IBM

Using UniOLEDB

Documentation Team: Claire Gustafson, Shelley Thompson, Anne Waite

Table of Contents

Appendix A   Examples of Consumer Source Code

Glossary

Introduction to UniOLEDB

This manual provides an overview of OLE DB technology and how the UniOLEDB provider uses it to expose UniData and UniVerse extended relational data. This manual also describes how to configure UniOLEDB, and it provides information about the OLE DB functionality UniOLEDB supports. From this latter information, developers can design OLE DB–based applications that can access and manipulate data in UniData and UniVerse data stores.

For UniData, use this manual in conjunction with the *Using VSG and the Schema API* manual. It describes:

- Visual Schema Generator (VSG) – A Microsoft Windows–based Graphical User Interface (GUI) tool for making data in UniData accessible to first normal form (1NF) applications.

- Schema API – An application programming interface (API) that consists of a series of UniBasic subroutines designed to accomplish the same tasks as VSG.

Note: IBM recommends that you use VSG to prepare files because it is easier and faster to use than the Schema API. It performs several processes automatically. You might want to use the Schema API if you have a large number of files or you deploy data files with your applications.

For UniVerse, Chapter 4, "Accessing UniVerse Data," describes making data in UniVerse accessible to 1NF applications.

This manual is organized in the following way:

- Chapter 1, "Introduction to UniOLEDB," introduces you to key concepts about OLE DB and UniOLEDB, and shows you how UniOLEDB fits in with other components in an enterprise network. This chapter also provides you with the hardware and software requirements for clients and servers that use UniOLEDB.

- Chapter 2, "Setting Up UniOLEDB," describes how to set up the UCI configuration file.

- Chapter 3, "Accessing UniData Data," describes how to make accounts in UniData databases accessible to consumers and normalize multivalued and multi-subvalued data in UniData.

- Chapter 4, "Accessing UniVerse Data," describes how to make data in the schemas and accounts accessible to consumers.

- Chapter 5, "UniOLEDB Functionality," describes how UniOLEDB supports OLE DB functionality.

- Appendix A, "Examples of Consumer Source Code," provides sample source code of consumer applications that can access data in UniData or UniVerse data stores through UniOLEDB.

- Glossary provides an alphabetic listing of words and concepts to help you understand and use OLE DB and UniOLEDB.

This chapter consists of the following sections:

- "Introduction"
- "Overview of OLE DB"
- "Overview of UniOLEDB"
- "Before You Use UniOLEDB"

Introduction

OLE DB is a low-level interface that enables Universal Data Access (UDA), Microsoft's solution to data access challenges in today's highly sophisticated and versatile business data environments. UniOLEDB harnesses this technology to provide applications in an enterprise network with direct access to UniData and UniVerse data stores.

The Data Challenge

Today's business decisions are based on vast and varied stores of information that generally do not share the same characteristics and cannot be coordinated easily by using traditional approaches. The need exists to manage diverse data that:

- Are in different formats, both relational and non-relational.
- Reside in different locations, both local and remote, and on a variety of platforms.
- Are accessed in essentially different ways.

Some examples of data stores that a business organization might want to access and manipulate in a uniform way include:

- 1NF relational databases, such as Microsoft SQL Server, Oracle, and Microsoft Access.
- Extended (NF^2) relational databases, such as UniData and UniVerse.
- Nonrelational data stores, such as:
  - Spreadsheets.
  - Indexed-sequential access method (ISAM) files.
  - Custom business objects.
  - E-mail information, such as in Microsoft Exchange.
  - Document and graphics files.
  - Internet data.

The Data Solution

Universal Data Access (UDA) is Microsoft's solution for accessing data from diverse sources in a way that addresses the challenges of traditional data access methods. It provides a common set of interfaces that offer uniform access to any source of data, wherever it resides. It enables the business to manipulate and manage the data directly. OLE DB is a central part of UDA. It enhances and builds on the innovations of Microsoft's Open Database Connectivity (ODBC) interface, a precursor UDA technology that offers uniform access to relational databases.

