queries are allowed in the templat <sql:query> tag is not allowed in a tell security reasons you may not want to in a template.</sql:query>
SSPROP_STREAM_MAPPINGSCHEMA	Type: VT_BSTR R/W: Read/Write Description: Is used for specifying a s XPath queries. The path specified car absolute. If the path specified is relative, base p SSPROP_STREAM_BASEPATH is a relative path. If the base path is not specified, the re relative to the current directory.
SSPROP_STREAM_XMLROOT	Type: VT_BSTR R/W: Read/Write Description: The result of a query (SO not be a well-formed document. Whe specified, the query result is wrapped provided by this property to return a vidocument (if query is executed in the cause the browser to display parser er

	the result. To avoid the error, SQL IS keyword ROOT. This keyword maps SSPROP_STREAM_XMLROOT proinformation, see <u>URL Access</u> .)
SSPROP_STREAM_XSL	Type: VT_BSTR R/W: Read/Write Description: Is used for specifying an specified can be relative or absolute. If the path specified is relative, the base SSPROP_STREAM_BASEPATH is relative path. If the base path is not specified, the rerelative to the current directory.

# **Initialization and Authorization Properties**

SQLOLEDB interprets OLE DB initialization and authorization properties as follows.

Property ID	Description
DBPROP_AUTH_CACHE_AUTHINFO	SQLOLEDB does not c authentication informati
	SQLOLEDB returns DB_S_ERRORSOCCU attempt to set the proper property structure <i>dwSta</i> indicates DBPROPSTATUS_NO
DBPROP_AUTH_CACHE_AUTHINFO	SQLOLEDB uses stand SQL Server <sup>™</sup> 2000 sec mechanisms to ensure p privacy.
	SQLOLEDB returns DB_S_ERRORSOCCU attempt to set the proper property structure <i>dwSta</i> indicates DBPROPSTATUS_NO
DBPROP_AUTH_INTEGRATED	If DBPROP_AUTH_IN set to a NULL pointer, a 'SSPI' VT_BSTR value, uses Windows Authenti authorize user access to database specified by th DBPROP_INIT_DATA DBPROP_INIT_CATA properties.

	If it is set to VT_EMPT SQL Server 2000 securi SQL Server 2000 login are specified in the DBPROP_AUTH_USE DBPROP_AUTH_PASS properties.
DBPROP_AUTH_MASK_PASSWORD	SQLOLEDB uses stand 2000 security mechanist password privacy.
	SQLOLEDB returns DB_S_ERRORSOCCU attempt to set the proper property structure <i>dwSta</i> indicates DBPROPSTATUS_NO
DBPROP_AUTH_PASSWORD	Password assigned to a 2000 login. This proper SQL Server Authenticat for authorizing access to database.
DBPROP_AUTH_PERSIST_ENCRYPTED	SQLOLEDB does not e authentication informati persisted.
	SQLOLEDB returns DB_S_ERRORSOCCU attempt to set the proper property structure <i>dwSta</i> indicates DBPROPSTATUS_NO
DBPROP_AUTH_PERSIST_SENSITIVE_AUTHINFO	SQLOLEDB persists au values, including an ima password, if requested t encryption is provided.

DBPROP_AUTH_USERID	SQL Server login. This when SQL Server Authorizing SQL Server database.
DBPROP_INIT_ASYNCH	SQLOLEDB does not saynchronous initiation.
	SQLOLEDB returns DB_S_ERRORSOCCU attempt to set the proper property structure <i>dwSta</i> indicates DBPROPSTATUS_NO
DBPROP_INIT_CATALOG	Name of an existing SQ database to which to co
DBPROP_INIT_DATASOURCE	Network name of a servinstance of Microsoft® If there are multiple instance 2000 running on then to connect to a specified as \$\ServerName\Instance\text{NerverName}\$\square\$ is use itself.
DBPROP_INIT_HWND	Window handle from the application. A valid win required for the initialized box displayed when proinitialization properties
DBPROP_INIT_IMPERSONATION_LEVEL	SQLOLEDB does not so impersonation level adjust SQLOLEDB returns DB_S_ERRORSOCCU attempt to set the proper

	property structure <i>dwSta</i> indicates DBPROPSTATUS_NO
DBPROP_INIT_LCID	SQLOLEDB validates t and returns an error if the not supported or is not i client.
DBPROP_INIT_LOCATION	SQLOLEDB returns DB_S_ERRORSOCCU attempt to set the proper property structure <i>dwSta</i> indicates DBPROPSTATUS_NO
DBPROP_INIT_MODE	SQLOLEDB returns DB_S_ERRORSOCCU attempt to set the proper property structure <i>dwSta</i> indicates DBPROPSTATUS_NO
DBPROP_INIT_PROMPT	SQLOLEDB supports a modes for data source in SQLOLEDB uses DBPROMPT_NOPROI default setting for the process.
DBPROP_INIT_PROTECTION_LEVEL	SQLOLEDB does not s protection level on conr instances of SQL Server
	SQLOLEDB returns DB_S_ERRORSOCCU attempt to set the proper property structure <i>dwSta</i> indicates DBPROPSTATUS_NO
DBPROP_INIT_PROVIDERSTRING	See SQLOLEDB Provious in this topic.

DBPROP_INIT_TIMEOUT	SQLOLEDB returns an
	initialization if a connec
	instance of SQL Server
	established within the n
	seconds specified.

In the provider-specific property set DBPROPSET\_SQLSERVERDBINIT, SQLOLEDB defines these additional initialization properties.

Property ID	Description
SSPROP_AUTH_REPL_SERVER_NAME	Type: VT_BSTR R/W: W Default: NULL Description: Replication server na option.
SSPROP_INIT_APPNAME	Type: VT_BSTR R/W: Read/write Description: The client application
SSPROP_INIT_AUTOTRANSLATE	Type: VT_BOOL R/W: Read/write Default: VARIANT_TRUE Description: OEM/ANSI character
	VARIANT_TRUE: SQLOLEDB t character strings sent between the server by converting through Unic problems in matching extended ch between the code pages on the clie server:
	Client DBTYPE_STR data sent to SQL Server <b>char</b> , <b>varchar</b> , or <b>tex</b> parameter, or column is converted to Unicode using the client ANSI (ACP), and then converted from U character using the ACP of the ser
	SQL Server 2000 <b>char, varchar</b> , of to a client DBTYPE_STR variable

from character to Unicode using the and then converted from Unicode using the client ACP.

These conversions are performed of SQLOLEDB. This requires that the code page (ACP) used on the serve on the client.

These settings have no effect on the that occur for these transfers:

Unicode DBTYPE\_WSTR client ( char, varchar, or text on the serv

**char, varchar**, or **text** server data Unicode DBTYPE\_WSTR variabl

ANSI DBTYPE\_STR client data s **nchar**, **nvarchar**, or **ntext** on the

Unicode **char**, **varchar**, or **text** se an ANSI DBTYPE\_STR variable

VARIANT\_FALSE: SQLOLEDB perform character translations.

SQLOLEDB does not translate cli character DBTYPE\_STR data sen varchar, or text variables, parame on the server. No translation is per char, varchar, or text data sent fr DBTYPE\_STR variables on the cl

If the client and the instance of SC are using different ACPs, extended be misinterpreted.

SSPROP\_INIT\_CURRENTLANGUAGE

Type: VT\_BSTR R/W: Read/write

Description: A SQL Server langua Identifies the language used for sy

	selection and formatting. The lang installed on the computer running SQL Server or data source initializ
SSPROP_INIT_ENCRYPT	Type: VT_BOOL R/W: Read/Write Default: VARIANT_FALSE Description: To encrypt the data gonetwork, SSPROP_INIT_ENCRY set to VARIANT_TRUE.
	Error occurs if the Enable Protoco set to ON on the client, and the SSPROP_INIT_ENCRYPT is set VARIANT_FALSE.
	If Enable Protocol Encryption is so client side, and SSPROP_INIT_El to VARIANT_TRUE, encryption on that particular connection.
SSPROP_INIT_FILENAME	Type: VT_BSTR R/W: Read/write Description: Specifies the primary attachable database. This database becomes the default database for the trace of the database at the initialization property DBPROP_INIT_CATALOG. If the does not exist, then it looks for the name specified in SSPROP_INIT_ and attaches that database with the in DBPROP_INIT_CATALOG. If was previously attached, SQL Server eattach it. This option is valid onliconnected to SQL Server 2000.
SSPROP_INIT_NETWORKADDRESS	Type: VT_BSTR R/W: Read/write

	Description: The network address running an instance of SQL Server the DBPROP_INIT_DATASOUR
SSPROP_INIT_NETWORKLIBRARY	Type: VT_BSTR R/W: Read/write Description: The name of the Net- used to communicate with an insta Server 2000. The name should not path or the .dll file name extensior The default is provided by the SQI Network Utility.
SSPROP_INIT_PACKETSIZE	Type: VT_I4 R/W: Read/write Description: A network packet size packet size property value must be and 32,767. The default SQLOLE packet size is 4,096.
SSPROP_INIT_TAGCOLUMNCOLLATION	Type: BOOL R/W:W Default: FALSE Description: Is used during a datal when server-side cursors are used. tags the data with collation inform from the server instead of the code client. Currently, this property is u distributed query process because collation of destination data and co correctly.
SSPROP_INIT_USEPROCFORPREP	Type: VT_I4 R/W: Read/write Default: SSPROPVAL_USEPROCFORPR Description: SQL Server stored pr Defines the use of SQL Server ten procedures to support the <b>IComm</b> interface. This property is meaning

	connecting to SQL Server 6.5. The ignored for later versions.
	SSPROPVAL_USEPROCFORPR temporary stored procedure is not command is prepared.
	SSPROPVAL_USEPROCFORPR temporary stored procedure is crea command is prepared. The tempor procedures are dropped when the streleased.
	SSPROPVAL_USEPROCFORPR A temporary stored procedure is command is prepared. The procedure when the command is unprepared ICommandPrepare::Unprepare, command is specified for the community if ICommandText::SetCommunity when all application references to are released.
SSPROP_INIT_WSID	Type: VT_BSTR R/W: Read/write Description: A string identifying tl

In the provider-specific property set DBPROPSET\_SQLSERVERDATASOURCEINFO, SQLOLEDB defines the following additional properties.

Property ID	Description
SSPROP_COLUMNLEVELCOLLATION	Type: VT_BOOL
	R/W: Read
	Default: VARIANT_TRUE
	Description: Used to determine
	if column collation is
	supported.

VARIANT_TRUE: Column level collation is supported (in case of SQL Server 2000)
VARIANT_FALSE: Column level collation is not supported.

#### **SQLOLEDB Provider String**

SQLOLEDB recognizes an ODBC-like syntax in provider string property values. The provider string property is provided as the value of the OLE DB initialization property DBPROP\_INIT\_PROVIDERSTRING when a connection is established to the OLE DB data source. This property specifies OLE DB provider-specific connection data required to implement a connection to the OLE DB data source. Within the string, elements are delimited by using a semicolon. The final element in the string must be terminated with a semicolon. Each element consists of a keyword, an equal sign character, and the value passed on initialization. For example:

Server=London1;UID=nancyd;

With SQLOLEDB, the consumer never needs to use the provider string property. The consumer can set any initialization property reflected in the provider string by using either OLE DB or SQLOLEDB-specific initialization properties.

SQLOLEDB recognizes the following keywords in the provider string property.

Keyword	PropertyID	Description
Address	SSPROP_INIT_NETWORKADDRESS	Network address
		an instance of So
		Server in the
		organization.
APP	SSPROP_INIT_APPNAME	String identifyin
		the application.
AttachDBFileName	DBPROP_INIT_PROVIDERSTRING	Name of the
		primary file

		(include the full path name) of ar attachable datab To use AttachDBFileNayou must also specify the datab name with the provider string DATABASE keyword. If the database was previously attacl SQL Server does not reattach it (it uses the attached database as the default for the connection).
AutoTranslate	SSPROP_INIT_AUTOTRANSLATE	Configures OEM/ANSI character translation. Recognized valuare "yes" and "n
Database	DBPROP_INIT_CATALOG	Database name.
Encrypt	SSPROP_INIT_ENCRYPT	Specifies if data should be encryl before sending i over the network
Language	SSPROPT_INIT_CURRENTLANGUAGE	SQL Server language record name.
Network	SSPROP_INIT_NETWORKLIBRARY	Net-Library used establish a connection to an

		instance of SQL Server in the organization.
PWD	DBPROP_AUTH_PASSWORD	SQL Server logi password.
Server	DBPROP_INIT_DATASOURCE	Name of an insta of SQL Server in the organization
Trusted_Connection	DBPROP_AUTH_INTEGRATED	Accepts the strir "yes" and "no" a values.
UID	DBPROP_AUTH_USERID	SQL Server logi record name.
UseProcForPrepare	SSPROP_INIT_USEPROCFORPREP	Accepts 0, 1, and as values. This keyword is meaningful only when connecting SQL Server 6.5. ignored for any newer versions.
WSID	SSPROP_INIT_WSID	Workstation identifier.

#### **Sessions**

A SQLOLEDB session represents a single connection to an instance of Microsoft® SQL Server<sup>TM</sup> 2000.

OLE DB requires that sessions delimit transaction space for a data source. All command objects created from a specific session object participate in the local or distributed transaction of the session object.

The first session object created on the initialized data source receives the SQL Server connection established at initialization. When all references on the interfaces of the session object are released, the connection to the instance of SQL Server becomes available to another session object created on the data source.

An additional session object created on the data source establishes its own connection to the instance of SQL Server as specified by the data source. The connection to the instance of SQL Server is dropped when the application releases all references to objects created that session.

This example shows SQLOLEDB SQL Server connection usage:

```
int main()
  // Interfaces used in the example.
  IDBInitialize*
                  pIDBInitialize
                                   = NULL:
  IDBCreateSession* pIDBCreateSession = NULL;
  IDBCreateCommand* pICreateCmd1
                                         = NULL;
  IDBCreateCommand*
                        pICreateCmd2
                                         = NULL;
  IDBCreateCommand*
                        pICreateCmd3
                                         = NULL;
  // Initialize COM.
  if (FAILED(CoInitialize(NULL)))
  {
    // Display error from CoInitialize.
    return (-1);
```

```
}
// Get the memory allocator for this task.
if (FAILED(CoGetMalloc(MEMCTX_TASK, &g_pIMalloc)))
  // Display error from CoGetMalloc.
  goto EXIT;
// Create an instance of the data source object.
if (FAILED(CoCreateInstance(CLSID_SQLOLEDB, NULL,
  CLSCTX_INPROC_SERVER, IID_IDBInitialize, (void**)
  &pIDBInitialize)))
  // Display error from CoCreateInstance.
  goto EXIT;
// The InitFromPersistedDS function
// performs IDBInitialize->Initialize() establishing
// the first application connection to the instance of SQL Server.
if (FAILED(InitFromPersistedDS(pIDBInitialize, L"MyDataSource'
  NULL, NULL)))
{
  goto EXIT;
}
// The IDBCreateSession interface is implemented on the data source
// object. Maintaining the reference received maintains the
// connection of the data source to the instance of SQL Server.
if (FAILED(pIDBInitialize->QueryInterface(IID_IDBCreateSession
  (void**) &pIDBCreateSession)))
{
```

```
// Display error from pIDBInitialize.
  goto EXIT;
// Releasing this has no effect on the SQL Server connection
// of the data source object because of the reference maintained by
// pIDBCreateSession.
pIDBInitialize->Release();
pIDBInitialize = NULL;
// The session created next receives the SQL Server connection of
// the data source object. No new connection is established.
if (FAILED(pIDBCreateSession->CreateSession(NULL,
  IID_IDBCreateCommand, (IUnknown**) &pICreateCmd1)))
{
  // Display error from pIDBCreateSession.
  goto EXIT;
// A new connection to the instance of SQL Server is established to s
// next session object created. On successful completion, the
// application has two active connections on the SQL Server.
if (FAILED(pIDBCreateSession->CreateSession(NULL,
  IID_IDBCreateCommand, (IUnknown**) &pICreateCmd2)))
{
  // Display error from pIDBCreateSession.
  goto EXIT;
// pICreateCmd1 has the data source connection. Because the
// reference on the IDBCreateSession interface of the data source
// has not been released, releasing the reference on the session
// object does not terminate a connection to the instance of SQL Serv
```

```
// However, the connection of the data source object is now
  // available to another session object. After a successful call to
  // Release, the application still has two active connections to the
  // instance of SQL Server.
  pICreateCmd1->Release();
  pICreateCmd1 = NULL;
  // The next session created gets the SQL Server connection
  // of the data source object. The application has two active
  // connections to the instance of SQL Server.
  if (FAILED(pIDBCreateSession->CreateSession(NULL,
    IID_IDBCreateCommand, (IUnknown**) &pICreateCmd3)))
  {
    // Display error from pIDBCreateSession.
    goto EXIT;
EXIT:
  // Even on error, this does not terminate a SQL Server connection
  // because pICreateCmd1 has the connection of the data source
  // object.
  if (pICreateCmd1 != NULL)
    pICreateCmd1->Release();
  // Releasing the reference on pICreateCmd2 terminates the SQL
  // Server connection supporting the session object. The application
  // now has only a single active connection on the instance of SQL Se
  if (pICreateCmd2 != NULL)
    pICreateCmd2->Release();
  // Even on error, this does not terminate a SQL Server connection
  // because pICreateCmd3 has the connection of the
  // data source object.
```

```
if (pICreateCmd3 != NULL)
  pICreateCmd3->Release();
// On release of the last reference on a data source interface, the
// connection of the data source object to the instance of SQL Server
// The example application now has no SQL Server connections activ
if (pIDBCreateSession != NULL)
  pIDBCreateSession->Release();
// Called only if an error occurred while attempting to get a
// reference on the IDBCreateSession interface of the data source.
// If so, the call to IDBInitialize::Uninitialize terminates the
// connection of the data source object to the instance of SQL Server
if (pIDBInitialize != NULL)
  if (FAILED(pIDBInitialize->Uninitialize()))
     // Uninitialize is not required, but it fails if an
     // interface has not been released. Use it for
     // debugging.
  pIDBInitialize->Release();
if (g_pIMalloc != NULL)
  g_pIMalloc->Release();
CoUninitialize();
return (0);
```

Connecting SQLOLEDB session objects to an instance of SQL Server can

generate significant overhead for applications that continually create and release session objects. The overhead can be minimized by managing SQLOLEDB session objects efficiently. SQLOLEDB applications can keep the SQL Server connection of a session object active by maintaining a reference on at least one interface of the object.

For example, maintaining a pool of command creation object references keeps active connections for those session objects in the pool. As session objects are required, the pool maintenance code passes a valid **IDBCreateCommand** interface pointer to the application method requiring the session. When the application method no longer requires the session, the method returns the interface pointer back to the pool maintenance code rather than releasing the application's reference to the command creation object.

**Note** In the preceding example, the **IDBCreateCommand** interface is used because the **ICommand** interface implements the **GetDBSession** method, the only method in command or rowset scope that allows an object to determine the session on which it was created. Therefore, a command object, and only a command object, allows an application to retrieve a data source object pointer from which additional sessions can be created.

## **Session Properties**

SQLOLEDB interprets OLE DB session properties as follows.

Property ID	Description
DBPROP_SESS_AUTOCOMMITISOLEVELS	SQLOLEDB supports all
	autocommit transaction
	isolation levels with the
	exception of the chaos
	level,
	DBPROPVAL_TI_CHAOS.

In the provider-specific property set DBPROPSET\_SQLSERVERSESSION, SQLOLEDB defines the following additional session property.

Property ID	Description
SSPROP_QUOTEDCATALOGNAMES	Type: VT_BOOL R/W: Read/write Default: VARIANT_FALSE Description: Quoted identifiers allowed in CATALOG restriction. VARIANT_TRUE: Quoted identifiers are recognized for a catalog restriction for the schema rowsets that supply distributed query support.
	VARIANT_FALSE: Quoted identifiers are not recognized for a catalog restriction for the schema rowsets that supply distributed query support.
	For more information about schema rowsets that supply distributed query support, see

Distributed Query Support in Schema Rowsets.

#### **Persisted Data Source Objects**

SQLOLEDB supports persisted data source objects with the **IPersistFile** interface.

#### **Examples**

#### A. Persist data source initialization properties

This example shows a function that persists data source initialization properties defining a server, database, and the use of the Windows Authentication Mode for connection. The server name and database name are received in the *pLocation* and *pDatasource* parameters of the function.

```
HRESULT SetAndSaveInitProps
  IDBInitialize* pIDBInitialize,
  WCHAR* pDataSource,
  WCHAR* pCatalog,
  BOOL bUseWinNTAuth
  const ULONG
                nProps = 4;
  ULONG
               nSSProps;
  ULONG
               nPropSets;
  ULONG
               nProp;
  IDBProperties* pIDBProperties = NULL;
              pIPersistFile = NULL;
  IPersistFile*
               aInitProps[nProps];
  DBPROP
                aSSInitProps = NULL;
  DBPROP*
  DBPROPSET*
                  aInitPropSets = NULL;
  HRESULT
                hr;
    nSSProps = 0;
```

```
nPropSets = 1;
aInitPropSets = new DBPROPSET[nPropSets];
// Initialize common property options.
for (nProp = 0; nProp < nProps; nProp++)
  {
  VariantInit(&aInitProps[nProp].vValue);
  aInitProps[nProp].dwOptions = DBPROPOPTIONS REQUIRED
  aInitProps[nProp].colid = DB NULLID;
  }
// Level of prompting that will be done to complete the connection
// process.
aInitProps[0].dwPropertyID = DBPROP_INIT_PROMPT;
aInitProps[0].vValue.vt = VT I2;
aInitProps[0].vValue.iVal = DBPROMPT NOPROMPT;
// Server name.
aInitProps[1].dwPropertyID = DBPROP INIT DATASOURCE;
aInitProps[1].vValue.vt = VT_BSTR;
aInitProps[1].vValue.bstrVal = SysAllocString(pDataSource);
// Database.
aInitProps[2].dwPropertyID = DBPROP_INIT_CATALOG;
aInitProps[2].vValue.vt = VT_BSTR;
aInitProps[2].vValue.bstrVal = SysAllocString(pCatalog);
aInitProps[3].dwPropertyID = DBPROP_AUTH_INTEGRATED;
if (bUseWinNTAuth == TRUE)
  aInitProps[3].vValue.vt = VT_BSTR;
  aInitProps[3].vValue.bstrVal = SysAllocString(L"SSPI");
```

```
} //end if
// Now that properties are set, construct the PropertySet array.
aInitPropSets[0].guidPropertySet = DBPROPSET_DBINIT;
aInitPropSets[0].cProperties = nProps;
aInitPropSets[0].rgProperties = aInitProps;
// Set initialization properties
pIDBInitialize->QueryInterface(IID_IDBProperties,
  (void**) &pIDBProperties);
hr = pIDBProperties->SetProperties(nPropSets, aInitPropSets);
if (FAILED(hr))
  // Display error from failed SetProperties.
pIDBProperties->Release();
// Free references on OLE known strings.
for (nProp = 0; nProp < nProps; nProp++)
  {
  if (aInitProps[nProp].vValue.vt == VT_BSTR)
     SysFreeString(aInitProps[nProp].vValue.bstrVal);
  }
for (nProp = 0; nProp < nSSProps; nProp++)
  if (aSSInitProps[nProp].vValue.vt == VT_BSTR)
     SysFreeString(aInitProps[nProp].vValue.bstrVal);
  }
// Free dynamically allocated memory.
delete [] aInitPropSets;
delete [] aSSInitProps;
```

#### B. Use persisted data source initialization properties

This example uses a persisted data source object with additional initialization properties that provide a Microsoft® SQL Server<sup>TM</sup> 2000 login ID and password.

```
const ULONG     nPropSets = 1;
ULONG
              nProp;
IDBProperties* pIDBProperties = NULL;
IPersistFile* pIPersistFile = NULL;
DBPROP
               aInitProps[nProps];
                 aInitPropSets[nPropSets];
DBPROPSET
HRESULT
               hr;
// First load the persisted data source information.
pIDBInitialize->QueryInterface(IID_IPersistFile,
  (void**) &pIPersistFile);
hr = pIPersistFile->Load(pPersistedDSN, STGM_DIRECT);
if (FAILED(hr))
  // Display errors from IPersistFile interface.
pIPersistFile->Release();
if (FAILED(hr))
  return (hr);
  }
// Initialize common property options.
for (nProp = 0; nProp < nProps; nProp++)
  VariantInit(&aInitProps[nProp].vValue);
  aInitProps[nProp].dwOptions = DBPROPOPTIONS_REQUIRED
  aInitProps[nProp].colid = DB NULLID;
```

```
// Level of prompting that will be done to complete the connection
// process.
aInitProps[0].dwPropertyID = DBPROP_INIT_PROMPT;
aInitProps[0].vValue.vt = VT_I2;
aInitProps[0].vValue.iVal = DBPROMPT NOPROMPT;
// Now that properties are set, construct the PropertySet array.
aInitPropSets[0].guidPropertySet = DBPROPSET_DBINIT;
aInitPropSets[0].cProperties = nProps;
aInitPropSets[0].rgProperties = aInitProps;
// Set initialization properties
pIDBInitialize->QueryInterface(IID_IDBProperties,
  (void**) &pIDBProperties);
hr = pIDBProperties->SetProperties(nPropSets, aInitPropSets);
if (SUCCEEDED(hr))
  hr = pIDBInitialize->Initialize();
  if (FAILED(hr))
    DumpError(pIDBInitialize, IID_IDBInitialize);
else
  // Display error from failed SetProperties.
pIDBProperties->Release();
// Free references on OLE known strings.
for (nProp = 0; nProp < nProps; nProp++)
  if (aInitProps[nProp].vValue.vt == VT_BSTR)
```

```
SysFreeString(aInitProps[nProp].vValue.bstrVal);
}
return (hr);
}
```

The **IPersistFile::Save** method can be called before or after calling **IDBInitialize::Initialize**. Calling the method after a successful return from **IDBInitialize::Initialize** ensures persisting a valid data source specification.

### **Commands**

SQLOLEDB exposes the **ICommand** interface and command objects.

### **Command Syntax**

SQLOLEDB recognizes command syntax specified by the DBGUID\_SQL macro. For SQLOLEDB, the specifier indicates that an amalgam of ODBC SQL, SQL-92, and Transact-SQL is valid syntax. For example, the following SQL statement uses an ODBC SQL escape sequence to specify the LCASE string function:

SELECT customerid={fn LCASE(CustomerID)} FROM Customers

LCASE returns a character string, converting all uppercase characters to their lowercase equivalents. The SQL-92 string function LOWER performs the same operation, so the following SQL statement is a SQL-92 equivalent to the ODBC statement presented above:

SELECT customerid=LOWER(CustomerID) FROM Customers

SQLOLEDB processes either form of the statement successfully when specified as text for a command.

#### **Stored Procedures**

When executing a Microsoft® SQL Server™ 2000 stored procedure using a SQLOLEDB command, use the ODBC CALL escape sequence in the command text. SQLOLEDB then uses the remote procedure call mechanism of SQL Server 2000 to optimize command processing. For example, the following ODBC SQL statement is preferred command text over the Transact-SQL form:

- ODBC SQL {call SalesByCategory('Produce', '1995')}
- Transact-SQL
   EXECUTE SalesByCategory 'Produce', '1995'

#### **Command Parameters**

Parameters are marked in command text with the ODBC-specified question mark character. For example, the following ODBC SQL statement is marked for a single input parameter:

{call SalesByCategory('Produce', ?)}

To improve performance by reducing network traffic, SQLOLEDB does not automatically derive parameter information unless

ICommandWithParameters::GetParameterInfo or

**ICommandPrepare::Prepare** is called before executing a command. This means that SQLOLEDB does not automatically:

- Verify the correctness of the data type specified with **ICommandWithParameters::SetParameterInfo**.
- Map from the DBTYPE specified in the accessor binding information to the correct Microsoft® SQL Server<sup>TM</sup> 2000 data type for the parameter.

Applications will receive possible errors or loss of precision with either of these methods if they specify data types that are not compatible with the SQL Server 2000 data type of the parameter.

To ensure this does not happen, the application should:

- If hard-coding **ICommandWithParameters::SetParameterInfo**, ensure that *pwszDataSourceType* matches the SQL Server data type for the parameter.
- If hard-coding an accessor, ensure that the DBTYPE value being bound to the parameter is of the same type as the SQL Server data type for the parameter.
- Code the application to call **ICommandWithParameters::GetParameterInfo** so the provider can

obtain the SQL Server data types of the parameters dynamically. Note that this causes an extra network roundtrip to the server.

SQLOLEDB supports input parameters in SQL statement commands. On procedure-call commands, SQLOLEDB supports input, output, and input/output parameters. Output parameter values are returned to the application either on execution or when all returned rowsets are exhausted by the application. To ensure that returned values are valid, use **IMultipleResults** to force rowset consumption.

```
// Macro used in the example.
#define COUNTRY_MAX_CHARS
                                     15
// Structure supporting the parameters of the example stored procedure
typedef struct tagSPROCPARAMS
  long
          lReturnValue;
          acCountry[COUNTRY MAX CHARS + 1];
  char
  } SPROCPARAMS;
  // Interfaces used in the example.
  ICommandText*
                    pICommandText = NULL;
  ICommandWithParameters* pICommandWithParameters = NULL;
  IAccessor*
                 pIAccessor = NULL;
  IMultipleResults* pIMultipleResults = NULL;
  IRowset*
                pIRowset = NULL;
  // Command parameter data.
  DBPARAMS
                    Params;
  const ULONG
                   nParams = 2;
  DBPARAMBINDINFO rgParamBindInfo[nParams] =
    {
    L"DBTYPE I4",
    L"ReturnVal",
    sizeof(long),
```

```
DBPARAMFLAGS ISOUTPUT,
  11,
  0,
  L"DBTYPE_VARCHAR",
  L"@Country",
  COUNTRY_MAX_CHARS,
  DBPARAMFLAGS ISINPUT,
  0,
  0 };
              rgParamOrdinals[nParams] = {1,2};
ULONG
// Parameter accessor data.
HACCESSOR
                 hAccessor;
                 acDBBinding[nParams];
DBBINDING
DBBINDSTATUS
                   acDBBindStatus[nParams];
// The command and parameter data.
WCHAR*
               wszSQLString =
  L"{? = call CustomersInCountry(?)}";
                   sprocparams = {0, "USA"};
SPROCPARAMS
// Returned count of rows affected.
              cRowsAffected = 0;
LONG
HRESULT
               hr;
// Create the command.
if (FAILED(hr = pIDBCreateCommand->CreateCommand(NULL,
  IID_ICommandText, (IUnknown**) &pICommandText)))
  // Process error from IDBCreateCommand and return.
```

```
// Set the command text value.
if (FAILED(hr = pICommandText->SetCommandText(DBGUID_D)
  wszSQLString)))
  // Process error from ICommand and return.
// Get the ICommandWithParameters interface to set up parameter
// values.
if (FAILED(hr = pICommandText->QueryInterface(
  IID ICommandWithParameters,
  (void**) &pICommandWithParameters)))
  // Process error from ICommand and return.
// Set parameter information.
if (FAILED(hr = pICommandWithParameters->SetParameterInfo(nI
  rgParamOrdinals, rgParamBindInfo)))
  // Process error from ICommandWithParameters and return.
// Create parameter accessor, but first set binding structures
// to indicate the characteristics of each parameter.
for (ULONG i = 0; i < nParams; i++)
  acDBBinding[i].obLength = 0;
  acDBBinding[i].obStatus = 0;
  acDBBinding[i].pTypeInfo = NULL;
  acDBBinding[i].pObject = NULL;
  acDBBinding[i].pBindExt = NULL;
  acDBBinding[i].dwPart = DBPART VALUE;
```

```
acDBBinding[i].dwMemOwner = DBMEMOWNER_CLIENTOV
  acDBBinding[i].dwFlags = 0;
  acDBBinding[i].bScale = 0;
acDBBinding[0].iOrdinal = 1;
acDBBinding[0].obValue = offsetof(SPROCPARAMS, lReturnValue
acDBBinding[0].eParamIO = DBPARAMIO_OUTPUT;
acDBBinding[0].cbMaxLen = sizeof(long);
acDBBinding[0].wType = DBTYPE_I4;
acDBBinding[0].bPrecision = 11;
acDBBinding[1].iOrdinal = 2;
acDBBinding[1].obValue = offsetof(SPROCPARAMS, acCountry);
acDBBinding[1].eParamIO = DBPARAMIO_INPUT;
acDBBinding[1].cbMaxLen = COUNTRY_MAX_CHARS;
acDBBinding[1].wType = DBTYPE_STR;
acDBBinding[1].bPrecision = 0;
// Get the IAccessor interface, then create the accessor for
// the defined parameters.
pICommandWithParameters->QueryInterface(IID_IAccessor,
  (void**) &pIAccessor);
hr = pIAccessor->CreateAccessor(DBACCESSOR_PARAMETERI
  nParams, acDBBinding, sizeof(SPROCPARAMS), &hAccessor,
  acDBBindStatus);
if (FAILED(hr))
  // Process error from IAccessor and return.
// Fill the DBPARAMS structure for the command execution.
```

```
Params.pData = &sprocparams;
Params.cParamSets = 1;
Params.hAccessor = hAccessor;
// Execute the command.
if (FAILED(hr = pICommandText->Execute(NULL, IID_IMultipleI
  &Params, &cRowsAffected, (IUnknown**) &pIMultipleResults)
  // Process error from ICommand and return.
// For each rowset or count of rows affected...
do
  hr = ((IMultipleResults*) pIResults)->GetResult(NULL, 0,
    IID_IRowset, &cRowsAffected, (IUnknown**) &pIRowset);
  switch (hr)
    case S OK:
       {
       if (pIRowset != NULL)
         // Process data from the rowset and release.
         pIRowset->Release();
       else if (cRowsAffected != -1)
         printf("Command succeeded. %ld rows affected.\n\n",
           cRowsAffected);
       else
```

```
printf("Command succeeded.\n\n");
       break;
    case DB S NORESULT:
    case DB_S_STOPLIMITREACHED:
       break;
    default:
       DumpError(pIResults, IID_IMultipleResults);
       break;
while (hr == S_OK);
if (SUCCEEDED(hr))
  // At this point, the value of the return is guaranteed correct.
  // If any other output parameters had been specified, then they
  // too would now contain their correct values.
  printf("Return value %d\n", sprocparams.lReturnValue);
  }
```

The names of stored procedure parameters need not be specified in a DBPARAMBINDINFO structure. Use NULL for the value of the *pwszName* member to indicate that SQLOLEDB should ignore the parameter name and use only the ordinal specified in the *rgParamOrdinals* member of **ICommandWithParameters::SetParameterInfo**. If the command text contains both named and unnamed parameters, all the unnamed parameters must be specified before any named parameters.

If the name of a stored procedure parameter is specified, SQLOLEDB checks the name to ensure that it is valid. SQLOLEDB returns an error when it receives an erroneous parameter name from the consumer.

### **Preparing Commands**

SQLOLEDB supports command preparation for optimized multiple execution of a single command; however, command preparation generates overhead, and a consumer does not need to prepare a command to execute it more than once. In general, a command should be prepared if it will be executed more than three times.

For performance reasons, the command preparation is deferred until the command is executed. This is the default behavior. Any errors in the command being prepared are not known until the command is executed or a metaproperty operation is performed. Setting the Microsoft® SQL Server<sup>TM</sup> 2000 property SSPROP\_DEFERPREPARE to FALSE can turn off this default behavior.

In SQL Server 2000, when a command is executed directly (without preparing it first), an execution plan is created and cached. If the SQL statement is executed again, SQL Server has an efficient algorithm to match the new statement with the existing execution plan in the cache, and reuses the execution plan for that statement.

For prepared commands, SQL Server provides native support for preparing and executing command statements. When you prepare a statement, SQL Server creates an execution plan, caches it, and returns a handle to this execution plan to the provider. The provider then uses this handle to execute the statement repeatedly. No stored procedures are created. Because the handle directly identifies the execution plan for an SQL statement instead of matching the statement to the execution plan in the cache (as is the case for direct execution), it is more efficient to prepare a statement than to execute it directly, if you know the statement will be executed more than a few times.

In SQL Server 2000 and SQL Server version 7.0, the prepared statements cannot be used to create temporary objects and cannot reference system stored procedures that create temporary objects, such as temporary tables. These procedures must be executed directly.

When connected to SQL Server version 6.5, SQLOLEDB may create a temporary stored procedure when command text is prepared. Some commands should never be prepared. For example, commands that specify stored procedure

execution or include invalid text for SQL Server stored procedure creation should not be prepared.

If a temporary stored procedure is created, SQLOLEDB executes the temporary stored procedure, returning results as if the statement itself was executed.

Temporary stored procedure creation is controlled by the SQLOLEDB-specific initialization property SSPROP\_INIT\_USEPROCFORPREP. If the property value is either SSPROPVAL\_USEPROCFORPREP\_ON or SSPROPVAL\_USEPROCFORPREP\_ON\_DROP, SQLOLEDB attempts to create a stored procedure when a command is prepared. Stored procedure creation succeeds if the application user has sufficient SQL Server permissions.

For consumers that infrequently disconnect, creation of temporary stored procedures can require significant resources of **tempdb**, the SQL Server system database in which temporary objects are created. When the value of SSPROP\_INIT\_USEPROCFORPREP is SSPROPVAL\_USEPROCFORPREP\_ON, temporary stored procedures created by SQLOLEDB are dropped only when the session that created the command loses its connection to the instance of SQL Server. If that connection is the default connection created on data source initialization, the temporary stored procedure is dropped only when the data source becomes uninitialized.

When the value of SSPROP\_INIT\_USEPROCFORPREP is SSPROPVAL\_USEPROCFORPREP\_ON\_DROP, SQLOLEDB temporary stored procedures are dropped when one of the following occurs:

- The consumer uses **ICommandText::SetCommandText** to indicate a new command.
- The consumer uses **ICommandPrepare::Unprepare** to indicate that it no longer requires the command text.
- The consumer releases all references to the command object using the temporary stored procedure.

A command object has at most one temporary stored procedure in **tempdb**. Any existing temporary stored procedure represents the current command text of a specific command object.

### **Commands Generating Multiple-Rowset Results**

SQLOLEDB can return multiple rowsets from Microsoft® SQL Server™ 2000 statements. SQL Server 2000 statements return multiple-rowset results under the following conditions:

- Batched SQL statements are submitted as a single command.
- Stored procedures implement a batch of SQL statements.
- SQL statements include the Transact-SQL COMPUTE or COMPUTE BY clause.

#### **Batches**

SQLOLEDB recognizes the semicolon character as a batch delimiter for SQL statements:

Sending multiple SQL statements in one batch is more efficient than executing each SQL statement separately. Sending one batch reduces the network roundtrips from the client to the server.

#### Stored Procedures

SQL Server 2000 returns a result set for each statement in a stored procedure, so most SQL Server 2000 stored procedures return multiple result sets.

#### **COMPUTE BY and COMPUTE**

The Transact-SQL COMPUTE BY clause generates subtotals within a SELECT statement result set. The COMPUTE clause generates a total at the end of the result set. SQLOLEDB returns each COMPUTE BY subtotal and the COMPUTE total as a separate rowset result.

# Using IMultipleResults to Process Multiple Result Sets

In general, consumers should use the **IMultipleResults** interface to process the rowset or rowsets returned by SQLOLEDB command execution.

When SQLOLEDB submits a command for execution, Microsoft® SQL Server<sup>TM</sup> 2000 executes the statement or statements and returns any results. The complete process is a round trip between the client and the instance of SQL Server. Each client connection to an instance of SQL Server can have at most one active round trip. That is, within a SQLOLEDB session, only a single command object can be actively executing or returning results on the connection. This is the default result set behavior of SQL Server client connections.

To complete a round trip, a client must process all results from command execution. Because SQLOLEDB command execution can generate multiple-rowset objects as results, use the **IMultipleResults** interface to ensure that application data retrieval completes the client-initiated roundtrip.

The following Transact-SQL statement generates multiple rowsets, some containing row data from the **OrderDetails** table and some containing results of the COMPUTE BY clause:

```
SELECT OrderID, FullPrice = (UnitPrice * Quantity), Discount,
Discounted = UnitPrice * (1 - Discount) * Quantity
FROM OrderDetails
ORDER BY OrderID
COMPUTE
SUM(UnitPrice * Quantity), SUM(UnitPrice * (1 - Discount) * Quantity)
BY OrderID
```

If a consumer executes a command containing this text and requests a rowset as the returned results interface, only the first set of rows is returned. The consumer may process all rows in the rowset returned but if the DBPROP\_MULTIPLECONNECTIONS data source property is set to

VARIANT\_FALSE, until the command is canceled, no other commands can be executed on the session object (SQLOLEDB will not create another connection). SQLOLEDB returns a DB\_E\_OBJECTOPEN error if DBPROP\_MULTIPLECONNECTIONS is VARIANT\_FALSE and returns E FAIL if there is an active transaction.

If the connection is busy running a command that does not produce a rowset or produces a rowset that is not a server cursor and the DBPROP\_MULTIPLECONNECTIONS data source property is set to VARIANT\_TRUE, SQLOLEDB creates additional connections to support concurrent command objects unless a transaction is active, in which case it returns an error. Transactions and locking are managed by SQL Server 2000 on a per connection basis. If a second connection is generated, the command on the separate connections do not share locks. Care must be taken to ensure that one command does not block another by holding locks on rows requested by the other command.

The consumer can cancel the command either by using **ICommand::Cancel** or by releasing all references held on the command object and the derived rowset.

Using **IMultipleResults** in all instances allows the consumer to get all rowsets generated by command execution and allows consumers to appropriately determine when to cancel command execution and free a session object for use by other commands.

**Note** When you use SQL Server 2000 cursors, command execution creates the cursor. SQL Server 2000 returns success or failure on the cursor creation; therefore, the round trip to the instance of SQL Server is complete upon the return from command execution. Each **GetNextRows** call then becomes a round trip. In this way, multiple active command objects can exist, each processing a rowset that is the result of a fetch from the server cursor. For more information, see <u>Rowsets and SQL Server Cursors</u>.

#### **Rowsets**

A rowset is a set of rows that contain columns of data. Rowsets are central objects that enable all OLE DB data providers to expose result set data in tabular form.

After a consumer creates a session by using the

**IDBCreateSession::CreateSession** method, the consumer can use either the **IOpenRowset** or **IDBCreateCommand** interface on the session to create a rowset. The SQLOLEDB provider supports both of these interfaces. Both of these methods are described here.

• Create a rowset by calling the **IOpenRowset::OpenRowset** method.

This is equivalent to creating a rowset over a single table. This method opens and returns a rowset that includes all the rows from a single base table. One of the arguments to **OpenRowset** is a table ID that identifies the table from which to create the rowset.

 Create a command object by calling the IDBCreateCommand::CreateCommand method.

The command object executes commands that the provider supports. In SQLOLEDB, the consumer can specify any Transact-SQL statement (such as a SELECT statement or a call to a stored procedure). The steps for creating a rowset by using a command object are:

- 1. The consumer calls the IDBCreateCommand::CreateCommand method on the session to get a command object requesting the ICommandText interface on the command object. This ICommandText interface sets and retrieves the actual command text. The consumer fills in the text command by calling the ICommandText::SetCommandText method.
- 2. The user calls the **ICommand::Execute** method on the command. The rowset object built when the command executes contains the result set from the command.

The consumer can use the **ICommandProperties** interface to get or set the properties for the rowset returned by the command executed by the **ICommand::Execute** interfaces. The most commonly requested properties are the interfaces the rowset must support. In addition to interfaces, the consumer can request properties that modify the behavior of the rowset or interface.

Consumers release rowsets with the **IRowset::Release** method. Releasing a rowset releases any row handles held by the consumer on that rowset. Releasing a rowset does not release the accessors. If you have an **IAccessor** interface, it still has to be released.

### **Creating a Rowset with IOpenRowset**

SQLOLEDB supports the **IOpenRowset::OpenRowset** method with the following restrictions:

- A base table or view must be specified in a DBID structure that the *pTableID* parameter points to.
- The DBID *eKind* member must indicate DBKIND\_NAME.
- The DBID *uName* member must specify name of an existing base table or a view as a Unicode character string.
- The *pIndexID* parameter of **OpenRowset** must be NULL.

The result set of **IOpenRowset::OpenRowset** contains a single rowset. Result sets containing a single rowset can be supported by Microsoft® SQL Server™ 2000 cursors. Cursor support allows the developer to use SQL Server concurrency mechanisms.

### **Creating Rowsets with ICommand::Execute**

For rowsets created with the **ICommand::Execute** method, the properties desired in the resulting rowset can constrain the text of the command. This is especially critical for consumers that support dynamic command text.

SQLOLEDB cannot use Microsoft® SQL Server™ 2000 cursors to support the multiple-rowset results generated by many commands. If a consumer requests a rowset requiring SQL Server 2000 cursor support, an error occurs if the command text used generates more than a single rowset as its result. For more information, see Commands Generating Multiple-Rowset Results.

Scrollable SQLOLEDB rowsets are supported by SQL Server 2000 cursors. SQL Server 2000 imposes limitations on cursors that are sensitive to changes made by other users of the database. Specifically, the rows in some cursors cannot be ordered, and attempting to create a rowset by using a command containing an SQL ORDER BY clause can fail. For more information, see <a href="Rowsets and SQL Server Cursors">Rowsets and SQL Server Cursors</a>.

## **Rowset Properties and Behaviors**

These are the SQLOLEDB rowset properties.

Property ID	Description
	R/W: Read/write Default: VARIANT_FALSE Description: The behavior of a rov operation is determined by this pro
	VARIANT_FALSE: SQLOLEDB an abort operaton. The rowset objectivitually lost. It supports only <b>IUn</b> the release of outstanding row and
	VARIANT_TRUE: SQLOLEDB 1
	R/W: Read/write Default: DBPROPVAL_AO_RAN Description: Access order. Order i be accessed on the rowset.
	DBPROPVAL_AO_RANDOM: C in any order.
	DBPROPVAL_AO_SEQUENTIA Columns bound as storage objects sequential order determined by the
	DBPROPVAL_AO_SEQUENTIA accessed in sequential order deterrordinal.
DBPROP_APPENDONLY	This rowset property is not implen Attempting to read or write the proan error.
DBPROP_BLOCKINGSTORAGEOBJECTS	R/W: Read-only Default: VARIANT_TRUE Description: SQLOLEDB storage

	other rowset methods.
DBPROP_BOOKMARKS DBPROP_LITERALBOOKMARKS	R/W: Read/write Default: VARIANT_FALSE Description: SQLOLEDB support row identification when DBPROP DBPROP_LITERALBOOKMAR VARIANT_TRUE.
	Setting either property to VARIAN enable rowset positioning by book DBPROP_IRowsetLocate or DBP VARIANT_TRUE to create a row positioning by bookmark.
	SQLOLEDB uses a Microsoft® S cursor to support a rowset contain more information, see Rowsets an
	Note: Setting these properties in consumer attempts to open a rows.
DBPROP_BOOKMARKSKIPPED	R/W: Read-only Default: VARIANT_FALSE Description: SQLOLEDB returns DB_E_BADBOOKMARK if the cinvalid bookmark when positionin bookmarked rowset.
DBPROP_BOOKMARKTYPE	R/W: Read-only Default: DBPROPVAL_BMK_NU Description: SQLOLEDB implem only. A SQLOLEDB bookmark is type DBTYPE_UI4.
DBPROP_CACHEDEFERRED	This rowset property is not implen Attempting to read or write the proan error.