UniOLEDB takes advantage of OLE DB technology to expose data in UniData and UniVerse databases to OLE DB–enabled client applications.

UniOLEDB is not built on top of UniData ODBC or UniVerse ODBC. It does not use ODBC as a way to implement OLE DB. UniOLEDB is a native OLE DB provider that takes advantage of OLE DB technology.

Overview of OLE DB

OLE DB is a part of Microsoft's UDA solution for accessing information across an enterprise network, regardless of type, format, or location. It benefits businesses and application developers in the following ways:

- Enables the development of applications that can access and manipulate data from multiple heterogeneous sources simultaneously.

- Provides common interfaces that enable diverse data stores to expose their data homogeneously in a standard tabular form.

- Partitions the functionality of database management systems into logical, reusable, and interoperable components that are not directly part of the data source itself, so they can be shared by applications and data sources across the enterprise. These components, which encapsulate interfaces that manage data access and data processing services (such as query processors, cursor engines, and transaction managers), enable an application to use only the database functionality it needs.

Comparing OLE DB with ODBC

The following table describes some of the primary differences between OLE DB and ODBC.

OLE DB	ODBC
Designed to access all types of data, both relational (SQL sources) and nonrelational (non-SQL sources).	Designed to access relational data only.
OLE DB is based on the Component Object Model (COM) architecture, which enables developers to create reusable and integrable service components that encapsulate various database services, such as cursor support and query processing.	Because ODBC is not based on the COM architecture (it is a Windows API), each ODBC driver must include all the functionality it needs to interact with the data source.

OLE DB and ODBC Comparison

Exploiting OLE DB Technology

Through common interfaces and reusable components, OLE DB enables businesses to integrate the applications, tools, and database products that are best suited for their purposes, regardless of vendor. IBM supports this effort through the development of UniOLEDB, which enables applications in OLE DB–based enterprise networks to access UniData and UniVerse data.

The following diagram shows an example of a data management application that uses Microsoft's ActiveX Data Objects (ADO), OLE DB interfaces, and other OLE DB–compliant interface software to access data simultaneously from a variety of sources, including UniData and UniVerse databases:
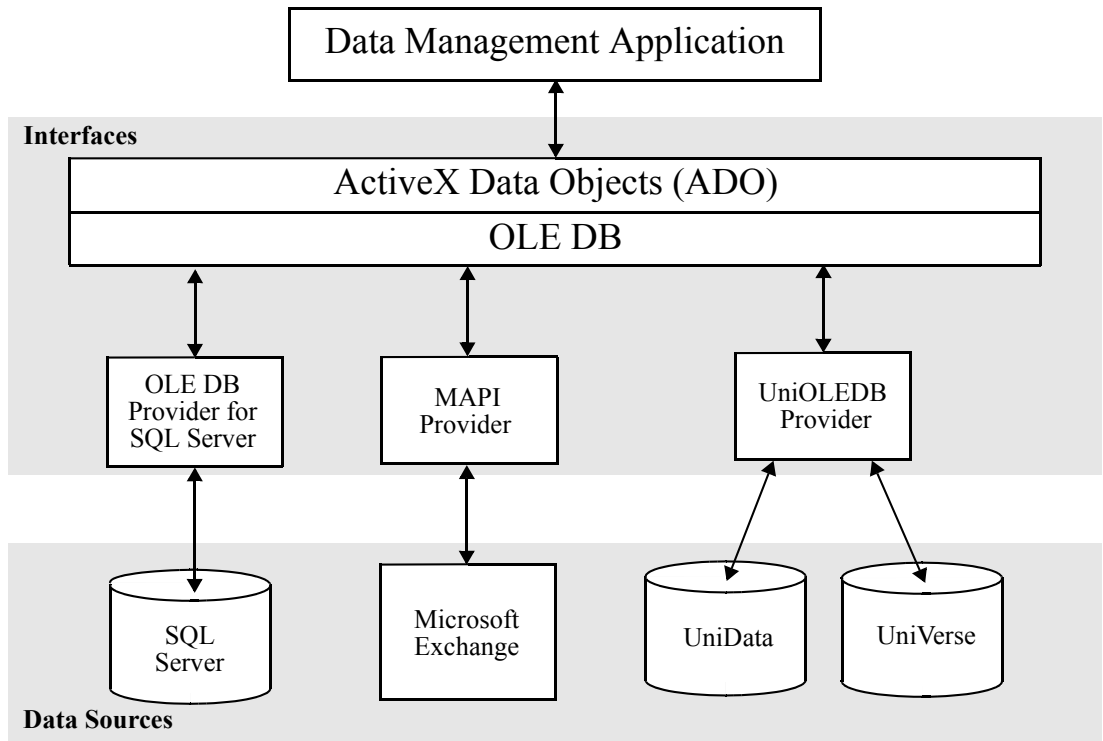
In this example, ADO functions as an application-level interface to OLE DB. ADO is not dependent on any particular programming tools or languages. OLE DB provides the underlying, system-level interfaces for data access. The ADO interface is optional. The data management application could use a different application-level interface, such as an SQL Server query processor, to access data. It also could use the OLE DB interfaces directly.