DBPROP_CANFETCHBACKWARDS DBPROP_CANSCROLLBACKWARDS	R/W: Read/write Default: VARIANT_FALSE Description: SQLOLEDB supports scrolling in nonsequential rowsets cursor-supported rowset when eith DBPROP_CANFETCHBACKWA
	DBPROP_CANSCROLLBACKW VARIANT_TRUE. For more informand SQL Server Cursors.
DBPROP_CANHOLDROWS	R/W: Read/write Default: VARIANT_FALSE Description: By default, SQLOLE DB_E_ROWSNOTRELEASED if to obtain more rows for a rowset w exist on those currently in the rows be altered.
	Setting both DBPROP_CANHOL DBPROP_IRowsetChange to VAF bookmarked rowset. If both proper VARIANT_TRUE, the <b>IRowsetL</b> available on the rowset and DBPR and DBPROP_LITERALBOOKM VARIANT_TRUE.
	SQLOLEDB rowsets containing b by SQL Server cursors.
DBPROP_CHANGEINSERTEDROWS	R/W: Read/write Default: VARIANT_FALSE Description: This property can onl VARIANT_TRUE if the rowset is cursor.
DBPROP_COLUMNRESTRICT	R/W: Read-only Default: VARIANT_FALSE Description: SQLOLEDB sets the VARIANT_TRUE when a column changed by the consumer. Other co

	may be updatable and the rows the deleted.
	When the property is VARIANT_' examines the <i>dwFlags</i> member of structure to determine whether the column can be written or not. For <i>dwFlags</i> exhibits DBCOLUMNFI
DBPROP_COMMANDTIMEOUT	R/W: Read/write Default: 0 Description: By default, SQLOLE the <b>ICommand::Execute</b> method
DBPROP_COMMITPRESERVE	R/W: Read/write Default: VARIANT_FALSE Description: The behavior of a rov operation is determined by this pro
	VARIANT_TRUE: SQLOLEDB r  VARIANT_FALSE: SQLOLEDB a commit operation. The rowset of virtually lost. It supports only <b>IUn</b> the release of outstanding row and
DBPROP_DEFERRED	R/W: Read/write Default: VARIANT_FALSE Description: When set to VARIAN attempts to use a server cursor for and <b>image</b> columns are not returne they are accessed by the application
DBPROP_DELAYSTORAGEOBJECTS	R/W: Read-only Default: VARIANT_FALSE Description: SQLOLEDB support mode on storage objects.
	Changes made to data in a sequent immediately submitted to SQL Searre committed based on the rowser

DBPROP_IAccessor DBPROP_IColumnsInfo DBPROP_IConvertType DBPROP_IRowset DBPROP_IrowsetInfo	R/W: Read-only Default: VARIANT_TRUE Description: SQLOLEDB supports rowsets.
DBPROP_IColumnsRowset	R/W: Read/write Default: VARIANT_TRUE Description: SQLOLEDB supports interface.
DBPROP_IconnectionPointContainer	R/W: Read/write Default: VARIANT_FALSE
DBPROP_IMultipleResults	R/W: Read/write Default: VARIANT_FALSE Description: SQLOLEDB supports interface.
DBPROP_IRowsetUpdate  DBPROP_IRowsetUpdate	R/W: Read/write Default: VARIANT_FALSE Description: SQLOLEDB supportant IRowsetUpdate interfaces.
	A rowset created with DBPROP_I VARIANT_TRUE exhibits immed behaviors.
	When DBPROP_IRowsetUpdate i DBPROP_IRowsetChange is also rowset exhibits delayed update mc
	SQLOLEDB uses a SQL Server 20 rowsets exposing either <b>IRowsetC IRowsetUpdate</b> . For more inform SQL Server Cursors.
DBPROP_IRowsetIdentity	R/W: Read/write Default: VARIANT_TRUE Description: SQLOLEDB supports interface. If a rowset supports this handles representing the same und

	reflect the same data and state. Co <b>IRowsetIdentity:: IsSameRow</b> m row handles to see if they refer to
DBPROP_IRowsetScroll  DBPROP_IRowsetScroll	R/W: Read/write Default: VARIANT_FALSE Description: SQLOLEDB can exp and <b>IRowsetScroll</b> interfaces.
	When DBPROP_IRowsetLocate is DBPROP_CANFETCHBACKWADBPROP_CANSCROLLBACKWATARIANT_TRUE.
	When DBPROP_IRowsetScroll is DBPROP_IRowsetLocate is also both interfaces are available on the
	Bookmarks are required for either sets DBPROP_BOOKMARKS an DBPROP_LITERALBOOKMAR VARIANT_TRUE when the constinterface.
	SQLOLEDB uses SQL Server 200  IRowsetLocate and IRowsetScro information, see Rowsets and SQI
	Setting these properties in conflict cursor-defining properties results i setting DBPROP_IRowsetScroll to when DBPROP_OTHERINSERT VARIANT_TRUE generates an erattempts to open a rowset.
DBPROP_IRowsetResynch	R/W: Read/write Default: VARIANT_FALSE Description: SQLOLEDB exposes interface on demand. SQLOLEDE interface on any rowset.
DBPROP_ISupportErrorInfo	R/W: Read/write

	Default: VARIANT_TRUE Description: SQLOLEDB exposes ISupportErrorInfo interface on r
DBPROP_IlockBytes	This interface is not implemented Attempting to read or write the proerror.
DBPROP_ISequentialStream	R/W: Read/write Default: VARIANT_FALSE Description: SQLOLEDB exposes interface to support long, variable- SQL Server 2000.
DBPROP_Istorage	This interface is not implemented Attempting to read or write the proerror.
DBPROP_Istream	This interface is not implemented Attempting to read or write the proerror.
DBPROP_IMMOBILEROWS	R/W: Read/write Default: VARIANT_TRUE Description: The property is only SQL Server keyset cursors; it is Voother cursors.
	VARIANT_TRUE: The rowset wi inserted or updated rows. For IRowsetChange::InsertRow, row of the rowset. For IRowsetChang rowset is not ordered, then the pos rows is not changed. If the rowset IRowsetChange::SetData change to order the rowset, the row is not build on a set of key columns (typeDBPROP_OTHERUPDATEDELIVARIANT_TRUE but DBPROP_VARIANT_TRUE but DBPROP_VARIANT_FALSE), changing the is generally equivalent to deleting inserting a new one. Thus, the row

	even disappear from the rowset (if DBPROP_OWNINSERT is VARI though the DBPROP_IMMOBILE VARIANT_TRUE.
	VARIANT_FALSE: If the rowset appear in the rowset's proper order ordered, the inserted row appears a <b>IRowsetChange::SetData</b> change to order the rowset, the row is movordered, then the position of the rowset.
DBPROP_LITERALIDENTITY	R/W: Read-only Default: VARIANT_TRUE Description: This property is alway
DBPROP_LOCKMODE	R/W: Read/write Default: DBPROPVAL_LM_NON Description: Level of locking perf (DBPROPVAL_LM_NONE, DBPROPVAL_LM_SINGLEROV
DBPROP_MAXOPENROWS	R/W: Read-only Default: 0 Description: SQLOLEDB does no rows that can be active in rowsets.
DBPROP_MAXPENDINGROWS	R/W: Read-only Default: 0 Description: SQLOLEDB does no rowset rows with changes pending
DBPROP_MAXROWS	R/W: Read/write Default: 0 Description: By default, SQLOLE number of rows in a rowset. When DBPROP_MAXROWS, SQLOLE ROWCOUNT statement to limit the rowset.
	SET ROWCOUNT can cause unir SQL Server 2000 statement execu

	information, see <u>SET ROWCOUN</u>
DBPROP_MAYWRITECOLUMN	This rowset property is not implen Attempting to read or write the proan error.
DBPROP_MEMORYUSAGE	This rowset property is not implen Attempting to read or write the proan error.
DBPROP_NOTIFICATIONGRANULARITY	This rowset property is not implen Attempting to read or write the proan error.
DBPROP_NOTIFICATIONPHASES	R/W: Read-only Default: DBPROPVAL_NP_OKT DBPROPVAL_NP_ABOUTTOD DBPROPVAL_NP_SYNCHAFTE DBPROPVAL_NP_FAILEDTOD DBPROPVAL_NP_DIDEVENT Description: SQLOLEDB support
DBPROP_NOTIFYCOLUMNSET	R/W: Read-only
DBPROP_NOTIFYROWDELETE	Default: DBPROPVAL_NP_OKT
DBPROP NOTIFYROWFIRSTCHANGE	DBPROPVAL_NP_ABOUTTOD(
DBPROP_NOTIFYROWINSERT	Description: SQLOLEDB notifica
DBPROP_NOTIFYROWRESYNCH	cancelable prior to an attempt to p
DBPROP_NOTIFYROWSETRELEASE	modification indicated. SQLOLEI
DBPROP_NOTIFYROWSETFETCH-	phase cancellation after the attemp
POSITIONCHANGE	
DBPROP_NOTIFYROWUNDOCHANGE	
DBPROP_NOTIFYROWUNDODELETE	
DBPROP_NOTIFYROWUNDOINSERT	
DBPROP_NOTIFYROWUPDATE	
DBPROP_ORDEREDBOOKMARKS	This rowset property is not implen
	Attempting to read or write the proan error.
DBPROP_OTHERINSERT	R/W: Read/write
DBPROP_OTHERUPDATEDELETE	Default: VARIANT_FALSE
DBPROP_OWNINSERT	Description: Setting change visibil
DBPROP_OWNUPDATEDELETE	SQLOLEDB to use SQL Server 20

	the rowset. For more information, Server Cursors.
DBPROP_QUICKRESTART	R/W: Read/write Default: VARIANT_FALSE Description: When set to VARIAN attempts to use a server cursor for
DBPROP_REENTRANTEVENTS	R/W: Read-only Default: VARIANT_TRUE Description: SQLOLEDB rowsets return DB_E_NOTREENTRANT to access a nonreentrant rowset me callback.
DBPROP_REMOVEDELETED	R/W: Read/write Default: VARIANT_FALSE Description: SQLOLEDB alters the based on the visibility of changes that a exposed by the rowset.
	VARIANT_TRUE: Rows deleted other SQL Server users are remove the rowset is refreshed. DBPROP_VARIANT_TRUE.
	VARIANT_FALSE: Rows deleted other SQL Server 2000 users are n rowset when the rowset is refreshe for deleted SQL Server rows in the DBROWSTATUS_E_DELETED. DBPROP_OTHERINSERT is VA
	This property only has value for re SQL Server 2000 cursors. For more Rowsets and SQL Server Cursors.
	When the DBPROP_REMOVEDI implemented on a keyset cursor ro removed at fetch time and it is posmethods (such as <b>GetNextRows</b> a return both S_OK and fewer rows

	that this behavior does not signify DB_S_ENDOFROWSET condition of rows returned will never be zero remaining rows.
DBPROP_REPORTMULTIPLECHANGES	This rowset property is not implen Attempting to read or write the proan error.
DBPROP_RETURNPENDINGINSERTS	R/W: Read-only Default: VARIANT_FALSE Description: When a method that 1 SQLOLEDB does not return pend
DBPROP_ROWRESTRICT	R/W: Read-only Default: VARIANT_TRUE Description: SQLOLEDB rowsets rights based on the row. If the <b>IRo</b> exposed on a rowset, then the <b>SetI</b> called by the consumer.
DBPROP_ROWSET_ASYNCH	This rowset property is not implen Attempting to read or write the proan error.
DBPROP_ROWTHREADMODEL	R/W: Read-only Default: DBPROPVAL_RT_FREE Description: SQLOLEDB support from multiple execution threads of
DBPROP_SERVERCURSOR	R/W: Read/write Default: VARIANT_FALSE Description: When set, a SQL Serto support the rowset. For more in and SQL Server Cursors.
DBPROP_SERVERDATAONINSERT	R/W: Read/write Default: VARIANT_FALSE Description: Server data on insert. VARIANT_TRUE: At the time an the server, the provider retrieves d update the local row cache.

	VARIANT_FALSE: The provider values for newly inserted rows.
DBPROP_STRONGIDENTITY	R/W: Read-only Default: VARIANT_TRUE Description: Strong row identity. I a rowset (either <b>IRowsetChange</b> of true), and DBPROP_UPDATABIL InsertRows, then the value of DBPROP_STRONGIDENTITY d DBPROP_CHANGEINSERTEDF VARIANT_FALSE if DBPROP_CHANGEINSERTEDF VARIANT_FALSE).
DBPROP_TRANSACTEDOBJECT	R/W: Read-only Default: VARIANT_FALSE Description: SQLOLEDB support objects. For more information, see
DBPROP_UNIQUEROWS	R/W: Read/write Default: VARIANT_FALSE Description: Unique rows.  VARIANT_TRUE: Each row is uncolumn values. The set of columns identify the row have the DBCOLUMNFLAGS_KEYCOLUDBCOLUMNINFO structure retunced to the column of th
	VARIANT_FALSE: Rows may or identified by their column values. or may not be flagged with DBCOLUMNFLAGS_KEYCOLU
DBPROP_UPDATABILITY	R/W: Read/write Default: 0 Description: SQLOLEDB support DBPROP_UPDATABILITY value DBPROP_UPDATABILITY does

rowset. To make a rowset modifial
DBPROP_IRowsetChange or DBl

SQLOLEDB defines the provider-specific property set DBPROPSET\_SQLSERVERROWSET as shown in this table.

Property ID	Description
SSPROP_DEFERPREPARE	Column: No R/W: Read/Write Type: VT_BOOL Default: VARIANT_TRUE Description: VARIANT_TRUE: In prepared execution, the command preparation is deferred until <b>Icommand::Execute</b> is called or a metaproperty operation is performed. If the property is set to
	VARIANT_FALSE: The statement is prepared when <b>ICommandPrepare::Prepare</b> is executed.
SSPROP_IRowsetFastLoad	Column: No R/W: r/w Type: VT_BOOL Default: VARIANT_FALSE Description: Set this property to VARIANT_TRUE to open a fast load rowset through IopenRowset::OpenRowset(). You cannot set this property in IcommandProperties::SetProperties().
SSPROP_MAXBLOBLENGTH	

size of the text returned by the server.
Therefore, it is set to the maximum, for example, 2147483647.
Description: SQLOLEDB executes a SET TEXTSIZE statement to restrict the length of BLOB data returned in a SELECT statement.

#### **See Also**

**SET TEXTSIZE** 

**WRITETEXT** 

## **Rowsets and SQL Server Cursors**

Microsoft® SQL Server<sup>TM</sup> 2000 returns result sets to consumers using two methods:

- Default result sets, which:
  - Minimize overhead.
  - Provide maximal performance in fetching data.
  - Support only the default forward-only, read-only cursor functionality.
  - Return rows to the consumer one row at a time.
  - Support only one active statement at a time on a connection.

    After a statement has been executed, no other statements can be executed on the connection until all of the results have been retrieved by the consumer, or the statement has been canceled.
  - Support all Transact-SQL statements.
- Server cursors, which:
  - Support all cursor functionality.
  - Can return blocks of rows to the consumer.
  - Support multiple active statements on a single connection.
  - Balance cursor functionality against performance.
     The support for cursor functionality can decrease performance

relative to a default result set. This can be offset if the consumer can use cursor functionality to retrieve a smaller set of rows.

• Do not support any Transact-SQL statement that returns more than a single result set.

Consumers can request different cursor behaviors in a rowset by setting certain rowset properties. If the consumer does not set any of these rowset properties, or sets them all to their default values, SQLOLEDB implements the rowset using a default result set. If any one of these properties is set to a value other than the default, SQLOLEDB implements the rowset using a server cursor.

The following rowset properties direct SQLOLEDB to use SQL Server 2000 cursors. Some properties can be safely combined with others. For example, a rowset that exhibits the DBPROP\_IRowsetScroll and DBPROP\_IRowsetChange properties will be a bookmark rowset exhibiting immediate update behavior. Other properties are mutually exclusive. For example, a rowset exhibiting DBPROP\_OTHERINSERT cannot contain bookmarks.

Property ID	Value	Rowset behavior
DBPROP_SERVERCURSOR	VARIANT_TRUE	Cannot update SQI
		2000 data through
		The rowset is seque
		supporting forward
		and fetching only. 1
		row positioning is s
		Command text can
		ORDER BY clause
DBPROP_CANSCROLLBACKWARDS	VARIANT_TRUE	Cannot update SQI
or		2000 data through
DBPROP_CANFETCHBACKWARDS		The rowset support
		and fetching in eith
		direction. Relative
		positioning is supp
		Command text can
		ORDER BY clause
DBPROP_BOOKMARKS or	VARIANT_TRUE	Cannot update SQI

DBPROP_LITERALBOOKMARKS		2000 data through the rowset is seque supporting forward and fetching only. I row positioning is a Command text can ORDER BY clause
DBPROP_OWNUPDATEDELETE or DBPROP_OWNINSERT or DBPROP_OTHERUPDATEDELETE	VARIANT_TRUE	Cannot update SQI data through the rorowset supports scr fetching in either d Relative row positi supported. Comma contain an ORDER clause.
DBPROP_OTHERINSERT	VARIANT_TRUE	Cannot update SQI 2000 data through the rowset support and fetching in eith direction. Relative positioning is suppositioning in the reference columns.  DBPROP_OTHER cannot be VARIAN if the rowset contain bookmarks. Attempore a rowset with visibility property a bookmarks results in the rowset supposition in the rowset contains the rowset with visibility property a bookmarks results.
DBPROP_IRowsetLocate or DBPROP_IrowsetScroll	VARIANT_TRUE	Cannot update SQI 2000 data through The rowset support and fetching in eith

		direction. Bookman absolute positionin the IRowsetLocate are supported in the Command text can ORDER BY clause DBPROP_IRowset DBPROP_IRowset require bookmarks rowset. Attempting rowset with bookm DBPROP_OTHER set to VARIANT_I results in an error.
DBPROP_IRowsetChange or DBPROP_IRowsetUpdate	VARIANT_TRUE	Can update SQL So data through the ro rowset is sequentia supporting forward and fetching only. I row positioning is a All the commands updatable cursors of these interfaces.
DBPROP_IRowsetLocate or DBPROP_IRowsetScroll and DBPROP_IRowsetChange or DBPROP_IRowsetUpdate	VARIANT_TRUE	Can update SQL Sethrough the rowset. rowset supports scrifetching in either d Bookmarks and abspositioning through IRowsetLocate are in the rowset. Com can contain an ORl clause.
DBPROP_IMMOBILEROWS	VARIANT_FALSE	Cannot update SQI 2000 data through The rowset support

		scrolling only. Relapositioning is suppositioning is suppositioning is suppositioning is suppositioned for the reference columns.  DBPROP_IMMOF is only available in that can show SQL 2000 rows inserted commands on other or by other users. A to open a rowset with property set to VARIANT_FALSE rowset for which DBPROP_OTHER cannot be VARIAN results in an error.
DBPROP_REMOVEDELETED	VARIANT_TRUE	Cannot update SQI 2000 data through The rowset support scrolling only. Relapositioning is suppositioning is suppommand text can ORDER BY clause constrained by another property.

A SQLOLEDB rowset supported by a server cursor can be easily created on a SQL Server 2000 base table or view by using the **IOpenRowset::OpenRowset** method. Specify the table or view by name, passing the required rowset property sets in the *rgPropertySets* parameter.

Command text that creates a rowset is restricted when the consumer requires that

the rowset be supported by a server cursor. Specifically, the command text is restricted to either a single SELECT statement that returns a single rowset result, or a stored procedure that implements a single SELECT statement returning a single rowset result.

These two tables show the mappings of various OLE DB properties and the cursor models. They also show which rowset properties should be set to use certain type of cursor model.

Each cell in the table contains a value of the rowset property for the specific cursor model. The data type of the rowset properties listed above are all VT\_BOOL and the default values are VARIANT\_FALSE. The following symbols are used in the table.

F = default value (VARIANT\_FALSE)

 $T = VARIANT_TRUE$ 

- = VARIANT\_TRUE or VARIANT\_FALSE

To use a certain type of cursor model, locate the column corresponding the cursor model, and find all the rowset properties with value 'T' in the column. Set these rowset properties to VARIANT\_TRUE to use the specific cursor model. The rowset properties with '-' as a value can be set to either VARIANT\_TRUE or VARIANT\_FALSE.

		Forward-		Keyset
Rowset properties/Cursor models	set (RO)	only (RO)		driven (RO)
DBPROP_SERVERCURSOR	F	T	T	T
DBPROP_DEFERRED	F	F	-	-
DBPROP_IrowsetChange	F	F	F	F
DBPROP_IrowsetLocate	F	F	-	-
DBPROP_IrowsetScroll	F	F	-	-
DBPROP_IrowsetUpdate	F	F	F	F
DBPROP_BOOKMARKS	F	F	-	-
DBPROP_CANFETCHBACKWARDS	F	F	-	-
DBPROP_CANSRCOLLBACKWARDS	F	F	-	-

DBPROP_CANHOLDROWS	F	F	-	-
DBPROP_LITERALBOOKMARKS	F	F	-	-
DBPROP_OTHERINSERT	F	Т	F	F
DBPROP_OTHERUPDATEDELETE	F	Т	F	Т
DBPROP_OWNINSERT	F	Т	F	Т
DBPROP_OWNUPDATEDELETE	F	Т	F	Т
DBPROP_QUICKSTART	F	F	-	-
DBPROP_REMOVEDELETED	F	F	F	-
DBPROP_IrowsetResynch	F	F	F	-
DBPROP_CHANGEINSERTEDROWS	F	F	F	F
DBPROP_SERVERDATAONINSERT	F	F	F	-
DBPROP_UNIQUEROWS	-	F	F	F
DBPROP_IMMOBILEROWS	-	_	-	$ \mathbf{T} $

	Dynamic	Keyset	Dynamic
Rowset properties/Cursor models	(RO)	(R/W)	(R/W)
DBPROP_SERVERCURSOR	T	T	T
DBPROP_DEFERRED	_	_	-
DBPROP_IrowsetChange	F	_	-
DBPROP_IrowsetLocate	F	-	F
DBPROP_IrowsetScroll	F	_	F
DBPROP_IrowsetUpdate	F	-	-
DBPROP_BOOKMARKS	F	-	F
DBPROP_CANFETCHBACKWARDS	-	-	-
DBPROP_CANSRCOLLBACKWARDS	-	-	-
DBPROP_CANHOLDROWS	F	-	F
DBPROP_LITERALBOOKMARKS	F	-	F
DBPROP_OTHERINSERT	Т	F	T
DBPROP_OTHERUPDATEDELETE	Т	Т	T
DBPROP_OWNINSERT	Т	Т	T
DBPROP_OWNUPDATEDELETE	Т	Т	T
DBPROP_QUICKSTART	-	-	-

DBPROP_REMOVEDELETED	Т	-	Т
DBPROP_IrowsetResynch	-	_	-
DBPROP_CHANGEINSERTEDROWS	F	-	F
DBPROP_SERVERDATAONINSERT	F	-	F
DBPROP_UNIQUEROWS	F	F	F
DBPROP_IMMOBILEROWS	F	Т	F

For a given set of rowset properties, which cursor model is selected is determined as follows.

From the given collection of rowset properties, obtain a subset of properties that is listed in the above tables. Divide these properties into two subgroups depending on the flag value (required (T, F) or optional (-)) of each of the rowset properties listed in the above tables. For each cursor model from left to right (starting from the first table), compare the values of the properties in the two subgroups with the values of the corresponding properties at that column. The cursor model that has no mismatch with the required properties and the least number of mismatches with the optional properties is selected. If there is more than one cursor model, the leftmost is chosen.

#### **SQL Server Cursor Block Size**

When a SQL Server 2000 cursor supports a SQLOLEDB rowset, the number of elements in the row handle array parameter of the **IRowset::GetNextRows** or the **IRowsetLocate::GetRowsAt** methods defines the cursor block size. The rows indicated by the handles in the array are the members of the cursor block.

For rowsets supporting bookmarks, the row handles retrieved by using the **IRowsetLocate::GetRowsByBookmark** method define the members of the cursor block.

Regardless of the method used to populate the rowset and form the SQL Server 2000 cursor block, the cursor block is active until the next row-fetching method is executed on the rowset.

#### To obtain FAST\_FORWARD cursor

### **Fetching Rows**

The **IRowset** interface is the base rowset interface. The **IRowset** interface provides methods for fetching rows sequentially, getting the data from those rows, and managing rows. Consumers use the methods in **IRowset** for all basic rowset operations, including fetching and releasing rows and getting column values.

When a consumer gets an interface pointer on a rowset, usually the first step is to determine the capabilities of the rowset by using the

**IRowsetInfo::GetProperties** method. This returns information about the interfaces exposed by the rowset as well as capabilities of the rowset that do not show up as distinct interfaces, such as the maximum number of active rows and how many rows can have pending updates at the same time.

The next step for consumers is to determine the characteristics, or metadata, of the columns in the rowset. For this they use the **IColumnsInfo** or **IColumnsRowset** methods, for simple or extended column information, respectively. The **GetColumnInfo** method returns:

- The number of columns in the result set.
- An array of DBCOLUMNINFO structures, one per column.
  - The order of the structures is the order in which the columns appear in the rowset. Each DBCOLUMNINFO structure includes column meta data, such as column name, ordinal of the column, maximum possible length of a value in the column, data type of the column, precision, and length.
- The pointer to a storage for all string values within a single allocation block.

The consumer determines which columns it needs, either from the meta data or on the basis of the text command that generated the rowset. It determines the ordinals of the needed columns from the ordering of the column information returned by **IColumnsInfo** or from the ordinals in the column meta data rowset returned by **IColumnsRowset**.

The **IColumnsRowset** and **IColumnsInfo** interfaces are used to extract information about the columns in the rowset. The **IColumnsInfo** interface returns a limited set of information, whereas **IColumnsRowset** provides all the meta data.

**Note** In SQL Server version 7.0 and earlier, the optional meta data column **DBCOLUMN\_COMPUTEMODE** returned by

**IColumnsInfo::GetColumnsInfo** returns DBSTATUS\_S\_ISNULL (instead of the values describing if the column is computed or not) because it cannot be determined if the underlying column is computed column or not.

The ordinals are used to specify a binding to a column. A binding is a structure that associates an element of the consumer's structure with a column. The binding can bind the data value, length, and status value of the column.

A set of bindings is gathered together in an accessor, which is created with the **IAccessor::CreateAccessor** method. An accessor can contain multiple bindings so that the data for multiple columns can be retrieved or set in a single call. The consumer can create several accessors to match different usage patterns in different parts of the application. It can create and release accessors at any time while the rowset remains in existence.

To fetch rows from the database, the consumer calls a method, such as IRowset::GetNextRows or IRowsetLocate::GetRowsAt. These fetch operations put row data from the server into the row buffer of the provider. The consumer does not have direct access to the row buffer of the provider. The consumer uses IRowset::GetData to copy data from the buffer of the provider to the consumer buffer and IRowsetChange::SetData to copy data changes from the consumer buffer to the provider buffer.

The consumer calls the **GetData** method and passes it the handle to a row, the handle to an accessor, and a pointer to a consumer-allocated buffer. **GetData** converts the data and returns the columns as specified in the bindings used to create the accessor. The consumer can call **GetData** more than once for a row, using different accessors and buffers; therefore, the consumer can have multiple copies of the same data.

Data from variable-length columns can be treated several ways. First, such columns can be bound to a finite section of the consumer's structure, which causes truncation when the length of the data exceeds the length of the buffer.

The consumer can determine that truncation has occurred by checking for the status DBSTATUS\_S\_TRUNCATED. The returned length is always the true length in bytes, so the consumer also can determine how much data was truncated.

When the consumer is finished fetching or updating rows, it releases them with the **ReleaseRows** method. This releases resources from the copy of the rows in the rowset and makes room for new rows. The consumer can then repeat its cycle of fetching or creating rows and accessing the data in them.

When the consumer is done with the rowset, it calls the IAccessor::ReleaseAccessor method to release any accessor. It calls the IUnknown::Release method on all interfaces exposed by the rowset to release the rowset. When the rowset is released, it forces the release of any remaining rows or accessors the consumer may hold.

#### **Next Fetch Position**

The SQLOLEDB provider keeps track of the next fetch position so that a sequence of calls to the **GetNextRows** method (with no skips, changes of direction, or intervening calls to the **FindNextRow**, **Seek**, or **RestartPosition** methods) reads the entire rowset without skipping or repeating any row. The next fetch position is changed either by calling **IRowset::GetNextRows**, **IRowset::RestartPosition**, or **IRowsetIndex::Seek**, or by calling **FindNextRow** with a null *pBookmark* value. Calling **FindNextRow** with a nonnull *pBookmark* value has no effect on the next fetch position.

To fetch rows from a result set

## Fetching a Single Row Using IRow

**IRow** interface implementation in SQLOLEDB is simplified to increase performance. **IRow** allows direct access to columns of a single row object. If you know ahead of time that the result of a command execution will produce exactly one row, **IRow** will retrieve the columns of that row. If the result set includes multiple rows, **IRow** will expose only the first row.

**IRow** implementation does not allow any navigation of the row. Each column in the row is accessed only once, with one exception: a column can be accessed twice, once to find the column size, and again to fetch the data.

**IRow::Open** supports only DBGUID\_STREAM and DBGUID\_NULL type of objects to be opened.

To obtain a row object using **Icommand::Execute**, method **IID\_IRow** must be passed.

**IMultipleResults** must be used to handle multiple result sets. **IMultipleResults** supports **IRow** and **IRowset**. **IRowset** is used for bulk operations.

## **Using IRow::GetColumns**

**IRow** implementation allows forward only sequential access to the columns. You can either access all the columns in the row with a single call to **IRow::GetColumns**, or call **IRow::GetColumns** multiple times each time accessing few columns in the row.

The multiple calls to **IRow::GetColumns** should not overlap. For example, if the first call to **IRow::GetColumns** retrieves columns 1, 2, and 3, the second call to **IRow::GetColumns** should call for columns 4, 5, and 6. If subsequent calls to **IRow::GetColumns** overlap, the status flag (dwstatus field in DBCOLUMNACCESS) will be set to **DBSTATUS\_E\_UNAVAILABLE**.

To fetch columns using IRow::GetColumns

# **Fetching BLOB Data Using IRow**

BLOB column in a row object can be retrieved using **IRow::GetColumns** or **IRow::Open** and **ISequentialStream.** 

# Fetching BLOB Data Using IRow::GetColumns and ISequentialStream

The following function uses **IRow::GetColumns** and **ISequentialStream** to fetch large data.

```
void InitializeAndExecuteCommand()
  ulong iidx;
  WCHAR* wCmdString=OLESTR(" SELECT * FROM MyTable");
  // Do the initialization, create the session, and set command text
  hr=pICommandText->Execute(NULL, IID_IRow, NULL,
             &cNumRows,(Iunknown **)&pIRow)))
  //Get 1 column at a time
  for(ulong i=0; i < NoOfColumns; i++)
   GetSequentialColumn(pIRow, iidx);
  //do the clean up
HRESULT GetSequentialColumn(IRow* pUnkRow, ULONG iCol)
  HRESULT hr = NOERROR;
  ULONG cbRead = 0;
  ULONG cbTotal = 0;
  ULONG cColumns = 0;
  ULONG cReads = 0;
  ISequentialStream* pIStream = NULL;
  WCHAR* pBuffer[kMaxBuff];//50 chars read by ISequentialStream
  DBCOLUMNINFO* prgInfo;
  OLECHAR* pColNames;
  IColumnsInfo* pIColumnsInfo;
  DBID columnid;
```

```
DBCOLUMNACCESS column;
 hr = pUnkRow->QueryInterface(IID_IColumnsInfo,
              (void**) &pIColumnsInfo);
 if(FAILED(hr))
   goto CLEANUP;
 hr = pIColumnsInfo->GetColumnInfo(&cColumns, &prgInfo, &pCo
 //Get Column ID
 columnid = (prgInfo + (iCol))->columnid;
 IUnknown* pUnkStream = NULL;
 ZeroMemory(&column, sizeof(column));
 column.columnid = prgInfo[iCol].columnid;
 // Ask for Iunknown interface pointer
 column.wType = DBTYPE IUNKNOWN;
 column.pData = (LPVOID*) &pUnkStream;
 hr = pUnkRow->GetColumns(1, &column);
 //Get ISequentialStream from Iunknown pointer retrieved from
 //GetColumns()
 hr = pUnkStream->QueryInterface(IID_ISequentialStream,
                  (LPVOID*) &pIStream);
 ZeroMemory(pBuffer, kMaxBuff * sizeof(WCHAR));
 //Read 50 chars at a time until no more data.
 do
   hr = pIStream->Read(pBuffer, kMaxBuff, &cbRead);
   cbTotal = cbTotal + cbRead:
   //Process the data
 } while(cbRead > 0);
//Do the cleanup.
 return hr;
```

To fetch large data using IRow::GetColumns (or IRow::Open) and

# **ISequentialStream**

# Fetching BLOB Data Using IRow::Open and ISequentialStream

**IRow::Open** supports only DBGUID\_STREAM and DBGUID\_NULL type of objects to be opened.

The following function uses **IRow::Open** and **ISequentialStream** to fetch large data.

Large data can be bound or retrieved by using the **ISequentialStream** interface. For bound columns, the status flag indicates if the data is truncated by setting **DBSTATUS\_S\_TRUNCATED**.

```
void InitializeAndExecuteCommand()
  ulong iidx;
  WCHAR* wCmdString=OLESTR(" SELECT * FROM MyTable");
  // Do the initialization, create the session, and set command text
  hr=pICommandText->Execute(NULL, IID_IRow, NULL,
             &cNumRows,(Iunknown **)&pIRow)))
  //Get 1 column at a time
  for(ulong i=1; i <= NoOfColumns; i++)</pre>
   GetSequentialColumn(pIRow, iidx);
  //do the clean up
HRESULT GetSequentialColumn(IRow* pUnkRow, ULONG iCol)
  HRESULT hr = NOERROR:
  ULONG cbRead = 0:
  ULONG cbTotal = 0;
  ULONG cColumns = 0;
  ULONG cReads = 0;
  ISequentialStream* pIStream = NULL;
```

```
WCHAR* pBuffer[kMaxBuff];//50 chars read by ISequentialStream
  DBCOLUMNINFO* prgInfo;
  OLECHAR* pColNames;
  IColumnsInfo* pIColumnsInfo;
  DBID columnid:
  DBCOLUMNACCESS column;
  hr = pUnkRow->QueryInterface(IID_IColumnsInfo,
               (void**) &pIColumnsInfo);
  hr = pIColumnsInfo->GetColumnInfo(&cColumns, &prgInfo, &pCo
  //Get Column ID
  columnid = (prgInfo + (iCol - 1))->columnid;
  //Get sequential stream object by calling IRow::Open
  hr = pUnkRow->Open(NULL, &columnid, DBGUID_STREAM, 0,
           IID_ISequentialStream,(LPUNKNOWN *)&pIStream);
  ZeroMemory(pBuffer, kMaxBuff * sizeof(WCHAR));
  //Read 50 chars at a time until no more data.
  do
  {
    hr = pIStream->Read(pBuffer, kMaxBuff, &cbRead);
    cbTotal = cbTotal + cbRead;
    //Process the data
  } while(cbRead > 0);
// do the clean up
  return hr;
To fetch large data using IRow::GetColumns (or IRow::Open) and
ISequentialStream
```

}

#### **Bookmarks**

Bookmarks allow consumers to return quickly to a row. With bookmarks, consumers can access rows randomly based on the bookmark value. The bookmark column is column 0 in the rowset. The consumer sets the dwFlag field value of the binding structure to DBCOLUMNSINFO\_ISBOOKMARK to indicate that the column is used as bookmark. The consumer also sets the rowset property DBPROP\_BOOKMARKS to VARIANT\_TRUE. This allows column 0 to be present in the rowset. The <code>IRowsetLocate::GetRowsAt</code> method is then used to fetch rows, starting with the row specified as an offset from a bookmark.

To retrieve rows using bookmarks

### **Running Stored Procedures (OLE DB)**

A stored procedure is an executable object stored in a database. Microsoft® SQL Server<sup>TM</sup> 2000 supports:

Stored procedures

One or more SQL statements that have been precompiled into a single executable procedure.

• Extended stored procedures

C or C++ DLLs written to the SQL Server Open Data Services API for extended stored procedures. The Open Data Services API extends the capabilities of stored procedures to include C or C++ code.

When executing statements, calling a stored procedure on the data source (instead of executing or preparing a statement in the client application directly) can provide:

- Higher performance.
- Reduced network overhead.
- Better consistency.
- Better accuracy.
- Added functionality.

The OLE DB provider supports three of the mechanisms that SQL Server 2000 stored procedures use to return data:

- Every SELECT statement in the procedure generates a result set.
- The procedure can return data through output parameters.

• The procedure can have an integer return code.

The application must be able to handle all of these outputs from stored procedures.

Different OLE DB providers return output parameters and return values at different times during result processing. In case of the Microsoft OLE DB Provider for SQL Server (SQLOLEDB), the output parameters and return codes are not supplied until after the consumer has retrieved or canceled the result sets returned by the stored procedure. The return codes and the output parameters are returned in the last TDS packet from the server.

Providers use the DBPROP\_OUTPUTPARAMETERAVAILABILITY property to report when it returns output parameters and return values. This property is in the DBPROPSET\_DATASOURCEINFO property set.

SQLOLEDB sets the DBPROP\_OUTPUTPARAMETERAVAILABILITY property to DBPROPVAL\_OA\_ATROWRELEASE to indicate that return codes and output parameters are not returned until the result set is processed or released.

**Execute stored procedure using ODBC CALL syntax and process return code and output parameters** 

## Calling a Stored Procedure (OLE DB)

A stored procedure can have zero or more parameters. It can also return a value. In OLE DB, parameters to a stored procedure can be passed by:

- Hard-coding the data value.
- Using a parameter marker (?) to specify parameters, bind a program variable to the parameter marker, and then place the data value in the program variable.

To support parameters, the **ICommandWithParameters** interface is exposed on the command object. To use parameters, the consumer first describes the parameters to the provider by calling the

**ICommandWithParameters::SetParameterInfo** method (or optionally prepares a calling statement that calls the **GetParameterInfo** method). The consumer then creates an accessor that specifies the structure of a buffer and places parameter values in this buffer. Finally, it passes the handle of the accessor and a pointer to the buffer to **Execute**. On later calls to **Execute**, the consumer places new parameter values in the buffer and calls **Execute** with the accessor handle and buffer pointer.

A command that calls a temporary stored procedure using parameters must first call **ICommandWithParameters::SetParameterInfo** to define the parameter information, before the command can be successfully prepared. This is because the internal name for a temporary stored procedure differs from the external name used by a client and SQLOLEDB cannot query the system tables to determine the parameter information for a temporary stored procedure.

These are the steps in the parameter binding process:

1. Fill in the parameter information in an array of DBPARAMBINDINFO structures; that is, parameter name, provider-specific name for the data type of the parameter, or a standard data type name, and so on. Each structure in the array describes one parameter. This array is then passed to the **SetParameterInfo** method.

- Call the ICommandWithParameters::SetParameterInfo method to describe parameters to the provider. SetParameterInfo specifies the native data type of each parameter. SetParameterInfo arguments are:
  - The number of parameters for which to set type information.
  - An array of parameter ordinals for which to set type information.
  - An array of DBPARAMBINDINFO structures.
- 3. Create a parameter accessor by using the **IAccessor::CreateAccessor** command. The accessor specifies the structure of a buffer and places parameter values in the buffer. The **CreateAccessor** command creates an accessor from a set of bindings. These bindings are described by the consumer by using an array of DBBINDING structures. Each binding associates a single parameter to the buffer of the consumer and contains information such as:
  - The ordinal of the parameter to which the binding applies.
  - What is bound (the data value, its length, and its status).
  - The offset in the buffer to each of these parts.
  - The length and type of the data value as it exists in the buffer of the consumer.

An accessor is identified by its handle, which is of type **HACCESSOR**. This handle is returned by the **CreateAccessor** method. Whenever the consumer finishes using an accessor, the consumer must call the **ReleaseAccessor** method to release the memory it holds.

When the consumer calls a method, such as **ICommand::Execute**, it passes the handle to an accessor and a pointer to a buffer itself. The

- provider uses this accessor to determine how to transfer the data contained in the buffer.
- 4. Fill in the DBPARAMS structure. The consumer variables from which input parameter values are taken and to which output parameter values are written are passed at run time to **ICommand::Execute** in the DBPARAMS structure. The DBPARAMS structure includes three elements:
  - A pointer to the buffer from which the provider retrieves input parameter data and to which the provider returns output parameter data, according to the bindings specified by the accessor handle.
  - The number of sets of parameters in the buffer.
  - The accessor handle created in Step 3.
- 5. Execute the command by using **ICommand::Execute**.

#### **Methods of Calling a Stored Procedure**

When executing a stored procedure in SQL Server 2000, Microsoft OLE DB Provider for SQL Server (SQLOLEDB) supports the:

- ODBC CALL escape sequence.
- RPC Escape sequence.
- Transact-SQL EXECUTE statement.

#### **ODBC CALL Escape Sequence**

When the ODBC CALL syntax is used in calling a stored procedure, the provider calls a helper function to find the stored procedure parameter information. Therefore, you do not need to call the

**ICommandWithParameters::SetParameterInfo** method to describe the parameters to the provider.

If you are not sure about the parameter information (parameter meta data), ODBC CALL syntax is recommended.

The general syntax for calling a procedure by using the ODBC CALL escape sequence is:

```
{[?=]call procedure_name[([parameter][,[parameter]]...)]}
```

For example:

{call SalesByCategory('Produce', '1995')}

#### **RPC Escape Sequence**

The PRC escape sequence is similar to the ODBC CALL syntax of calling a stored procedure. The RPC escape sequence provides most optimal performance among the three methods of calling a stored procedure.

When the RPC escape sequence is used to execute a stored procedure, the provider does not call any helper function to determine the parameter information (as it does in the case of ODBC CALL syntax). This improves the performance. In this case, you need to provide the parameter information by executing **ICommandWithParameters::SetParameterInfo**.

The RPC escape sequence requires you to have a return value. If the stored procedure does not return a value, the server returns a 0 by default. In addition, you cannot open a SQL Server cursor on the stored procedure. The stored procedure is prepared implicitly and actual call the

ICommandPrepare::Prepare will fail.

If you know all the parameter meta data, RPC escape sequence is the recommended way to execute stored procedures.

This is an example of RPC escape sequence for calling a stored procedure:

{rpc SalesByCategory}

#### **Transact-SQL EXECUTE Statement**

The ODBC CALL escape sequence and the RPC escape sequence are the preferred methods for calling a stored procedure rather than the Transact-SQL EXECUTE statement. SQLOLEDB uses the remote procedure call (RPC) mechanism of SQL Server 2000 to optimize command processing. This RPC protocol increases performance by eliminating much of the parameter processing and statement parsing done on the server.

This is an example of the Transact-SQL EXECUTE statement:

EXECUTE SalesByCategory 'Produce', '1995'

**Execute stored procedure using ODBC CALL syntax and process return code and output parameters** 

# **Running User-Defined Functions (OLE DB)**

The syntax for calling user-defined functions using the Microsoft OLE DB Provider for SQL Server is similar to calling the stored procedures.

To call a user-defined function and processing the return code

## **Bulk-Copy Rowsets**

SQLOLEDB implements the provider-specific **IRowsetFastLoad** interface to expose support for Microsoft® SQL Server<sup>TM</sup> 2000 bulk copy from a consumer to a SQL Server 2000 table. **IRowsetFastLoad** exposes the two member functions:

#### • InsertRow

Bulk copies a single row of data to a SQL Server 2000 table.

#### • Commit

Marks the end of a batch of bulk copy insertions and writes inserted data to the SQL Server 2000 table.

## Enabling a Session for IRowsetFastLoad

The consumer notifies SQLOLEDB of its need for bulk copy by setting the SQLOLEDB provider-specific data source property SSPROP\_ENABLEFASTLOAD to VARIANT\_TRUE. With the property set on the data source, the consumer creates a SQLOLEDB session. The new session allows consumer access to the **IRowsetFastLoad** interface.

Enabling a session for bulk copy constrains SQLOLEDB support for interfaces on the session. A bulk copy-enabled session exposes only the following interfaces:

- IDBSchemaRowset
- IGetDataSource
- IOpenRowset
- ISupportErrorInfo
- **ITransactionJoin** (not supported for Microsoft® SQL Server™ version 6.5)

To disable the creation of bulk copy-enabled rowsets and cause the SQLOLEDB session to revert to standard processing, reset SSPROP\_ENABLEFASTLOAD to VARIANT\_FALSE.

Fastload sessions are not available with **IDataInitialize** (part of OLE DB service components).

To bulk copy data using IRowsetFastLoad

#### IRowsetFastLoad Rowsets

SQLOLEDB bulk copy rowsets are write-only, but the rowset exposes interfaces that allow the consumer to determine the structure of a Microsoft® SQL Server<sup>TM</sup> 2000 table. The following interfaces are exposed on a bulk copyenabled SQLOLEDB rowset:

- IAccessor
- IColumnsInfo
- IColumnsRowset
- IConvertType
- IRowsetFastLoad
- IRowsetInfo
- ISupportErrorInfo

The provider-specific properties SSPROP\_FASTLOADOPTIONS, SSPROP\_FASTLOADKEEPNULLS, and SSPROP\_FASTLOADKEEPIDENTITY control behaviors of a SQLOLEDB bulk-copy rowset. The properties are specified in the *rgProperties* member of an *rgPropertySets* **IOpenRowset** parameter member.

Property ID	Description
SSPROP_FASTLOADKEEPIDENTITY	Column: No
	R/W: Read/write
	Type: VT_BOOL
	Default: VARIANT_FALSE
	Description: Maintains identity
	values supplied by the consumer.

VARIANT\_FALSE: Values for an identity column in the SQL Server 2000 table are generated by SQL Server 2000. Any value bound for the column is ignored by SQLOLEDB.

VARIANT\_TRUE: The consumer binds an accessor providing a value for a SQL Server 2000 identity column. The identity property is not available on columns accepting NULL, so the consumer provides a unique value on each

IRowsetFastLoad::Insert call.