OLE DB Architecture

The general OLE DB architecture consists of logical components that represent functional aspects of traditional databases. The basic components include data consumers, data providers, and service components, as the following diagram shows:

```
                    ┌─────────────┐
                    │  Consumer   │
                    └─────────────┘
                            │
                         OLE DB
                            │
                    ┌─────────────┐
                    │   Service   │
                    │  Component  │
                    └─────────────┘
                            │
        ┌──────────── OLE DB ────────────┐
        │               │                │
  ┌───────────┐   ┌───────────┐    ┌───────────┐
  │ Provider  │   │ Provider  │    │ Provider  │
  └───────────┘   └───────────┘    └───────────┘
        │               │                │
    ┌───────┐       ┌───────┐        ┌───────┐
    │ Data  │       │ Data  │        │ Data  │
    │Source │       │Source │        │Source │
    └───────┘       └───────┘        └───────┘
```
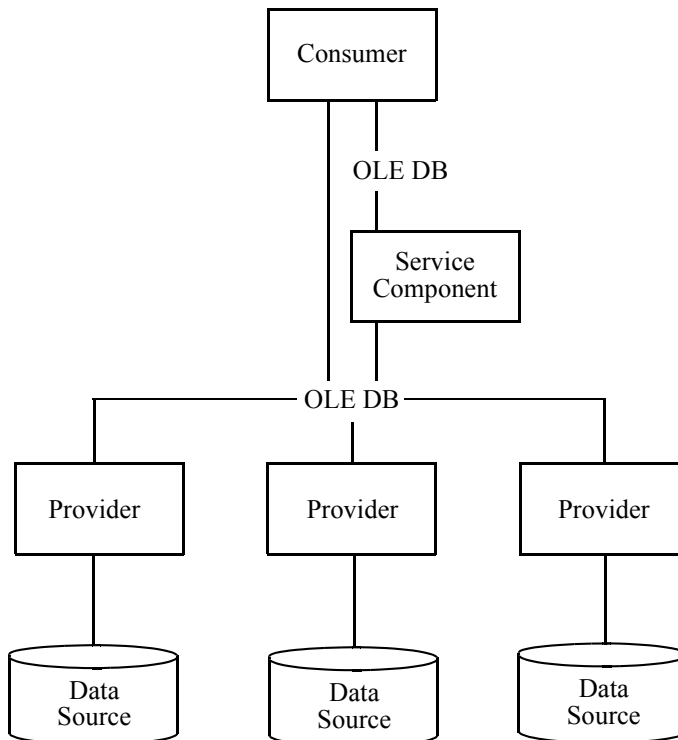
OLE DB provides a common set of object-oriented interfaces, which is the "wiring" the various OLE DB components need to work seamlessly together. These cooperating components produce and consume data that is exposed from data sources in the form of rowsets. The rowset interface guarantees that data can be used across the enterprise in a consistent and uniform way, regardless of its original source.