#### SSPROP FASTLOADKEEPNULLS

Column: No

R/W: Read/write Type: VT\_BOOL

Default: VARIANT\_FALSE
Description: Maintains NULL for
columns with a DEFAULT
constraint. Affects only SQL
Server 2000 columns that accept
NULL and have a DEFAULT
constraint applied.

VARIANT\_FALSE: SQL Server 2000 inserts the default value for the column when the SQLOLEDB consumer inserts a row containing NULL for the column.

VARIANT\_TRUE: SQL Server 2000 inserts NULL for the column value when the SQLOLEDB consumer inserts a

	row containing NULL for the
	column.
SSPROP_FASTLOADOPTIONS	Column: No R/W: Read/write Type: VT_BSTR Default: none Description: This property is the same as the -h "hint[,n]" option of the bcp utility. The following string(s) can be used as option(s) in the bulk copying of data into a table.
	ORDER(column[ASC   DESC] [,n]) Sort order of data in the data file. Bulk copy performance is improved if the data file being loaded is sorted according to the clustered index on the table.
	<b>ROWS_PER_BATCH</b> = <i>bb</i> Number of rows of data per batch (as <i>bb</i> ). The server optimizes the bulk load according to the value <i>bb</i> . By default, <b>ROWS_PER_BATCH</b> is unknown.
	KILOBYTES_PER_BATCH = cc Number of kilobytes (KB) of data per batch (as cc). By default, KILOBYTES_PER_BATCH is unknown.
	TABLOCK A table-level lock is acquired for

the duration of the bulk copy operation. This option significantly improves performance because holding a lock only for the duration of the bulk copy operation reduces lock contention on the table. A table can be loaded by multiple clients concurrently if the table has no indexes and **TABLOCK** is specified. By default, the locking behavior is determined by the table option **table lock on bulk load**.

#### CHECK\_CONSTRAINTS

Any constraints on *table\_name* are checked during the bulk copy operation. By default, constraints are ignored.

# **Updating Data in Rowsets**

SQLOLEDB updates Microsoft® SQL Server<sup>TM</sup> 2000 data when a consumer updates a modifiable rowset containing that data. A modifiable rowset is created when the consumer requests support for either the **IRowsetChange** or **IRowsetUpdate** interface.

All SQLOLEDB modifiable rowsets use SQL Server 2000 cursors to support the rowset. The OLE DB rowset property DBPROP\_LOCKMODE alters SQL Server 2000 concurrency control behavior in cursors and determines the behavior of rowset row fetching and data integrity error generation in updatable rowsets.

SQLOLEDB supports row synchronization before or after an update.

## **Updating Data in SQL Server Cursors**

When fetching and updating data through Microsoft® SQL Server<sup>TM</sup> 2000 cursors, a SQLOLEDB consumer application is bound by the same considerations and constraints that apply to any other client application.

Only rows in SQL Server 2000 cursors participate in concurrent data-access control. When the consumer requests a modifiable rowset, the concurrency control is controlled by DBPROP\_LOCKMODE. To alter the level of concurrent access control, the consumer sets the DBPROP\_LOCKMODE property prior to opening the rowset.

Transaction isolation levels can cause significant lags in row positioning if client application design allows transactions to remain open for long periods of time. By default, SQLOLEDB uses the read-committed isolation level specified by DBPROPVAL\_TI\_READCOMMITTED. SQLOLEDB supports dirty read isolation when the rowset concurrency is read-only. Therefore, the consumer can request a higher level of isolation in a modifiable rowset but cannot request any lower level successfully.

#### **Immediate and Delayed Update Modes**

In immediate update mode, each call to **IRowsetChange::SetData** results in a round trip to the SQL Server 2000. If the consumer makes multiple changes to a single row, it is more efficient to submit all changes with a single **SetData** call.

In delayed update mode, a roundtrip is made to the SQL Server 2000 for each row indicated in the *cRows* and *rghRows* parameters of **IRowsetUpdate**::**Update**.

In either mode, a round trip represents a distinct transaction when no transaction object is open for the rowset.

When using **IRowsetUpdate::Update**, SQLOLEDB attempts to process each indicated row. An error occurring due to invalid data, length, or status values for any row does not stop SQLOLEDB processing. All or none of the other rows participating in the update may be modified. The consumer must check the returned *prgRowStatus* array to determine failure for any specific row when

#### SQLOLEDB returns DB\_S\_ERRORSOCCURED.

A consumer should not assume that rows are processed in any specific order. If a consumer requires ordered processing of data modification over more than a single row, then the consumer should establish that order in the application logic and open a transaction to enclose the process.

# **Resynchronizing Rows**

SQLOLEDB supports **IRowsetResynch** on Microsoft® SQL Server<sup>TM</sup> 2000 cursor-supported rowsets only. **IRowsetResynch** is not available on demand. The consumer must request the interface prior to opening the rowset.

### **BLOBs and OLE Objects**

SQLOLEDB exposes the **ISequentialStream** interface to support consumer access to Microsoft® SQL Server<sup>™</sup> 2000 **ntext**, **text**, and **image** data types as binary large objects (BLOBs). The **Read** method on **ISequentialStream** allows the consumer to retrieve large amounts of data in manageable chunks.

SQLOLEDB can use a consumer-implemented **IStorage** interface when the consumer provides the interface pointer in an accessor bound for data modification.

#### **SQLOLEDB Storage Object Limitations**

- SQLOLEDB can support only a single open storage object. Attempts to open more than one storage object (attempts to get a reference on more than one **ISequentialStream** interface pointer) return DBSTATUS\_E\_CANTCREATE.
- In SQLOLEDB, the default value of the DBPROP\_BLOCKINGSTORAGEOBJECTS read-only property is VARIANT\_TRUE. This indicates that if a storage object is active, some methods (other than those on the storage objects) will fail with E\_UNEXPECTED.
- The length of data presented by a consumer-implemented storage object must be made known to SQLOLEDB when the row accessor that references the storage object is created. The consumer must bind a length indicator in the DBBINDING structure used for accessor creation.
- SQLOLEDB supports the **ISequentialStream::Write** method for zero-length strings and NULL values only. Attempts to write more than zero bytes through **ISequentialStream::Write** fail.

• If a row contains more than a single large data value, and DBPROP\_ACCESSORDER is not DBPROPVAL\_AO\_RANDOM, the consumer must either use a SQLOLEDB cursor-supported rowset to retrieve row data or process all large data values prior to retrieving other row values. If DBPROP\_ACCESSORDER is DBPROPVAL\_AO\_RANDOM, SQLOLEDB caches all the BLOB data so it can be accessed in any order.

## **Getting Large Data**

In general, consumers should isolate code that creates a SQLOLEDB storage object from other code that handles data not referenced through an **ISequentialStream** interface pointer.

If the DBPROP\_ACCESSORDER property (in the rowset property group) is set to either of the values DBPROPVAL\_AO\_SEQUENTIAL or DBPROPVAL\_AO\_SEQUENTIALSTORAGEOBJECTS, the consumer should fetch only a single row of data in a call to the **GetNextRows** method because BLOB data is not buffered. If the value of DBPROP\_ACCESSORDER is set to DBPROPVAL\_AO\_RANDOM, the consumer can fetch multiple rows of data in **GetNextRows**.

SQLOLEDB does not retrieve large data from Microsoft® SQL Server™ until requested to do so by the consumer. The consumer should bind all short data in one accessor, and then use one or more temporary accessors to retrieve large data values as required.

This example retrieves a large data value from a single column:

```
HRESULT GetUnboundData

(
IRowset* pIRowset,
HROW hRow,
ULONG nCol,
BYTE* pUnboundData
)
{
UINT cbRow = sizeof(IUnknown*) + sizeof(ULONG);
BYTE* pRow = new BYTE[cbRow];

DBOBJECT dbobject;

IAccessor* pIAccessor = NULL;
HACCESSOR haccessor;
```

```
dbbinding;
DBBINDING
               ulbindstatus;
ULONG
               dwStatus:
ULONG
ISequentialStream* pISequentialStream;
ULONG
               cbRead:
HRESULT
                hr;
// Set up the DBOBJECT structure.
dbobject.dwFlags = STGM_READ;
dbobject.iid = IID_ISequentialStream;
// Create the DBBINDING, requesting a storage-object pointer from
// SQLOLEDB.
dbbinding.iOrdinal = nCol;
dbbinding.obValue = 0;
dbbinding.obStatus = sizeof(IUnknown*);
dbbinding.obLength = 0;
dbbinding.pTypeInfo = NULL;
dbbinding.pObject = &dbobject;
dbbinding.pBindExt = NULL;
dbbinding.dwPart = DBPART_VALUE | DBPART_STATUS;
dbbinding.dwMemOwner = DBMEMOWNER_CLIENTOWNED;
dbbinding.eParamIO = DBPARAMIO_NOTPARAM;
dbbinding.cbMaxLen = 0;
dbbinding.dwFlags = 0;
dbbinding.wType = DBTYPE_IUNKNOWN;
dbbinding.bPrecision = 0;
dbbinding.bScale = 0;
if (FAILED(hr = pIRowset->
```

```
QueryInterface(IID_IAccessor, (void**) &pIAccessor)))
  // Process QueryInterface failure.
  return (hr);
// Create the accessor.
if (FAILED(hr = pIAccessor->CreateAccessor(DBACCESSOR_RO
  &dbbinding, 0, &haccessor, &ulbindstatus)))
  // Process error from CreateAccessor.
  pIAccessor->Release();
  return (hr);
  }
// Read and process BLOCK_SIZE bytes at a time.
if (SUCCEEDED(hr = pIRowset->GetData(hRow, haccessor, pRow
  dwStatus = *((ULONG*) (pRow + dbbinding.obStatus));
  if (dwStatus == DBSTATUS_S_ISNULL)
    // Process NULL data
  else if (dwStatus == DBSTATUS_S_OK)
    pISequentialStream = *((ISequentialStream**)
       (pRow + dbbinding.obValue));
    do
      if (SUCCEEDED(hr =
         pISequentialStream->Read(pUnboundData,
```

```
BLOCK_SIZE, &cbRead)))
         pUnboundData += cbRead;
      }
    while (SUCCEEDED(hr) && cbRead >= BLOCK_SIZE);
    pISequentialStream->Release();
else
  // Process error from GetData.
pIAccessor->ReleaseAccessor(haccessor, NULL);
pIAccessor->Release();
delete [] pRow;
return (hr);
```

## **Setting Large Data**

With the SQLOLEDB provider, you can set BLOB data by passing a pointer to a consumer storage object.

The consumer creates a storage object containing the data and passes a pointer to this storage object to the provider. The provider then reads data from the consumer storage object and writes it to the BLOB column.

To pass a pointer to its own storage object, the consumer creates an accessor that binds the value of the BLOB column. The consumer then calls the **IRowsetChange::SetData** or **IRowsetChange::InsertRow** method with the accessor that binds the BLOB column. It passes a pointer to a storage interface on the storage object of the consumer.

To set large data

#### **Tables and Indexes**

SQLOLEDB exposes the **IIndexDefinition** and **ITableDefinition** interfaces, allowing consumers to create, alter, and drop Microsoft® SQL Server<sup>TM</sup> 2000 tables and indexes. Valid table and index definitions depend on the version of SQL Server.

The ability to create or drop tables and indexes depends on the SQL Server 2000 access rights of the consumer-application user. Dropping a table can be further constrained by the presence of declarative referential integrity constraints or other factors.

Most applications targeting SQL Server 2000 use SQL-DMO instead of these OLE DB interfaces. SQL-DMO is a collection of OLE Automation objects that support all the administrative functions of SQL Server 2000. Applications targeting multiple OLE DB providers use these generic OLE DB interfaces that are supported by the various OLE DB providers.

In the provider-specific property set DBPROPSET\_SQLSERVERCOLUMN, SQL Server defines the following property.

Property ID	Description
SSPROP_COL_COLLATIONNAME	Type: VT_BSTR
	R/W:W
	Default: Null
	Description: This property is used
	only in <b>ITableDefinition</b> . The string
	specified in this property is used
	when creating a CREATE TABLE
	statement.

#### See Also

CREATE TABLE
DROP TABLE

# CREATE INDEX DROP INDEX

#### **Creating SQL Server Tables**

SQLOLEDB exposes the **ITableDefinition::CreateTable** function, allowing consumers to create Microsoft® SQL Server<sup>TM</sup> 2000 tables. Consumers use **CreateTable** to create consumer-named permanent tables, and permanent or temporary tables with unique names generated by SQLOLEDB.

When the consumer calls **ITableDefinition::CreateTable**, if the value of the DBPROP\_TBL\_TEMPTABLE property is VARIANT\_TRUE, SQLOLEDB generates a temporary table name for the consumer. The consumer sets the *pTableID* parameter of the **CreateTable** method to NULL. The temporary tables with names generated by SQLOLEDB do not appear in the **TABLES** rowset, but are accessible through the **IOpenRowset** interface.

When consumers specify the table name in the *pwszName* member of the *uName* union in the *pTableID* parameter, SQLOLEDB creates a SQL Server 2000 table with that name. SQL Server 2000 table naming constraints apply, and the table name can indicate a permanent table, or either a local or global temporary table. For more information, see <u>CREATE TABLE</u>. The *ppTableID* parameter can be NULL.

SQLOLEDB can generate the names of permanent or temporary tables. When the consumer sets the *pTableID* parameter to NULL and sets *ppTableID* to point to a valid DBID\*, SQLOLEDB returns the generated name of the table in the *pwszName* member of the *uName* union of the DBID pointed to by the value of *ppTableID*. To create a temporary, SQLOLEDB-named table, the consumer includes the OLE DB table property DBPROP\_TBL\_TEMPTABLE in a table property set referenced in the *rgPropertySets* parameter. SQLOLEDB-named temporary tables are local.

**CreateTable** returns DB\_E\_BADTABLEID if the *eKind* member of the *pTableID* parameter does not indicate DBKIND\_NAME.

#### **DBCOLUMNDESC Usage**

The consumer can indicate a column data type by using either the *pwszTypeName* member or the *wType* member. If the consumer specifies the data type in *pwszTypeName*, SQLOLEDB ignores the value of *wType*.

If using the *pwszTypeName* member, the consumer specifies the data type by using SQL Server data type names. Valid data type names are those returned in the TYPE\_NAME column of the PROVIDER\_TYPES schema rowset.

SQLOLEDB recognizes a subset of OLE DB-enumerated DBTYPE values in the *wType* member. For more information, see <u>Data Type Mapping in</u> ITableDefinition.

**CreateTable** returns DB\_E\_BADTYPE if consumer sets either the *pTypeInfo* or *pclsid* member to specify the column data type.

The consumer specifies the column name in the *pwszName* member of the *uName* union of the DBCOLUMNDESC *dbcid* member. The column name is specified as a Unicode character string. The *eKind* member of *dbcid* must be DBKIND\_NAME. **CreateTable** returns DB\_E\_BADCOLUMNID if *eKind* is invalid, *pwszName* is NULL, or if the value of *pwszName* is not a valid SQL Server 2000 identifier.

All column properties are available on all columns defined for the table. **CreateTable** can return DB\_S\_ERRORSOCCURRED or

DB\_E\_ERRORSOCCURRED if property values are set in conflict. **CreateTable** returns an error when invalid column property settings cause SQL Server table-creation failure.

Column properties in a DBCOLUMNDESC are interpreted as follows.

Property ID	Description
DBPROP_COL_AUTOINCREMENT	R/W: Read/write
	Default: VARIANT_FALSE
	Description: Sets the identity property
	on the column created. For SQL
	Server 2000, the identity property is
	valid for a single column within a
	table. Setting the property to
	VARIANT_TRUE for more than a
	single column generates an error when
	SQLOLEDB attempts to create the
	table on the server.
	The SQL Server 2000 identity

property is only valid for the **integer**, **numeric**, and **decimal** types when the scale is 0. Setting the property to VARIANT\_TRUE on a column of any other data type generates an error when SQLOLEDB attempts to create the table on the server.

SQLOLEDB returns DB S ERRORSOCCURRED when DBPROP COL AUTOINCREMENT and DBPROP\_COL\_NULLABLE are both VARIANT\_TRUE and the dwOption of DBPROP COL NULLABLE is not DBPROPOPTIONS REQUIRED. DB E ERRORSOCCURRED is returned when DBPROP COL AUTOINCREMENT and DBPROP\_COL\_NULLABLE are both VARIANT TRUE and the dwOption of DBPROP COL NULLABLE equals DBPROPOPTIONS REQUIRED. The column is defined with the SQL Server identity property and the DBPROP COL NULLABLE dwStatus member is set to DBPROPSTATUS CONFLICTING.

DBPROP COL DEFAULT

R/W: Read/write

Default: None

Description: Creates a SQL Server DEFAULT constraint for the column.

The *vValue* DBPROP member can be any of a number of types. The *vValue.vt* member should specify a

	type compatible with the data type of the column. For example, defining BSTR N/A as the default value for a column defined as DBTYPE_WSTR is a compatible match. Defining the same default on a column defined as DBTYPE_R8 generates an error when SQLOLEDB attempts to create the table on the server.
DBPROP_COL_DESCRIPTION	R/W: Read/write Default: None Description: The DBPROP_COL_DESCRIPTION column property is not implemented by SQLOLEDB.
	The <i>dwStatus</i> member of the DBPROP structure returns DBPROPSTATUS_NOTSUPPORTED when the consumer attempts to write the property value.
	Setting the property does not constitute a fatal error for SQLOLEDB. If all other parameter values are valid, the SQL Server table is created.
DBPROP_COL_FIXEDLENGTH	R/W: Read/write Default: VARIANT_FALSE Description: SQLOLEDB uses DBPROP_COL_FIXEDLENGTH to determine data type-mapping when the consumer defines a column's data type by using the <i>wType</i> member of the DBCOLUMNDESC. For more information, see <u>Data Type Mapping</u> in ITableDefinition.
DBPROP_COL_NULLABLE	R/W: Read/write

Default: None

Description: When creating the table, SQLOLEDB indicates whether the column should accept null values if the property is set. When the property is not set, the ability of the column to accept NULL as a value is determined by the SQL Server ANSI\_NULLS default database option.

SQLOLEDB is an SQL-92 compliant provider. Connected sessions exhibit SQL-92 behaviors. If the consumer does not set DBPROP\_COL\_NULLABLE, columns accept null values.

#### DBPROP\_COL\_PRIMARYKEY

R/W: Read/write

Default: VARIANT\_FALSE
Description: When VARIANT\_TRUE,
SQLOLEDB creates the column with a
PRIMARY KEY constraint.

When defined as a column property, only a single column can determine the constraint. Setting the property VARIANT\_TRUE for more than a single column returns an error when SQLOLEDB attempts to create the SQL Server 2000 table.

Note: The consumer can use **IIndexDefinition::CreateIndex** to create a PRIMARY KEY constraint on two or more columns.

SQLOLEDB returns
DB\_S\_ERRORSOCCURRED when
DBPROP\_COL\_PRIMARYKEY and

DBPROP\_COL\_UNIQUE are both VARIANT\_TRUE and the *dwOption* of DBPROP\_COL\_UNIQUE is not DBPROPOPTIONS\_REQUIRED.

DB\_E\_ERRORSOCCURRED is returned when DBPROP\_COL\_PRIMARYKEY and DBPROP\_COL\_UNIQUE are both VARIANT\_TRUE and the *dwOption* of DBPROP\_COL\_UNIQUE equals DBPROPOPTIONS\_REQUIRED. The column is defined with the SQL Server identity property and the DBPROP\_COL\_PRIMARYKEY *dwStatus* member is set to DBPROPSTATUS CONFLICTING.

SQLOLEDB returns an error when DBPROP\_COL\_PRIMARYKEY and DBPROP\_COL\_NULLABLE are both VARIANT\_TRUE.

SQLOLEDB returns an error from SQL Server when the consumer attempts to create a PRIMARY KEY constraint on a column of invalid SQL Server data type. PRIMARY KEY constraints cannot be defined on columns created with the SQL Server data types **bit**, **text**, **ntext**, and **image**.

DBPROP\_COL\_UNIQUE

R/W: Read/write

Default: VARIANT\_FALSE
Description: Applies a SQL Server
UNIQUE constraint to the column.

When defined as a column property, the constraint is applied on a single

column only. The consumer can use **IIndexDefinition::CreateIndex** to apply a UNIQUE constraint on the combined values of two or more columns.

SQLOLEDB returns
DB\_S\_ERRORSOCCURRED when
DBPROP\_COL\_PRIMARYKEY and
DBPROP\_COL\_UNIQUE are both
VARIANT\_TRUE and *dwOption* is
not DBPROPOPTIONS\_REQUIRED.

DB\_E\_ERRORSOCCURRED is returned when DBPROP\_COL\_PRIMARYKEY and DBPROP\_COL\_UNIQUE are both VARIANT\_TRUE and *dwOption* equals DBPROPOPTIONS\_REQUIRED. The column is defined with the SQL Server identity property and the DBPROP\_COL\_PRIMARYKEY

dwStatus member is set to

SQLOLEDB returns
DB\_S\_ERRORSOCCURRED when
DBPROP\_COL\_NULLABLE and
DBPROP\_COL\_UNIQUE are both
VARIANT\_TRUE and dwOption is
not DBPROPOPTIONS\_REQUIRED.

DBPROPSTATUS\_CONFLICTING.

DB\_E\_ERRORSOCCURRED is returned when DBPROP\_COL\_NULLABLE and DBPROP\_COL\_UNIQUE are both VARIANT\_TRUE and *dwOption* equals

DBPROPOPTIONS\_REQUIRED.
The column is defined with the SQL
Server identity property and the
DBPROP\_COL\_NULLABLE
dwStatus member is set to
DBPROPSTATUS\_CONFLICTING.

SQLOLEDB returns an error from
SQL Server 2000 when the consumer
attempts to create a UNIQUE
constraint on a column of invalid SQL
Server 2000 data type. UNIQUE
constraints cannot be defined on
columns created with the SQL Server
2000 bit data type.

When the consumer calls **ITableDefinition::CreateTable**, SQLOLEDB interprets table properties as follows.

Property ID	Description
DBPROP_TBL_TEMPTABLE	R/W: Read/write
	Default: VARIANT_FALSE
	Description: By default, SQLOLEDB
	creates tables named by the consumer.
	When VARIANT_TRUE, SQLOLEDB
	generates a temporary table name for the
	consumer. The consumer sets the <i>pTableID</i>
	parameter of <b>CreateTable</b> to NULL. The
	ppTableID parameter must contain a valid
	pointer.

If the consumer requests that a rowset be opened on a successfully created table, SQLOLEDB opens a cursor-supported rowset. Any rowset properties can be indicated in the property sets passed.

```
This example creates a SQL Server 2000 table.
// This CREATE TABLE statement shows the details of the table create
// the following example code.
//
// CREATE TABLE OrderDetails
// (
//
   OrderID
              int
                    NOT NULL
   ProductID int
                    NOT NULL
   CONSTRAINT PK OrderDetails
      PRIMARY KEY CLUSTERED (OrderID, ProductID),
//
                      NOT NULL,
   UnitPrice money
   Quantity int
                  NOT NULL.
//
   Discount decimal(2,2) NOT NULL
     DEFAULT 0
//
//)
//
// The PRIMARY KEY constraint is created in an additional example.
HRESULT CreateTable
  ITableDefinition* pITableDefinition
  {
              dbidTable;
  DBID
  const ULONG
                  nCols = 5;
  ULONG
                nCol;
  ULONG
                nProp;
  DBCOLUMNDESC dbcoldesc[nCols];
  HRESULT
                 hr;
  // Set up column descriptions. First, set default property values for
  // the columns.
  for (nCol = 0; nCol < nCols; nCol++)
```

```
dbcoldesc[nCol].pwszTypeName = NULL;
dbcoldesc[nCol].pTypeInfo = NULL;
dbcoldesc[nCol].rgPropertySets = new DBPROPSET;
dbcoldesc[nCol].pclsid = NULL;
dbcoldesc[nCol].cPropertySets = 1;
dbcoldesc[nCol].ulColumnSize = 0;
dbcoldesc[nCol].dbcid.eKind = DBKIND_NAME;
dbcoldesc[nCol].wType = DBTYPE I4;
dbcoldesc[nCol].bPrecision = 0;
dbcoldesc[nCol].bScale = 0;
dbcoldesc[nCol].rgPropertySets[0].rgProperties =
  new DBPROP[NCOLPROPS MAX];
dbcoldesc[nCol].rgPropertySets[0].cProperties = NCOLPROPS_I
dbcoldesc[nCol].rgPropertySets[0].guidPropertySet =
  DBPROPSET_COLUMN;
for (nProp = 0; nProp < NCOLPROPS_MAX; nProp++)
  {
  dbcoldesc[nCol].rgPropertySets[0].rgProperties[nProp].
    dwOptions = DBPROPOPTIONS REQUIRED;
  dbcoldesc[nCol].rgPropertySets[0].rgProperties[nProp].colid
     = DB NULLID;
  VariantInit(
    &(dbcoldesc[nCol].rgPropertySets[0].rgProperties[nProp].
      vValue));
  dbcoldesc[nCol].rgPropertySets[0].rgProperties[nProp].
    vValue.vt = VT BOOL;
  }
}
```

```
// Set the column-specific information.
dbcoldesc[0].dbcid.uName.pwszName = L"OrderID";
dbcoldesc[0].rgPropertySets[0].rgProperties[0].dwPropertyID =
  DBPROP COL NULLABLE;
dbcoldesc[0].rgPropertySets[0].rgProperties[0].vValue.boolVal =
  VARIANT FALSE;
dbcoldesc[0].rgPropertySets[0].cProperties = 1;
dbcoldesc[1].dbcid.uName.pwszName = L"ProductID";
dbcoldesc[1].rgPropertySets[0].rgProperties[0].dwPropertyID =
  DBPROP COL NULLABLE;
dbcoldesc[1].rgPropertySets[0].rgProperties[0].vValue.boolVal =
  VARIANT FALSE;
dbcoldesc[1].rgPropertySets[0].cProperties = 1;
dbcoldesc[2].dbcid.uName.pwszName = L"UnitPrice";
dbcoldesc[2].wType = DBTYPE_CY;
dbcoldesc[2].rgPropertySets[0].rgProperties[0].dwPropertyID =
  DBPROP COL NULLABLE;
dbcoldesc[2].rgPropertySets[0].rgProperties[0].vValue.boolVal =
  VARIANT FALSE;
dbcoldesc[2].rgPropertySets[0].cProperties = 1;
dbcoldesc[3].dbcid.uName.pwszName = L"Quantity";
dbcoldesc[3].rgPropertySets[0].rgProperties[0].dwPropertyID =
  DBPROP_COL_NULLABLE;
dbcoldesc[3].rgPropertySets[0].rgProperties[0].vValue.boolVal =
  VARIANT FALSE;
dbcoldesc[3].rgPropertySets[0].cProperties = 1;
dbcoldesc[4].dbcid.uName.pwszName = L"Discount";
dbcoldesc[4].wType = DBTYPE NUMERIC;
```

```
dbcoldesc[4].bPrecision = 2;
  dbcoldesc[4].bScale = 2;
  dbcoldesc[4].rgPropertySets[0].rgProperties[0].dwPropertyID =
    DBPROP COL NULLABLE;
  dbcoldesc[4].rgPropertySets[0].rgProperties[0].vValue.boolVal =
    VARIANT FALSE;
  dbcoldesc[4].rgPropertySets[0].rgProperties[1].dwPropertyID =
    DBPROP_COL_DEFAULT;
  dbcoldesc[4].rgPropertySets[0].rgProperties[1].vValue.vt = VT_BS]
  dbcoldesc[4].rgPropertySets[0].rgProperties[1].vValue.bstrVal =
    SysAllocString(L"0");
  dbcoldesc[4].rgPropertySets[0].cProperties = 2;
  // Set up the dbid for OrderDetails.
  dbidTable.eKind = DBKIND NAME;
  dbidTable.uName.pwszName = L"OrderDetails";
  if (FAILED(hr = pITableDefinition->CreateTable(NULL, &dbidTab
    nCols, dbcoldesc, NULL, 0, NULL, NULL, NULL)))
    {
    DumpError(pITableDefinition, IID_ITableDefinition);
    goto SAFE EXIT;
SAFE EXIT:
  // Clean up dynamic allocation in the property sets.
  for (nCol = 0; nCol < nCols; nCol++)
    for (nProp = 0; nProp < NCOLPROPS_MAX; nProp++)
      if (dbcoldesc[nCol].rgPropertySets[0].rgProperties[nProp].
         vValue.vt == VT BSTR)
```

#### Adding a Column to a SQL Server Table

SQLOLEDB exposes the **ITableDefinition::AddColumn** function, allowing consumers to add a column to a Microsoft® SQL Server<sup>TM</sup> 2000 table.

When adding a column to a SQL Server 2000 table, the SQLOLEDB consumer is constrained as follows:

- If DBPROP\_COL\_AUTOINCREMENT is VARIANT\_TRUE, DBPROP\_COL\_NULLABLE must be VARIANT\_FALSE.
- If the column is defined with the SQL Server 2000 **timestamp** data type, DBPROP\_COL\_NULLABLE must be VARIANT\_FALSE.
- For any other column definition, DBPROP\_COL\_NULLABLE must be VARIANT\_TRUE.

Consumers specify the table name as a Unicode character string in the *pwszName* member of the *uName* union in the *pTableID* parameter. The *eKind* member of *pTableID* must be DBKIND\_NAME.

The new column name is specified as a Unicode character string in the *pwszName* member of the *uName* union in the *dbcid* member of the DBCOLUMNDESC parameter *pColumnDesc*. The *dbcid eKind* member must be DBKIND\_NAME.

#### See Also

<u>ALTER TABLE</u>

#### Removing a Column from a SQL Server Table

SQLOLEDB exposes the **ITableDefinition::DropColumn** function, allowing consumers to remove a column from a Microsoft® SQL Server<sup>TM</sup> 2000 table.

Consumers specify the table name as a Unicode character string in the *pwszName* member of the *uName* union in the *pTableID* parameter. The *eKind* member of *pTableID* must be DBKIND\_NAME.

The consumer indicates a column name in the *pwszName* member of the *uName* union in the *pColumnID* parameter. The column name is a Unicode character string. The *eKind* member of *pColumnID* must be DBKIND\_NAME.

**Note** Removing a column is not supported for a consumer connected to a server running SQL Server version 6.5. **ITableDefinition::DropColumn** returns E\_NOTIMPL when the consumer application attempts to remove a column.

# **Dropping a SQL Server Table**

SQLOLEDB exposes the **ITableDefinition::DropTable** function, allowing consumers to remove a Microsoft® SQL Server™ 2000 table from a database.

Consumers specify the table name as a Unicode character string in the *pwszName* member of the *uName* union in the *pTableID* parameter. The *eKind* member of *pTableID* must be DBKIND\_NAME.

#### **Creating SQL Server Indexes**

SQLOLEDB exposes the **IIndexDefinition::CreateIndex** function, allowing consumers to define new indexes on Microsoft® SQL Server<sup>TM</sup> 2000 tables.

SQLOLEDB creates table indexes as either indexes or constraints. SQL Server 2000 gives constraint-creation privilege to the table owner, database owner, and members of certain administrative roles. By default, only the table owner can create an index on a table. Therefore, **CreateIndex** success or failure depends not only on the application user's access rights but also on the type of index created.

Consumers specify the table name as a Unicode character string in the *pwszName* member of the *uName* union in the *pTableID* parameter. The *eKind* member of *pTableID* must be DBKIND\_NAME.

The *pIndexID* parameter can be NULL, and if it is, SQLOLEDB creates a unique name for the index. The consumer can capture the name of the index by specifying a valid pointer to a DBID in the *ppIndexID* parameter.

The consumer can specify the index name as a Unicode character string in the *pwszName* member of the *uName* union of the *pIndexID* parameter. The *eKind* member of *pIndexID* must be DBKIND\_NAME.

The consumer specifies the column or columns participating in the index by name. For each DBINDEXCOLUMNDESC structure used in **CreateIndex**, the *eKind* member of the *pColumnID* must be DBKIND\_NAME. The name of the column is specified as a Unicode character string in the *pwszName* member of the *uName* union in the *pColumnID*.

SQLOLEDB and SQL Server 2000 support ascending order on values in the index. SQLOLEDB returns E\_INVALIDARG if the consumer specifies DBINDEX\_COL\_ORDER\_DESC in any DBINDEXCOLUMNDESC structure.

**CreateIndex** interprets index properties as follows.

Property ID	Description
DBPROP_INDEX_AUTOUPDATE	R/W: Read/write
	Default: None

	Description: SQLOLEDB does not support this property. Attempts to set the property in <b>CreateIndex</b> cause a DB_S_ERRORSOCCURED return value. The <i>dwStatus</i> member of the property structure indicates DBPROPSTATUS_BADVALUE.
DBPROP_INDEX_CLUSTERED	R/W: Read/write Default: VARIANT_FALSE Description: Controls index clustering.
	VARIANT_TRUE: SQLOLEDB attempts to create a clustered index on the SQL Server 2000 table. SQL Server 2000 supports at most one clustered index on any table.
	VARIANT_FALSE: SQLOLEDB attempts to create a nonclustered index on the SQL Server 2000 table.
DBPROP_INDEX_FILLFACTOR	R/W: Read/write Default: 0 Description: Specifies the percentage of an index page used for storage. For more information, see <u>CREATE</u> <u>INDEX</u> .
	The type of the variant is VT_I4. The value must be greater than or equal to 1 and less than or equal to 100.

DBPROP_INDEX_INITIALIZE	R/W: Read/write Default: None Description: SQLOLEDB does not support this property. Attempts to set the property in CreateIndex cause a DB_S_ERRORSOCCURED return value. The dwStatus member of the property structure indicates DBPROPSTATUS_BADVALUE.
DBPROP_INDEX_NULLCOLLATION	R/W: Read/write Default: None Description: SQLOLEDB does not support this property. Attempts to set the property in CreateIndex cause a DB_S_ERRORSOCCURED return value. The dwStatus member of the property structure indicates DBPROPSTATUS_BADVALUE.
DBPROP_INDEX_NULLS	R/W: Read/write Default: None Description: SQLOLEDB does not support this property. Attempts to set the property in CreateIndex cause a DB_S_ERRORSOCCURED return value. The dwStatus member of the property structure indicates DBPROPSTATUS_BADVALUE.
DBPROP_INDEX_PRIMARYKEY	R/W: Read/write Default: VARIANT_FALSE Description: Creates the index as a referential integrity, PRIMARY

KEY constraint.  VARIANT_TRUE: The index is created to support the PRIMARY KEY constraint of the table. The columns must be nonnullable.  VARIANT_FALSE: The index is not used as a PRIMARY KEY constraint for row values in the table.
R/W: Read/write Default: None Description: SQLOLEDB does not support this property. Attempts to set the property in CreateIndex cause a DB_S_ERRORSOCCURED return value. The dwStatus member of the property structure indicates DBPROPSTATUS_BADVALUE.
R/W: Read/write Default: None Description: SQLOLEDB does not support this property. Attempts to set the property in CreateIndex cause a DB_S_ERRORSOCCURED return value. The dwStatus member of the property structure indicates DBPROPSTATUS_BADVALUE.
R/W: Read/write Default: None Description: SQLOLEDB does not support this property.

	Attempts to set the property in <b>CreateIndex</b> cause a DB_S_ERRORSOCCURED return value. The <i>dwStatus</i> member of the property structure indicates DBPROPSTATUS_BADVALUE.
DBPROP_INDEX_UNIQUE	R/W: Read/write Default: VARIANT_FALSE Description: Creates the index as a UNIQUE constraint on the participating column or columns.
	VARIANT_TRUE: The index is used to uniquely constrain row values in the table.
	VARIANT_FALSE: The index does not uniquely constrain row values.

This example creates a primary key index:

```
// This CREATE TABLE statement shows the referential integrity and
// PRIMARY KEY constraint on the OrderDetails table that will be cre
// by the following example code.
// CREATE TABLE OrderDetails
// (
   OrderID
              int
                   NOT NULL
   ProductID int
                   NOT NULL
     CONSTRAINT PK_OrderDetails
//
     PRIMARY KEY CLUSTERED (OrderID, ProductID),
//
   UnitPrice money
                      NOT NULL,
   Quantity int
                  NOT NULL,
```

```
Discount decimal(2,2) NOT NULL
     DEFAULT 0
//
//)
//
HRESULT CreatePrimaryKey
  IIndexDefinition* pIIndexDefinition
  HRESULT
                  hr = S OK;
  DBID
                dbidTable;
                dbidIndex;
  DBID
  const ULONG
                   nCols = 2;
  ULONG
                 nCol;
  const ULONG
                   nProps = 2;
  ULONG
                 nProp;
  DBINDEXCOLUMNDESC dbidxcoldesc[nCols];
  DBPROP
                  dbpropIndex[nProps];
                    dbpropset;
  DBPROPSET
                pdbidIndexOut = NULL;
  DBID*
  // Set up identifiers for the table and index.
  dbidTable.eKind = DBKIND NAME;
  dbidTable.uName.pwszName = L"OrderDetails";
  dbidIndex.eKind = DBKIND NAME;
  dbidIndex.uName.pwszName = L"PK_OrderDetails";
  // Set up column identifiers.
  for (nCol = 0; nCol < nCols; nCol++)
```

```
dbidxcoldesc[nCol].pColumnID = new DBID;
  dbidxcoldesc[nCol].pColumnID->eKind = DBKIND_NAME;
  dbidxcoldesc[nCol].eIndexColOrder = DBINDEX COL ORDEF
dbidxcoldesc[0].pColumnID->uName.pwszName = L"OrderID";
dbidxcoldesc[1].pColumnID->uName.pwszName = L"ProductID";
// Set properties for the index. The index is clustered,
// PRIMARY KEY.
for (nProp = 0; nProp < nProps; nProp++)
  dbpropIndex[nProp].dwOptions = DBPROPOPTIONS_REQUIR
  dbpropIndex[nProp].colid = DB_NULLID;
  VariantInit(&(dbpropIndex[nProp].vValue));
  dbpropIndex[nProp].vValue.vt = VT_BOOL;
dbpropIndex[0].dwPropertyID = DBPROP_INDEX_CLUSTERED;
dbpropIndex[0].vValue.boolVal = VARIANT TRUE;
dbpropIndex[1].dwPropertyID = DBPROP_INDEX_PRIMARYKE
dbpropIndex[1].vValue.boolVal = VARIANT_TRUE;
dbpropset.rgProperties = dbpropIndex;
dbpropset.cProperties = nProps;
dbpropset.guidPropertySet = DBPROPSET_INDEX;
hr = pIIndexDefinition->CreateIndex(&dbidTable, &dbidIndex, nCc
  dbidxcoldesc, 1, &dbpropset, &pdbidIndexOut);
```

```
// Clean up dynamically allocated DBIDs.
for (nCol = 0; nCol < nCols; nCol++)
    {
    delete dbidxcoldesc[nCol].pColumnID;
    }
return (hr);
}</pre>
```

#### **Dropping a SQL Server Index**

SQLOLEDB exposes the **IIndexDefinition::DropIndex** function, allowing consumers to remove an index from a Microsoft® SQL Server<sup>TM</sup> 2000 table.

SQLOLEDB exposes some SQL Server 2000 PRIMARY KEY and UNIQUE constraints as indexes. The table owner, database owner, and some administrative role members can alter a SQL Server 2000 table, dropping a constraint. By default, only the table owner can drop an existing index. Therefore, **DropIndex** success or failure depends not only on the application user's access rights but also on the type of index indicated.

Consumers specify the table name as a Unicode character string in the *pwszName* member of the *uName* union in the *pTableID* parameter. The *eKind* member of *pTableID* must be DBKIND\_NAME.

Consumers specify the index name as a Unicode character string in the *pwszName* member of the *uName* union in the *pIndexID* parameter. The *eKind* member of *pIndexID* must be DBKIND\_NAME. SQLOLEDB does not support the OLE DB feature of dropping all indexes on a table when *pIndexID* is null. If *pIndexID* is null, E\_INVALIDARG is returned.

#### See Also

ALTER TABLE

**DROP INDEX** 

#### **Notifications**

SQLOLEDB supports consumer notification on rowset modification. The consumer receives notification at every phase of rowset modification and on any attempted change.

To receive notification, the consumer queries the rowset for a connection-point interface, then connects a consumer-implemented **IRowsetNotify** interface to the rowset.

The consumer can cancel a rowset-modification attempt on receiving notification from SQLOLEDB. Any rowset-modification attempt can be canceled prior to the application of the modification by SQLOLEDB. That is, rowset modifications can be canceled when an **IRowsetNotify** member function indicates the event phase DBEVENTPHASE\_OKTODO or DBEVENTPHASE\_ABOUTTODO.

# Data Types (OLE DB)

To execute Transact-SQL statements and process the results by using the SQLOLEDB provider, you need to know how the SQLOLEDB provider maps Microsoft® SQL Server<sup>TM</sup> 2000 data types to OLE DB data types when binding parameters or columns in a rowset, and when using the **ITableDefinition** interface to create a table in SQL Server 2000.

# **Data Type Mapping in Rowsets and Parameters**

In rowsets and as parameter values, SQLOLEDB represents Microsoft® SQL Server<sup>TM</sup> 2000 data by using the following OLE DB defined data types, reported in the functions **IColumnsInfo::GetColumnInfo** and **ICommandWithParameters::GetParameterInfo**.

SQL Server data type	SQLOLEDB data type
bigint	DBTYPE_I8
binary	DBTYPE_BYTES
bit	DBTYPE_BOOL
char	DBTYPE_STR
datetime	DBTYPE_DBTIMESTAMP
decimal	DBTYPE_NUMERIC
float	DBTYPE_R8
image	DBTYPE_BYTES
int	DBTYPE_I4
money	DBTYPE_CY
nchar	DBTYPE_WSTR
ntext	DBTYPE_WSTR
numeric	DBTYPE_NUMERIC
nvarchar	DBTYPE_WSTR
real	DBTYPE_R4
smalldatetime	DBTYPE_DBTIMESTAMP
smallint	DBTYPE_I2
smallmoney	DBTYPE_CY
sql_variant	DBTYPE_VARIANT,
	DBTYPE_SQLVARIANT*
sysname	DBTYPE_WSTR
text	DBTYPE_STR
timestamp	DBTYPE_BYTES
tinyint	DBTYPE_UI1
uniqueidentifier	DBTYPE_GUID

varbinary	DBTYPE_BYTES
varchar	DBTYPE_STR

SQLOLEDB supports consumer-requested data conversions as shown in the illustration.

To data type:	Γ																												É	L
Form data to a	DBTYPE EMPTY	1				DBTYPE R8		- 1	- 1	- 1	OBTYPE_ERROR		DBTYPE VARIANT	DBTYPE_IUNKNOWN	DBTYPE_DECIMAL		DBTYPE UI1		- 1					DBTYPE STR	DBTYPE WSTR	DBTYPE NUMERIC	DBTYPE DBDATE	DBTYPE_DBTIME	DBTYPE_DBTIMESTAMP	DBTYPE_SLQVARIANT
From data type:	_		_		_		_		_				_			_	_	_	_	_	_		_					0	의	9
DBTYPE_EMPTY	•	0	0	0	9	0	9	0	9			0	=		0	0	이		0	0	0	0		0	0	0				
DBTYPE_NULL		•	_										0		Ļ									_	_					0
DBTYPE_12	_			0	_	_	_	_	_			0			0		$\rightarrow$	_							0					0
DBTYPE_14	_			•	_	_	_	$\overline{}$	_			0			0		$\overline{}$	$\rightarrow$							0					0
DBTYPE_R4				0								0			0		_	_							0					0
DBTYPE_R8	_			0	_	_	_	_	_			0			0	_	_	_							0					0
DBTYPE_CY		_		0	_	_	=	_	=			0			0	$\rightarrow$	_	_							0					0
DBTYPE_DATE			_	0	$\rightarrow$	_	_	_	_		_	0	=	_	0	_	_	-	_		_		=	=		=	=		_	0
DBTYPE_BSTR	_	_	0	0	9	0	9	0	•			0	=			이	이	이	이	0	0	0	0	0	0	0	0	0	0	0
DBTYPE_IDISPATCH	0	0			4	1	4	4		•		_	_	0			1	4												
DBTYPE_ERROR					4	4	4	4	_		•		0					4			_									
DBTYPE_BOOL	_	$\overline{}$		0	-	_	_	_	$\rightarrow$			◙			0	$\rightarrow$	$\rightarrow$	_	$\rightarrow$						0					0
DBTYPE_VARIANT			0	0	9	0	9	0	이	0	0	_	_	_	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
DBTYPE_IUNKNOWN	0				4			4	_	0				•				4												
DBTYPE_DECIMAL	_			0	_	$\overline{}$	_	$\overline{}$	$\rightarrow$			0			▣		$\overline{}$	$\rightarrow$							0					0
DBTYPE_I1	_	$\overline{}$		0	_	_	_	_	$\rightarrow$			0			0	$\rightarrow$	$\rightarrow$	$\rightarrow$	$\rightarrow$						0					0
DBTYPE_UI1	_			0	_	_	_	_	_			0			0	0	•	0	0	0	0			0	0	0				0
DBTYPE_UI2	0	0	0	0	O	0	0	0	0			0	0		О	0	0	•	0	0	0			0	0	0				0
DBTYPE_UI4				0			0	0				0	0		0	0	0	0	•	0	0				0	0				0
DBTYPE_I8	0	0	0	0		0										0	0	0	0	•	0			0	0					0
DBTYPE_UI8	0	0	0	0	0	0		-	0							0	0	0	0	0	•			0	0					
DBTYPE_GUID	0	0							0													•		0	0					0
DBTYPE_BYTES	0	0			T		П		0				0						0				•	0	0					0
DBTYPE_STR	0	0	0	0	0	0	0	0	0		1	0	0		0	0	0	0	0	0	0	0	0	•	0	0				0
DBTYPE_WSTR	0	0	0	0	0	0	0	0	0			0	0		0															0
DBTYPE_NUMERIC	0	0	0	0	0	0	0	0	0			0	0		0	0	0	0	0	0	0			0	0	•				0
DBTYPE DBDATE	0	0					0	0					0											0	0		•	0	0	0
DBTYPE_DBTIME	0	0				1	0	0					0											0	0		0	•	0	0
DBTYPE_DBTIMESTAMP	0	0				1	0	0					0											0	0		0	0	•	0
DBTYPE_SQLVARIANT			0	0	0	0	0	0	0			0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	•

- Supported conversion. No data loss occurs.
- O Supported conversion. Data loss can occur.

#### sql\_variant (OLE DB)

The **sql\_variant** data type column can contain any of the data types in SQL Server except large objects (LOBs), such as **text**, **ntext**, and **image**. For example, the column can contain **smallint** values for some rows, **float** values for other rows, and **char/nchar** values in the remainder.