Consumers

An OLE DB consumer is an application that calls OLE DB interfaces to retrieve and manipulate data. The consumer receives and sends data in the form of a standard OLE DB rowset.

Examples of consumers include data management applications, such as Microsoft SQL Server and Microsoft Access.

Providers

An OLE DB provider is a software component that uses OLE DB interfaces to expose data to which consumers request access. The provider formats the exposed data in a standard tabular form (the rowset) that all OLE DB components can recognize. UniOLEDB serves as a provider.

Service Components

An OLE DB service component provides extended functionality the provider does not support, such as advanced cursor and query processing features. Service components lay on top of providers, and process and transport OLE DB data that come to them. The Service Component Manager, which is a core component of UDA products, calls interfaces in service components when a consumer requests OLE DB functionality that the provider does not support.

Examples of service components include query processors, cursor engines, and transaction managers.

OLE DB COM Objects

OLE DB consists of a comprehensive set of data access interfaces that are designed according to a fundamental component architecture. This architecture specifies a set of COM objects that enable application programs to access and manipulate data stored in diverse data sources. Each object encapsulates a set of interfaces that define its functions. Each interface includes one or more methods, each of which performs a specific task.

The following diagram shows a generic OLE DB COM object and the interfaces and methods it encapsulates:

OLE DB COM Object

Method 1 | Interface 1 | Method 3
Method 2 |

Method 1 | Interface 2 | Method 2
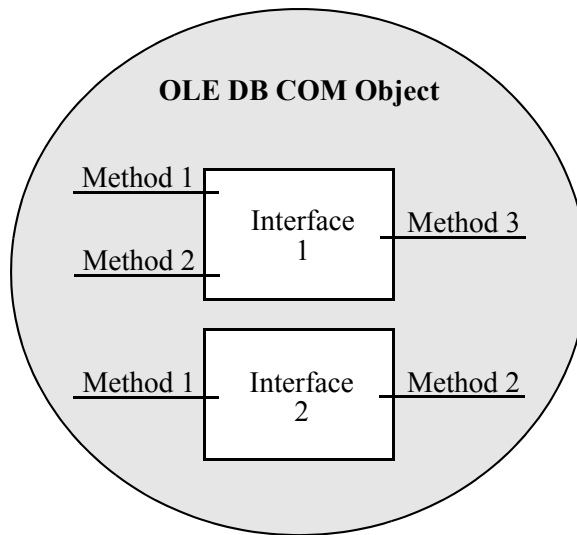
Object Example

The rowset object is the most fundamental OLE DB object. It is made up of rows and columns that contain data. In its basic form, a rowset object encapsulates the following interface functions:

- The IAccessor interface provides methods for managing accessors on a rowset, including methods for adding a reference count to an existing accessor, creating an accessor from a set of bindings, and releasing an accessor. An accessor describes how the consumer stores data in its buffer. The provider uses the accessor information to transfer data to and from this consumer buffer.

- The IColumnsInfo interface exposes information about rowset columns, such as column metadata (structural information, such as the type and length of data in the column) and ordinals of columns.

- The IConvertType interface exposes information about the types of data conversions a rowset supports.

- The IRowset interface provides methods for managing rowsets, including those for adding a reference count to an existing row handle, fetching rows sequentially, getting the data from those rows, and releasing rows.

- The IRowsetInfo interface returns information about the properties a rowset supports (including current property settings), obtains an interface pointer to the command session or IOpenRowset interface that created the rowset, or obtains an interface pointer to the rowset to which a bookmark applies.