The **sql\_variant** data type is similar to the **variant** data type in Microsoft Visual

Basic® and the DBTYPE VARIANT in OLE DB.

When **sql\_variant** data is fetched as DBTYPE\_VARIANT (defined in Oledb.h), it is put in a VARIANT structure (defined in Oaidl.h) in the buffer. But the subtypes in the VARIANT structure may not map to subtypes defined in the **sql\_variant** data type. The **sql\_variant** data must then be fetched as DBTYPE\_SQLVARIANT in order for all the subtypes to match.

#### DBTYPE\_SQLVARIANT Data Type

To support the **sql\_variant** data type, the Microsoft OLE DB Provider for SQL Server (SQLOLEDB) exposes a provider-specific data type called DBTYPE\_SQLVARIANT. When **sql\_variant** data is fetched in as DBTYPE\_SQLVARIANT (defined in Sqloleb.h), it is stored in a provider-specific SSVARIANT structure (defined in Sqloledb.h). The SSVARIANT structure contains all of the subtypes that match the subtypes of the **sql\_variant** data type.

The session property SSPROP\_ALLOWNATIVEVARIANT must also be set to TRUE.

#### Provider-Specific Property SSPROP\_ALLOWNATIVEVARIANT

In fetching data, you can specify explicitly what kind of data type should be returned for a column or for a parameter. **IColumnInfo** can also be used to get the column information and use that to do the binding. When **IColumnInfo** is used to obtain column information for binding purposes, if the SSPROP\_ALLOWNATIVEVARIANT session property is FALSE (default value), DBTYPE\_VARIANT is returned for **sql\_variant** columns. If SSPROP\_ALLOWNATIVEVARIANT property is FALSE DBTYPE\_SQLVARIANT is not supported. If SSPROP\_ALLOWNATIVEVARIANT property is set to TRUE, the column type is returned as DBTYPE\_SQLVARIANT, in which case the buffer will hold the SSVARIANT structure. In fetching **sql\_variant** data as DBTYPE\_SQLVARIANT, the session property SSPROP\_ALLOWNATIVEVARIANT must be set to TRUE.

SSPROP\_ALLOWNATIVEVARIANT property is part of the provider-specific DBPROPSET\_SQLSERVERSESSION property set, and is a session property.

DBTYPE\_VARIANT applies to all other OLE DB providers.

DBTYPE\_VARIANT is defined in Oledb.h, whereas DBTYPE\_SQLVARIANT is specific to SQL Server and is defined in Sqloledb.h.

#### SSPROP\_ALLOWNATIVEVARIANT

SSPROP\_ALLOWNATIVEVARIANT is a session property and is part of DBPROPSET, SQLServer Session property set.

SSPROP_ALLOWNATIVEVARIANT	Type: VT_BOOL
	R/W: Read/Write
	Default: VARIANT_FALSE
	Description: Determines if the data
	fetched in is as
	DBTYPE_VARIANT or
	DBTYPE_SQLVARIANT.
	VARIANT_TRUE: Column type is returned as DBTYPE_SQLVARIANT in which case the buffer will hold SSVARIANT structure.  VARIANT_FALSE: Column type
	is returned as DBTYPE_VARIANT and the buffer will have VARIANT structure.

## **Data Type Mapping in ITableDefinition**

When creating tables by using the **ITableDefinition::CreateTable** function, the SQLOLEDB consumer can specify Microsoft® SQL Server<sup>TM</sup> 2000 data types in the *pwszTypeName* member of the DBCOLUMNDESC array that is passed. If the consumer specifies the data type of a column by name, then OLE DB data type mapping, represented by the *wType* member of the DBCOLUMNDESC structure, is ignored.

When specifying new column data types with OLE DB data types using the DBCOLUMNDESC structure *wType* member, SQLOLEDB maps OLE DB data types as follows.

OLE DB data type	SQL Server data type	Additional information
DBTYPE_BOOL	bit	
DBTYPE_BYTES	binary, varbinary, or image	SQLOLEDB inspects the ulColumnSize member of the DBCOLUMNDESC structure. Based on the value, and version of the SQL Server 2000 instance, SQLOLEDB maps the type to image.  If the value of ulColumnSize is smaller than the maximum length of a binary data type column, then SQLOLEDB inspects the DBCOLUMNDESC rgPropertySets member. If DBPROP_COL_FIXEDLENGTH is VARIANT_TRUE, SQLOLEDB maps the type to binary. If the value of the property is VARIANT_FALSE, SQLOLEDB maps the type to

		varbinary. In either case, the DBCOLUMNDESC ulColumnSize member determines the width of the SQL Server 2000 column created.
DBTYPE_CY	money	
DBTYPE_DBTIMESTAMP	datetime	
DBTYPE_GUID	uniqueidentifier	
DBTYPE_I2	smallint	
DBTYPE_I4	int	
DBTYPE_NUMERIC	numeric	SQLOLEDB inspects the
		DBCOLUMDESC bPrecision and
		bScale members to determine
		precision and scale for the
	_	numeric column.
DBTYPE_R4	real	
DBTYPE_R8	float	
DBTYPE_STR		SQLOLEDB inspects the
	or <b>text</b>	ulColumnSize member of the
		DBCOLUMNDESC structure.
		Based on the value and version of the SQL Server 2000 instance, SQLOLEDB maps the type to <b>text.</b>
		If the value of <i>ulColumnSize</i> is smaller than the maximum length of a multibyte character data type column, then SQLOLEDB inspects the DBCOLUMNDESC <i>rgPropertySets</i> member. If DBPROP_COL_FIXEDLENGTH is VARIANT_TRUE, SQLOLEDB maps the type to <b>char</b> . If the value of the property is VARIANT_FALSE,

		SQLOLEDB maps the type to <b>varchar</b> . In either case, the DBCOLUMNDESC <i>ulColumnSize</i> member determines the width of the SQL Server 2000 column created.
DBTYPE_UI1	tinyint	
DBTYPE_WSTR	nchar, nvarchar, or ntext	Using DBTYPE_WSTR to define a column is supported for SQL Server version 7.0 servers only.  SQLOLEDB inspects the ulColumnSize member of the DBCOLUMNDESC structure. Based on the value, SQLOLEDB maps the type to <b>ntext</b> .
		If the value of <i>ulColumnSize</i> is smaller than the maximum length of a Unicode character data type column, then SQLOLEDB inspects the DBCOLUMNDESC <i>rgPropertySets</i> member. If DBPROP_COL_FIXEDLENGTH is VARIANT_TRUE, SQLOLEDB maps the type to <b>nchar</b> . If the value of the property is VARIANT_FALSE, SQLOLEDB maps the type to <b>nvarchar</b> . In either case, the DBCOLUMNDESC <i>ulColumnSize</i> member determines the width of the SQL Server 2000 column created.

**Note** When creating a new table, SQLOLEDB maps only the OLE DB data type

enumeration values specified in the preceding table. Attempting to create a table with a column of any other OLE DB data type generates an error.

## **Schema Rowset Support in SQLOLEDB**

If you connect to an earlier version of Microsoft® SQL Server<sup>TM</sup> 2000, you must upgrade the catalog stored procedures on that server before the SQLOLEDB provider can give proper results in schema rowsets. SQLOLEDB also supports returning schema information from a linked server when processing Transact-SQL distributed queries.

The following tables list schema rowsets and the restriction columns supported by SQLOLEDB.

Schema rowset	Restriction columns
DBSCHEMA_CATALOGS	CATALOG_NAME
DBSCHEMA_COLUMN_PRIVILEGES	All the restrictions are supported.
	TABLE_CATALOG TABLE_SCHEMA TABLE_NAME COLUMN_NAME GRANTOR GRANTEE
DBSCHEMA_COLUMNS	All the restrictions are supported.  TABLE_CATALOG TABLE_SCHEMA TABLE_NAME COLUMN_NAME
DBSCHEMA_FOREIGN_KEYS	All restrictions are supported.  PK_TABLE_CATALOG PK_TABLE_SCHEMA PK_TABLE_NAME FK_TABLE_CATALOG

	FK_TABLE_SCHEMA FK_TABLE_NAME
DBSCHEMA_INDEXES	Restrictions 1, 2, 3, and 5 are supported.
	TABLE_CATALOG TABLE_SCHEMA INDEX_NAME TABLE_NAME
DBSCHEMA_PRIMARYKEYS	All restrictions are supported.
	TABLE_CATALOG TABLE_SCHEMA TABLE_NAME
DBSCHEMA_PROCEDURE_PARAMETERS	All restrictions are supported.
	PROCEDURE_CATALOG PROCEDURE_SCHEMA PROCEDURE_NAME PARAMETER_NAME
DBSCHEMA_PROCEDURES	Restrictions 1, 2, and 3 are supported.
	PROCEDURE_CATALOG PROCEDURE_SCHEMA PROCEDURE_NAME
DBSCHEMA_PROVIDER_TYPES	All restrictions are supported.
	DATA_TYPE BEST_MATCH
DBSCHEMA_SCHEMATA	All restrictions are supported.

	CATALOG_NAME SCHEMA_NAME SCHEMA_OWNER
DBSCHEMA_STATISTICS	All restrictions are supported.  TABLE_CATALOG
	TABLE_CATALOG TABLE_SCHEMA TABLE_NAME
DBSCHEMA_TABLE_CONSTRAINTS	All restrictions are supported.
	CONSTRAINT_CATALOG CONSTRAINT_SCHEMA CONSTRAINT_NAME TABLE_CATALOG TABLE_SCHEMA TABLE_NAME CONSTRAINT_TYPE
DBSCHEMA_TABLE_PRIVILEGES	All restrictions are supported.
	TABLE_CATALOG TABLE_SCHEMA TABLE_NAME GRANTOR GRANTEE
DBSCHEMA_TABLES	All restrictions are supported.
	TABLE_CATALOG TABLE_SCHEMA TABLE_NAME TABLE_TYPE
DBSCHEMA_TABLES_INFO	All restrictions are supported.

TABLE_CATALOG
TABLE_SCHEMA
TABLE_NAME
TABLE_TYPE

### **Catalog Stored Procedures**

To support reporting of schema data, Microsoft® SQL Server™ 2000 client interfaces rely on system stored procedures that extract data from a server's catalog. As client software evolves, the catalog stored procedures also evolve.

When a SQLOLEDB consumer connects to SQL Server version 6.5, SQLOLEDB returns an informational error message stating that the catalog stored procedures are out of date.

SQLOLEDB is compatible with earlier versions of SQL Server. However, not all schema rowsets are supported on earlier versions of SQL Server unless the catalog stored procedures are upgraded to the current release level.

To upgrade the catalog stored procedures, use an appropriate client utility to run the Transact-SQL Instcat.sql script that ships with the most recent version of SQLOLEDB. Instcat.sql requires system administrator privilege.

Depending on the version of the server, Instcat.sql execution can generate many error messages. All generated errors can be safely ignored if the final line of execution output indicates success.

## **Distributed Query Support in Schema Rowsets**

To support Microsoft® SQL Server™ 2000 distributed queries, the SQLOLEDB **IDBSchemaRowset** interface returns meta data on linked servers.

If the DBPROPSET\_SQLSERVERSESSION property SSPROP\_QUOTEDCATALOGNAMES is VARIANT\_TRUE, a quoted identifier can be specified for the catalog name (for example "my.catalog"). When restricting schema rowset output by catalog, SQLOLEDB recognizes a two-part name containing the linked server and catalog name. For the schema rowsets in the table below, specifying a two-part catalog name as <code>linked\_server.catalog</code> restricts output to the applicable catalog of the named linked server.

DBSCHEMA_CATALOGS	CATALOG_NAME
DBSCHEMA_COLUMNS	TABLE_CATALOG
DBSCHEMA_PRIMARY_KEYS	TABLE_CATALOG
DBSCHEMA_TABLES	TABLE_CATALOG
DBSCHEMA_FOREIGN_KEYS	PK_TABLE_CATALOG
	FK_TABLE_CATALOG
DBSCHEMA_INDEXES	TABLE_CATALOG
DBSCHEMA_COLUMN_PRIVILEGES	TABLE_CATALOG
DBSCHEMA_TABLE_PRIVILEGES	TABLE_CATALOG

**Note** To restrict a schema rowset to all catalogs from a linked server, use the syntax *linked\_server*. (where the period separator is part of the name specification). This syntax is equivalent to specifying NULL for the catalog name restriction and is also used when the linked server indicates a data source that does not support catalogs.

SQLOLEDB defines the schema rowset LINKEDSERVERS, returning a list of OLE DB data sources registered as linked servers.

#### **See Also**

### LINKEDSERVERS Rowset (OLE DB)

#### **Transactions**

SQLOLEDB implements local transaction support. The consumer can use distributed or coordinated transactions by using Microsoft Distributed Transaction Coordinator (MS DTC). For consumers requiring transaction control that spans multiple sessions, SQLOLEDB can join transactions initiated and maintained by MS DTC.

By default, SQLOLEDB uses an autocommit transaction mode, where each discrete action on a consumer session comprises a complete transaction against an instance of Microsoft® SQL Server<sup>TM</sup> 2000. SQLOLEDB autocommit mode is local and autocommit transactions never span more than a single session.

SQLOLEDB exposes the **ITransactionLocal** interface, allowing the consumer to use explicitly and implicitly started transactions on a single connection to an instance of SQL Server 2000. SQLOLEDB does not support nested local transactions.

### **Supporting Local Transactions**

A session delimits transaction scope for a SQLOLEDB local transaction. When, at the direction of a consumer, SQLOLEDB submits a request to a connected Microsoft® SQL Server<sup>TM</sup> 2000 instance, the request constitutes a unit of work for SQLOLEDB. Local transactions always wrap one or more units of work on a single SQLOLEDB session.

Using the default SQLOLEDB autocommit mode, a single unit of work is treated as the scope of a local transaction. Only one unit participates in the local transaction. When a session is created, SQLOLEDB begins a transaction for the session. Upon successful completion of a work unit, the work is committed. On failure, any work begun is rolled back and the error is reported to the consumer. In either case, SQLOLEDB begins a new local transaction for the session so that all work is conducted within a transaction.

The SQLOLEDB consumer can direct more precise control over local transaction scope by using the **ITransactionLocal** interface. When a consumer session initiates a transaction, all session work units between the transaction start point and the eventual **Commit** or **Abort** method calls are treated as an atomic unit. SQLOLEDB implicitly begins a transaction when directed to do so by the consumer. If the consumer does not request retention, the session reverts to parent transaction-level behavior, most commonly autocommit mode.

SQLOLEDB supports **ITransactionLocal::StartTransaction** parameters as follows.

Parameter	Description
IsoLevel	In local transactions, SQLOLEDB supports
	ISOLATIONLEVEL_READCOMMITTED,
	ISOLATIONLEVEL_REPEATABLEREAD,
	ISOLATIONLEVEL_ISOLATED, and the synonyms
	ISOLATIONLEVEL_CURSORSTABILITY and
	ISOLATIONLEVEL_SERIALIZABLE.
IsoFlags	SQLOLEDB returns an error for any value other than
	zero.
POtherOptions	If not NULL, SQLOLEDB requests the options object

	from the interface. SQLOLEDB returns  XACT_E_NOTIMEOUT if the options object's
	ulTimeout member is not zero. SQLOLEDB ignores
	the value of the <i>szDescription</i> member.
<b>PulTransactionLevel</b>	If not NULL, SQLOLEDB returns the nested level of
	the transaction.

For local transactions, SQLOLEDB implements **ITransaction::Abort** parameters as follows.

Parameter	Description
pboidReason	Ignored if set. Can safely be NULL.
Fretaining	When TRUE, a new transaction is implicitly begun for the session. The transaction must be committed or terminated by the consumer. When FALSE, SQLOLEDB reverts to autocommit mode for the
	session.
Fasync	Asynchronous abort is not supported by SQLOLEDB. SQLOLEDB returns XACT_E_NOTSUPPORTED if the value is not FALSE.

For local transactions, SQLOLEDB implements **ITransaction::Commit** parameters as follows.

Parameter	Description
fRetaining	When TRUE, a new transaction is implicitly begun for
	the session. The transaction must be committed or
	terminated by the consumer. When FALSE,
	SQLOLEDB reverts to autocommit mode for the
	session.
GrfTC	Asynchronous and phase one returns are not supported
	by SQLOLEDB. SQLOLEDB returns
	XACT_E_NOTSUPPORTED for any value other than
	XACTTC_SYNC.
GrfRM	Must be 0.

SQLOLEDB rowsets on the session are preserved on a local commit or abort operation based on the values of the rowset properties DBPROP\_ABORTPRESERVE and DBPROP\_COMMITPRESERVE. By default, these properties are both VARIANT\_FALSE and all SQLOLEDB rowsets on the session are lost following an abort or commit operation.

SQLOLEDB does not implement the **ITransactionObject** interface. A consumer attempt to retrieve a reference on the interface returns E\_NOINTERFACE.

This example uses **ITransactionLocal**.

```
// Interfaces used in the example.
IDBCreateSession* pIDBCreateSession = NULL;
ITransaction*
                pITransaction
                                 = NULL;
IDBCreateCommand* pIDBCreateCommand = NULL;
                               = NULL;
               pIRowset
IRowset*
HRESULT
                 hr;
// Get the command creation and local transaction interfaces for the
// session.
if (FAILED(hr = pIDBCreateSession->CreateSession(NULL,
  IID_IDBCreateCommand, (IUnknown**) &pIDBCreateCommand)
  // Process error from session creation. Release any references and
  // return.
if (FAILED(hr = pIDBCreateCommand->QueryInterface(IID_ITransac
  (void**) &pITransaction)))
  // Process error. Release any references and return.
```

```
// Start the local transaction.
if (FAILED(hr = ((ITransactionLocal*) pITransaction)->StartTransacti
  ISOLATIONLEVEL_REPEATABLEREAD, 0, NULL, NULL)))
  // Process error from StartTransaction. Release any references and
  // return.
// Get data into a rowset, then update the data. Functions are not
// illustrated in this example.
if (FAILED(hr = ExecuteCommand(pIDBCreateCommand, &pIRowse
  // Release any references and return.
// If rowset data update fails, then terminate the transaction, else
// commit. The example doesn't retain the rowset.
if (FAILED(hr = UpdateDataInRowset(pIRowset, bDelayedUpdate)))
  // Get error from update, then terminate.
  pITransaction->Abort(NULL, FALSE, FALSE);
else
  if (FAILED(hr = pITransaction->Commit(FALSE, XACTTC_SYNC
    // Get error from failed commit.
if (FAILED(hr))
  // Update of data or commit failed. Release any references and
```

```
// return.
}
// Release any references and continue.
```

### **Supporting Distributed Transactions**

SQLOLEDB consumers can use the **ITransactionJoin::JoinTransaction** method to participate in a distributed transaction coordinated by MS DTC.

MS DTC exposes COM objects that allow clients to initiate and participate in coordinated transactions across multiple connections to a variety of data stores. To initiate a transaction, the SQLOLEDB consumer uses the MS DTC ITransactionDispenser interface. The BeginTransaction member of ITransactionDispenser returns a reference on a distributed transaction object. This reference is passed to SQLOLEDB using JoinTransaction.

MS DTC supports asynchronous commit and abort on distributed transactions. For notification on asynchronous transaction status, the consumer implements the **ITransactionOutcomeEvents** interface and connects the interface to an MS DTC transaction object.

For distributed transactions, SQLOLEDB implements **ITransactionJoin::JoinTransaction** parameters as follows.

Parameter	Description
punkTransactionCoord	A pointer to an MS DTC transaction object.
IsoLevel	Ignored by SQLOLEDB. The isolation level for MS DTC-coordinated transactions is determined when the consumer acquires a transaction object from MS DTC.
5	Must be 0. SQLOLEDB returns  XACT_E_NOISORETAIN if any other value is specified by the consumer.
•	If not NULL, SQLOLEDB requests the options object from the interface. SQLOLEDB returns XACT_E_NOTIMEOUT if the options object's <i>ulTimeout</i> member is not zero. SQLOLEDB ignores the value of the <i>szDescription</i> member.

This example coordinates transaction by using MS DTC.

```
// SQLOLEDB interfaces used in the example.
IDBCreateSession*
                     pIDBCreateSession = NULL;
                  pITransactionJoin = NULL;
ITransactionJoin*
IDBCreateCommand*
                        pIDBCreateCommand = NULL;
                                 = NULL:
IRowset*
                 pIRowset
// Transaction dispenser and transaction from MS DTC.
ITransactionDispenser* pITransactionDispenser = NULL;
ITransaction*
                  pITransaction
                                   = NULL:
  HRESULT
                   hr;
// Get the command creation interface for the session.
if (FAILED(hr = pIDBCreateSession->CreateSession(NULL,
  IID_IDBCreateCommand, (IUnknown**) &pIDBCreateCommand)
  // Process error from session creation. Release any references and
  // return.
// Get a transaction dispenser object from MS DTC and
// start a transaction.
if (FAILED(hr = DtcGetTransactionManager(NULL, NULL,
  IID_ITransactionDispenser, 0, 0, NULL,
  (void**) &pITransactionDispenser)))
  // Process error message from MS DTC, release any references,
  // and then return.
if (FAILED(hr = pITransactionDispenser->BeginTransaction()
  NULL, ISOLATIONLEVEL READCOMMITTED, ISOFLAG RE
  NULL, &pITransaction)))
```

```
// Process error message from MS DTC, release any references,
  // and then return.
// Join the transaction.
if (FAILED(pIDBCreateCommand->QueryInterface(IID_ITransaction.
  (void**) &pITransactionJoin)))
  // Process failure to get an interface, release any references, and
  // then return.
if (FAILED(pITransactionJoin->JoinTransaction(
  (IUnknown*) pITransaction, 0, 0, NULL)))
  // Process join failure, release any references, and then return.
// Get data into a rowset, then update the data. Functions are not
// illustrated in this example.
if (FAILED(hr = ExecuteCommand(pIDBCreateCommand, &pIRowse
  // Release any references and return.
// If rowset data update fails, then terminate the transaction, else
// commit. The example doesn't retain the rowset.
if (FAILED(hr = UpdateDataInRowset(pIRowset, bDelayedUpdate)))
  // Get error from update, then abort.
  pITransaction->Abort(NULL, FALSE, FALSE);
else
```

```
if (FAILED(hr = pITransaction->Commit(FALSE, 0, 0)))
    // Get error from failed commit.
    // If a distributed commit fails, application logic could
    // analyze failure and retry. In this example, terminate. The
    // consumer must resolve this somehow.
    pITransaction->Abort(NULL, FALSE, FALSE);
  }
if (FAILED(hr))
  // Update of data or commit failed. Release any references and
  // return.
// Un-enlist from the distributed transaction by setting
// the transaction object pointer to NULL.
if (FAILED(pITransactionJoin->JoinTransaction()
  (IUnknown*) NULL, 0, 0, NULL)))
  // Process failure, and then return.
// Release any references and continue.
```

### **Isolation Levels in SQLOLEDB**

Microsoft® SQL Server™ 2000 clients can control transaction-isolation levels for a connection. To control transaction-isolation level, the SQLOLEDB consumer uses:

DBPROPSET\_SESSION property
 DBPROP\_SESS\_AUTOCOMMITISOLEVELS for SQLOLEDB default autocommit mode.

The SQLOLEDB default for the level is DBPROPVAL\_TI\_READCOMMITTED.

- The *isoLevel* parameter of the **ITransactionLocal::StartTransaction** method for local manual-commit transactions.
- The *isoLevel* parameter of the
   ITransactionDispenser::BeginTransaction method for MS DTC-coordinated distributed transactions.

SQL Server 2000 allows read-only access at the dirty read isolation level. All other levels restrict concurrency by applying locks to SQL Server 2000 objects. As the client requires greater concurrency levels, SQL Server 2000 applies greater restrictions on concurrent access to data. To maintain the highest level of concurrent access to data, the SQLOLEDB consumer should intelligently control its requests for specific concurrency levels.

#### See Also

<u>Isolation Levels</u>

### **SQLOLEDB** Enumerator

Each OLE DB provider has an enumerator that a consumer can call to get a list of data sources that the consumer can access with that provider. The SQLOLEDB provider has an enumerator that lists all servers you can connect to with this provider.

For a client running on the Microsoft® Windows® 95 or Windows 98 operating system, the SQLOLEDB enumerator cannot enumerate the list of servers running Microsoft SQL Server™ 2000 because the enumerator uses the **NetServerEnum** API. This API is not available for the Windows 95 and Windows 98 operating systems (it is available only for the Microsoft Windows NT® 4.0 and Windows 2000 operating systems).

To enumerate OLE DB data sources

## **Errors**

OLE/COM objects report errors through the HRESULT return code of object member functions. An OLE/COM HRESULT is a bit-packed structure. OLE provides macros that dereference structure members.

OLE/COM specifies the **IErrorInfo** interface. The interface exposes methods such as **GetDescription**, allowing clients to extract error details from OLE/COM servers. OLE DB extends **IErrorInfo** to support the return of multiple error information packets on a single-member function execution.

SQLOLEDB exposes the OLE DB record-enhanced **IErrorInfo**, the custom **ISQLErrorInfo**, and the provider-specific **ISQLServerErrorInfo** error object interfaces.

## **Return Codes**

At the most basic level, a member function either succeeds or fails. At a somewhat more precise level, a function can succeed, but its success may not be identical to that intended by the application developer.

When a SQLOLEDB member function returns S OK, the function succeeded.

When a SQLOLEDB member function does not return S\_OK, the OLE/COM HRESULT-unpacking FAILED and IS\_ERROR macros can determine the overall success or failure of a function.

If FAILED or IS\_ERROR returns TRUE, the SQLOLEDB consumer is assured that member function execution failed. When FAILED or IS\_ERROR return FALSE, and the HRESULT does not equal S\_OK, then the SQLOLEDB consumer is assured that the function succeeded in some sense. The consumer can retrieve detailed information on this success-with-information return from SQLOLEDB error interfaces. Also, in the case where a function clearly fails (the FAILED macro returns TRUE), extended error information is available from the SQLOLEDB error interfaces.

SQLOLEDB consumers commonly encounter the DB\_S\_ERRORSOCCURRED success-with-information HRESULT return. Typically, member functions that return DB\_S\_ERRORSOCCURRED define one or more parameters that deliver status values to the consumer. No error information may be available to the consumer other than that returned in status-value parameters, so consumers should implement application logic that retrieves status values when they are available.

SQLOLEDB member functions do not return the success code S\_FALSE. Any SQLOLEDB member function always returns S\_OK to indicate success.

## **Information in OLE DB Error Interfaces**

SQLOLEDB reports some error and status information in the OLE DB-defined error interfaces **IErrorInfo**, **IErrorRecords**, and **ISQLErrorInfo**.

SQLOLEDB supports **IErrorInfo** member functions as follows.

Member function	Description	
GetDescription	Descriptive error message string.	
GetGUID	GUID of the interface that defined the error.	
GetHelpContext	Not supported. Returns zero always.	
GetHelpFile	Not supported. Returns NULL always.	
GetSource	String Sqloledb.dll.	

SQLOLEDB supports consumer-available **IErrorRecords** member functions as follows.

Member function	Description		
GetBasicErrorInfo	Fills an ERRORINFO structure with basic		
	information about an error. An ERRORINFO		
	structure contains members that identify the		
	HRESULT return value for the error, and the		
	provider and interface on which the error applies.		
GetCustomErrorObject	ect Returns a reference on interfaces		
	ISQLErrorInfo, and ISQLServerErrorInfo.		
GetErrorInfo	Returns a reference on an <b>IErrorInfo</b> interface.		
GetErrorParameters	SQLOLEDB does not return parameters to the		
	consumer through <b>GetErrorParameters</b> .		
GetRecordCount	Count of error records available.		

SQLOLEDB supports **ISQLErrorInfo::GetSQLInfo** parameters as follows.

Parameter	Description	
pbstrSQLState	Returns a SQLSTATE value for the error.	

	SQLSTATE values are defined in the SQL-92, ODBC and ISO SQL, and API specifications. Neither Microsoft® SQL Server™ 2000 nor SQLOLEDB define implementation-specific SQLSTATE values.	
plNativeError	Returns the SQL Server 2000 error number from <b>master.dbo.sysmessages</b> when available. Native errors are available after a successful attempt to initialize a SQLOLEDB data source. Prior to the attempt, SQLOLEDB always returns zero.	

## **SQL Server Error Detail**

SQLOLEDB defines the provider-specific error interface **ISQLServerErrorInfo**. The interface returns more detail about a Microsoft® SQL Server<sup>TM</sup> 2000 error and is valuable when command execution or rowset operations fail.

There are two ways to obtain access to **ISQLServerErrorInfo** interface.

The consumer may call <code>IErrorRecords::GetCustomerErrorObject</code> (no need to obtain <code>ISQLErrorInfo</code>) to obtain an <code>ISQLServerErrorInfo</code> pointer (as shown in the following code sample). Both <code>ISQLErrorInfo</code> and <code>ISQLServerErrorInfo</code> are custom OLE DB error objects, with <code>ISQLServerErrorInfo</code> being the interface to use to obtain information of server errors, including such details as procedure name and line numbers.

Another way to get an **ISQLServerErrorInfo** pointer is to call the **QueryInterface** method on an already obtained **ISQLErrorInfo** pointer. Note that because **ISQLServerErrorInfo** contains a superset of the information available from **ISQLErrorInfo**, it makes sense to go directly to **ISQLServerErrorInfo** through **GetCustomerErrorObject**.

The **ISQLServerErrorInfo** interface exposes one member function, **GetErrorInfo**. The function returns a pointer to an SSERRORINFO structure and a pointer to a string buffer. Both pointers reference memory the consumer must deallocate by using the **IMalloc::Free** method.

SSERRORINFO structure members are interpreted by the consumer as follows.

Member	Description

pwszMessage	SQL Server 2000 error message. Identical to the string returned in <b>IErrorInfo::GetDescription</b> .		
pwszServer	Name of the instance of SQL Server for the session.		
pwszProcedure	If appropriate, the name of the procedure in which the error originated. An empty string otherwise.		
lNative	SQL Server native error number. Identical to the value returned in the <i>plNativeError</i> parameter of <b>ISQLErrorInfo::GetSQLInfo</b> .		
bState	State of a SQL Server 2000 error message.		
bClass	Severity of a SQL Server 2000 error message.		
wLineNumber	When applicable, the line number of a stored procedure on which the error occurred.		

# See Also

**RAISERROR** 

## **SQLOLEDB Example: Retrieving Error Information**

This example obtains information from the various error interfaces exposed by SQLOLEDB.

```
// DumpErrorInfo queries SQLOLEDB error interfaces, retrieving avai
// status or error information.
void DumpErrorInfo
  (
  IUnknown* pObjectWithError,
  REFIID IID InterfaceWithError
  {
  // Interfaces used in the example.
  IErrorInfo*
                    pIErrorInfoAll
                                       = NULL;
  IErrorInfo*
                    pIErrorInfoRecord
                                         = NULL:
  IErrorRecords*
                      pIErrorRecords
                                          = NULL;
  ISupportErrorInfo*
                       pISupportErrorInfo
                                            = NULL;
                      pISQLErrorInfo
  ISQLErrorInfo*
                                           = NULL;
  ISQLServerErrorInfo* pISQLServerErrorInfo
  // Number of error records.
  ULONG
                     nRecs:
  ULONG
                     nRec;
  // Basic error information from GetBasicErrorInfo.
  ERRORINFO
                       errorinfo;
  // IErrorInfo values.
                   bstrDescription;
  BSTR
  BSTR
                   bstrSource;
```

```
// ISQLErrorInfo parameters.
                  bstrSQLSTATE;
BSTR
                   lNativeError;
LONG
// ISQLServerErrorInfo parameter pointers.
SSERRORINFO*
                         pSSErrorInfo = NULL;
OLECHAR*
                      pSSErrorStrings = NULL;
// Hard-code an American English locale for the example.
DWORD
                     MYLOCALEID = 0x0409;
// Only ask for error information if the interface supports
// it.
if (FAILED(pObjectWithError->QueryInterface(IID_ISupportErrorIi
   (void**) &pISupportErrorInfo)))
   wprintf(L"SupportErrorErrorInfo interface not supported");
   return:
   }
if (FAILED(pISupportErrorInfo->
  InterfaceSupportsErrorInfo(IID_InterfaceWithError)))
   wprintf(L"InterfaceWithError interface not supported");
   return;
   }
// Do not test the return of GetErrorInfo. It can succeed and return
// a NULL pointer in pIErrorInfoAll. Simply test the pointer.
GetErrorInfo(0, &pIErrorInfoAll);
if (pIErrorInfoAll != NULL)
   // Test to see if it's a valid OLE DB IErrorInfo interface
```

```
// exposing a list of records.
if (SUCCEEDED(pIErrorInfoAll->QueryInterface(IID_IErrorRec
  (void**) &pIErrorRecords)))
  pIErrorRecords->GetRecordCount(&nRecs);
  // Within each record, retrieve information from each
  // of the defined interfaces.
  for (nRec = 0; nRec < nRecs; nRec++)
     {
    // From IErrorRecords, get the HRESULT and a reference
    // to the ISQLErrorInfo interface.
    pIErrorRecords->GetBasicErrorInfo(nRec, &errorinfo);
    pIErrorRecords->GetCustomErrorObject(nRec,
       IID_ISQLErrorInfo, (IUnknown**) &pISQLErrorInfo);
    // Display the HRESULT, then use the ISQLErrorInfo.
    wprintf(L"HRESULT:\t%#X\n", errorinfo.hrError);
    if (pISQLErrorInfo != NULL)
       pISQLErrorInfo->GetSQLInfo(&bstrSQLSTATE,
         &lNativeError);
       // Display the SQLSTATE and native error values.
       wprintf(L"SQLSTATE:\t%s\nNative Error:\t%ld\n",
         bstrSQLSTATE, lNativeError);
       // SysFree BSTR references.
       SysFreeString(bstrSQLSTATE);
       // Get the ISQLServerErrorInfo interface from
       // ISQLErrorInfo before releasing the reference.
```

```
pISQLErrorInfo->QueryInterface(
     IID_ISQLServerErrorInfo,
     (void**) &pISQLServerErrorInfo);
  pISQLErrorInfo->Release();
// Test to ensure the reference is valid, then
// get error information from ISQLServerErrorInfo.
if (pISQLServerErrorInfo != NULL)
  pISQLServerErrorInfo->GetErrorInfo(&pSSErrorInfo,
     &pSSErrorStrings);
  // ISQLServerErrorInfo::GetErrorInfo succeeds
  // even when it has nothing to return. Test the
  // pointers before using.
  if (pSSErrorInfo)
     // Display the state and severity from the
     // returned information. The error message comes
     // from IErrorInfo::GetDescription.
     wprintf(L"Error state:\t%d\nSeverity:\t%d\n",
          pSSErrorInfo->bState,
          pSSErrorInfo->bClass);
     // IMalloc::Free needed to release references
     // on returned values. For the example, assume
     // the g_pIMalloc pointer is valid.
     g_pIMalloc->Free(pSSErrorStrings);
     g_pIMalloc->Free(pSSErrorInfo);
```

```
pISQLServerErrorInfo->Release();
    if (SUCCEEDED(pIErrorRecords->GetErrorInfo(nRec,
       MYLOCALEID, &pIErrorInfoRecord)))
       // Get the source and description (error message)
       // from the record's IErrorInfo.
       pIErrorInfoRecord->GetSource(&bstrSource);
       pIErrorInfoRecord->GetDescription(&bstrDescription);
       if (bstrSource != NULL)
         wprintf(L"Source:\t\t%s\n", bstrSource);
         SysFreeString(bstrSource);
       if (bstrDescription != NULL)
         wprintf(L"Error message:\t%s\n",
            bstrDescription);
         SysFreeString(bstrDescription);
       pIErrorInfoRecord->Release();
     }
  pIErrorRecords->Release();
else
  // IErrorInfo is valid; get the source and
  // description to see what it is.
```

```
pIErrorInfoAll->GetSource(&bstrSource);
    pIErrorInfoAll->GetDescription(&bstrDescription);
    if (bstrSource != NULL)
       wprintf(L"Source:\t\t%s\n", bstrSource);
       SysFreeString(bstrSource);
    if (bstrDescription != NULL)
       wprintf(L"Error message:\t%s\n", bstrDescription);
       SysFreeString(bstrDescription);
     }
  pIErrorInfoAll->Release();
else
  wprintf(L"GetErrorInfo failed.");
pISupportErrorInfo->Release();
return;
```

## **SQL Server Message Results**

These Transact-SQL statements do not generate SQLOLEDB rowsets or a count of affected rows when executed:

- PRINT
- RAISERROR with a severity of 10 or lower
- DBCC
- SET SHOWPLAN
- SET STATISTICS

These statements either return one or more informational messages, or cause Microsoft® SQL Server<sup>TM</sup> 2000 to return informational messages in place of rowset or count results. On successful execution, SQLOLEDB returns S\_OK and the message or messages are available to the SQLOLEDB consumer.

SQLOLEDB returns S\_OK and has one or more informational messages available following the execution of many Transact-SQL statements or the consumer execution of a SQLOLEDB member function.

The SQLOLEDB consumer allowing dynamic specification of query text should check error interfaces after every member function execution regardless of the value of the return code, the presence or absence of a returned **IRowset** or **IMultipleResults** interface reference, or a count of affected rows.

## **SQL Server OLE DB Programmer's Reference**

SQLOLEDB, the Microsoft OLE DB Provider for SQL Server, exposes interfaces to consumers wanting access to data on one or more computers running Microsoft® SQL Server<sup>TM</sup> 2000. SQLOLEDB is an OLE DB version 2.0–compliant provider.

This OLE DB programming reference does not document all of the OLE DB interfaces and methods, only those interfaces and methods that exhibit provider-specific behavior when using SQLOLEDB. For a full description of the OLE DB API, see the Microsoft OLE DB Software Development Kit (SDK). The OLE DB SDK is part of the Microsoft Developer Network (MSDN®) and can be downloaded from Microsoft Web site.

# **Interfaces (OLE DB)**

The SQLOLEDB provider supports these provider-specific interfaces:

- IRowsetFastLoad
- ISQLServerErrorInfo

# IRowsetFastLoad (OLE DB)

**IRowsetFastLoad** exposes support for Microsoft® SQL Server™ 2000 bulk-copy processing. SQLOLEDB consumers use the interface to rapidly add data to an existing SQL Server 2000 table.

Method	Description	
Commit	Marks the end of a batch of inserted rows and writes	
	the rows to the SQL Server 2000 table.	
InsertRow	Adds a row to the bulk-copy rowset.	

### See Also

**Bulk-Copy Rowsets** 

## IRowsetFastLoad::Commit (OLE DB)

Marks the end of a batch of inserted rows and writes the rows to the Microsoft® SQL Server<sup>TM</sup> 2000 table.

## **Syntax**

#### **HRESULT Commit( BOOLEAN** *bDone***)**;

## **Arguments**

bDone [in]

If FALSE, the rowset maintains validity and can be used by the consumer for additional row insertion. If TRUE, the rowset loses validity and no further insertion can be done by the consumer.

### **Return Code Values**

S\_OK

The method succeeded and all inserted data has been written to the SQL Server 2000 table.

E\_FAIL

An error occurred.

**E\_UNEXPECTED** 

The method was called on a bulk-copy rowset previously invalidated by **IRowsetFastLoad::Commit**.

### Remarks

A SQLOLEDB bulk-copy rowset behaves as a delayed-update mode rowset. As the user inserts row data through the rowset, inserted rows are treated in the same fashion as pending inserts on a rowset supporting **IRowsetUpdate**.

The consumer must call **Commit** on the bulk-copy rowset to write inserted rows to the SQL Server 2000 table in the same way as the **IRowsetUpdate::Update** member function is used to submit pending rows to an instance of SQL Server 2000.

If the consumer releases its reference on the bulk-copy rowset without calling **Commit**, all inserted rows not previously written are lost.

The consumer can batch inserted rows by calling **Commit** with *bDone* FALSE. When *bDone* is TRUE, the rowset becomes invalid. An invalid bulk-copy rowset supports only **ISupportErrorInfo** and **IRowsetFastLoad::Release**.

## IRowsetFastLoad::InsertRow (OLE DB)

Adds a row to the bulk-copy rowset.

## **Syntax**

**HRESULT InsertRow( HACCESSOR** *hAccessor*, **void\*** *pData*);

## **Arguments**

hAccessor [in]

Is the handle of the accessor defining the row data for bulk copy. The accessor referenced is a row accessor, binding consumer-owned memory containing data values.

pData [in]

Is a pointer to the consumer-owned memory containing data values.

#### **Return Code Values**

S\_OK

The method succeeded. Any bound status values for all columns have value DBSTATUS\_S\_OK or DBSTATUS\_S\_NULL.

E\_FAIL

An error occurred. Error information is available from the rowset's error interfaces.

E\_INVALIDARG

*pData* was a NULL pointer.

E\_OUTOFMEMORY

SQLOLEDB was unable to allocate sufficient memory to complete the

request.

#### **E\_UNEXPECTED**

The method was called on a bulk-copy rowset previously invalidated by **IRowsetFastLoad::Commit(TRUE)**.

#### DB\_E\_BADACCESSORHANDLE

The *hAccessor* provided by the consumer was invalid.

#### DB\_E\_BADACCESSORTYPE

The specified accessor was not a row accessor or did not specify consumerowned memory.

#### Remarks

An error converting consumer data to the Microsoft® SQL Server™ 2000 data type for a column causes an E\_FAIL return from SQLOLEDB. Data can be transmitted to SQL Server on any **InsertRow** or only on **Commit**. Therefore, the consumer application can call **InsertRow** many times with erroneous data before it receives notice that a data type conversion error exists. Because **Commit** ensures that all data is correctly specified by the consumer, the consumer can use **Commit** appropriately to validate data as necessary.

SQLOLEDB bulk-copy rowsets are write-only. SQLOLEDB exposes no methods allowing consumer query of the rowset. To terminate processing, the consumer can release its reference on **IRowsetFastLoad** without calling **Commit**. There are no facilities for accessing a consumer-inserted row in the rowset and changing its values, or removing it individually from the rowset.

Bulk-copied rows are formatted on the server for SQL Server version 7.0. The row format is affected by any options that may have been set for the connection or session such as ANSI\_PADDING. This option is set on by default for any connection made through SQLOLEDB. If connected to SQL Server 6.5, the bulk-copied rows are formatted on the client and none of the option settings have any effect.

# ISQLServerErrorInfo (OLE DB)

SQLOLEDB defines the **ISQLServerErrorInfo** error interface. The interface returns details from a Microsoft® SQL Server<sup>TM</sup> 2000 error, including its severity and state.

Method	Description
GetErrorInfo	Returns a pointer to a SQLOLEDB SSERRORINFO
	structure containing SQL Server 2000 error detail.

### See Also

**SQL Server Error Detail** 

# ISQLServerErrorInfo::GetErrorInfo (OLE DB)

Returns a pointer to a SQLOLEDB SSERRORINFO structure containing Microsoft® SQL Server<sup>TM</sup> 2000 error detail.

## **Syntax**

**HRESULT GetErrorInfo**( **SSERRORINFO**\*\**ppSSErrorInfo*, **OLECHAR**\*\**ppErrorStrings*);

## **Arguments**

ppSSErrorInfo [out]

Is a pointer to an SSERRORINFO structure. If the method fails or there is no SQL Server 2000 information associated with an error, the provider does not allocate any memory, and ensures that \*\*ppSSErrorInfo is a null pointer on output.

ppErrorStrings [out]

Is a pointer to a Unicode character-string pointer. If the method fails or there is no SQL Server information associated with an error, the provider does not allocate any memory, and ensures that \*\*ppErrorStrings is a null pointer on output. Freeing ppErrorStrings with the **IMalloc::Free** function frees the three individual string members of the returned SSERRORINFO structure, as the memory is allocated in a block.

### **Return Code Values**

S\_OK

The method succeeded.

**E\_INVALIDARG** 

Either *ppSSErrorInfo* or *ppErrorStrings* was NULL.

**E\_OUTOFMEMORY** 

SQLOLEDB was unable to allocate sufficient memory to complete the request.

#### **Remarks**

SQLOLEDB allocates memory for the SSERRORINFO and OLECHAR strings returned through the pointers passed by the consumer. The consumer must deallocate this memory by using **IMalloc::Free** when it no longer requires access to the error data.

The SSERRORINFO structure is defined as follows:

```
typedef struct tagSSErrorInfo
{
   LPOLESTR pwszMessage;
   LPOLESTR pwszServer;
   LPOLESTR pwszProcedure;
   LONG lNative;
   BYTE bState;
   BYTE bClass;
   WORD wLineNumber;
}
SSERRORINFO;
```

Member	Description
pwszMessage	Error message from SQL Server 2000. The message is returned through the <b>IErrorInfo::GetDescription</b> method.
pwszServer	Name of the instance of SQL Server 2000 on which the error occurred.
pwszProcedure	Name of the stored procedure generating the error if the error occurred in a stored procedure; otherwise, an empty string.
lNative	SQL Server error number. The error number is identical to that returned in the <i>plNativeError</i> parameter of the <b>ISQLErrorInfo::GetSQLInfo</b> method.

bState	State of a SQL Server 2000 error.	
bClass	Severity of a SQL Server 2000 error.	
	When applicable, the line of a SQL Server 2000 stored procedure that generated the error message. The default value if there is no procedure involved is 1.	

Pointers in the structure reference addresses in the string returned in *ppErrorStrings*.

## See Also

**RAISERROR** 

# Schema Rowsets (OLE DB)

SQLOLEDB exposes the database schema rowset LINKEDSERVERS, enumerating organization data sources that can participate in Microsoft® SQL Server<sup>TM</sup> 2000 distributed queries.

### See Also

Schema Rowset Support in SQLOLEDB

# LINKEDSERVERS Rowset (OLE DB)

The LINKEDSERVERS rowset enumerates organization data sources that can participate in Microsoft® SQL Server™ 2000 distributed queries.

The LINKEDSERVERS rowset contains the following columns.

Column name	Type indicator	Description
SVR_NAME	DBTYPE_WSTR	Name of a linked server.
SVR_PRODUCT	DBTYPE_WSTR	Manufacturer or other name
		identifying the type of data store
		represented by the name of the link
		server.
SVR_PROVIDERNAME	DBTYPE_WSTR	Friendly name of the OLE DB
		provider used to consume data from
		the server.
SVR_DATASOURCE	DBTYPE_WSTR	OLE DB
		DBPROP_INIT_DATASOURCE
		string used to acquire a data source
		from the provider.
SVR_PROVIDERSTRING	DBTYPE_WSTR	OLE DB
		DBPROP_INIT_PROVIDERSTR
		value used to acquire a data source
		from the provider.
SVR_LOCATION	DBTYPE_WSTR	OLE DB
		DBPROP_INIT_LOCATION strin
		used to acquire a data source from
		provider.

The rowset is sorted on SRV\_NAME and a single restriction is supported on SRV\_NAME.