Every rowset object must support these basic capabilities. Other rowset interfaces support other capabilities, such as inserting, updating, and deleting rows from a rowset.

Object Descriptions

The rowset is only one of several COM objects that define OLE DB. The following table describes the various COM objects that consumers, providers, and service components can use. For detailed descriptions of COM objects, see the *Microsoft OLE DB 2.0 Programmer's Reference and Software Development Kit* manual.

OLE DB components can support all or a subset of the COM objects described in the following table. At a minimum, an OLE DB provider must support data source, session, and rowset objects. Beyond the minimum, UniOLEDB also supports command and error objects.

Object	Description
Data Source	Provides a way for a consumer to connect to a specific data source. The data source object is distinct from the data source, which is the data the consumer wants to access, such as the data in a file or database.
	A data source object encapsulates interfaces and methods that identify a specific provider and establish a connection to the provider's underlying data source.
	An instance of a data source object is created when the consumer calls the CoCreateInstance interface on the class ID assigned to the provider.
Session	Defines the scope and context of transactions that can be transacted implicitly or explicitly.
	Consumers can use session objects to generate rowsets from the data source. Session objects also generate command objects and, if supported, transaction objects.
	In UniOLEDB, each session consumes a database connection.
Rowset	Contains information in the form of a set of rows, each of which has columns of data. The rowset is the central object that is produced, shared, and consumed by OLE DB components. It enables any data source, regardless of its type, to expose its data in a standard format that any OLE DB component recognizes.
	A provider can generate rowsets in either of the following ways:
	By using the ICommand interface, a consumer can send a query, which is a text command (such as an SQL statement), to a provider that supports query processing.

COM Objects

Object	Description
	A consumer can call the IOpenRowset::OpenRowset interface, which enables the provider that does not support the ICommand interface to expose data in the form of rowsets.
	Service components also present data, such as query results, as rowsets.
	Other OLE DB methods, if supported, return results in the form of the following specialized types of rowsets:
	Schema Rowset – Contains structural information (metadata) about a DBMS.
	Index Rowset – Contains index information, which provides access to DBMS data in a sequence of rows within a range of key values. Some service components that perform query processing implement this type of rowset.
	View Object – Defines a subset of the rows and columns in a rowset. View objects do not contain their own data.
	Session or command objects generate rowsets.
	UniOLEDB supports schema rowsets for UniData and UniVerse, but does not support index rowsets or view objects.
Command	Contains an SQL or CALL command.
	The command object encapsulates interfaces and methods that enable a provider to form, prepare, and execute data definition language (DDL) or data manipulation language (DML) commands (such as queries) that a consumer invokes. It also includes command properties that control how the command executes and determine how the resulting rowset behaves. The command object represents the basic query processing services that a typical data base management system (DBMS) provides.
	A session object can generate one or more command objects. Commands generate rowsets.

COM Objects

Object	Description
Error	Contains extended error information beyond the return codes and status information an OLE DB interface method typically provides, such as error descriptions and the error's SQL state (SQLSTATE). OLE DB error objects are extensions of OLE Automation error objects.
Enumerator	Searches for and produces a descriptive list of all data source objects and other enumerators that the enumerator recognizes. Through an enumerator, a provider can expose all of the data source objects that it potentially can access. From this list, a consumer can choose the appropriate data source object to use to connect to the target data source.
	To create an enumerator, the consumer calls the CoCreateInstance interface on the class ID assigned to the enumerator.
	Microsoft provides a root enumerator to all OLE DB consumers, which lists all available data providers and other enumerators that are listed in the client's registry.
	UniOLEDB does not support enumerator objects.
Transaction	Provides transaction functionality.
	Session objects generate transaction objects.
	UniOLEDB does not support transaction objects, but it does support transaction processing as described in Chapter 5, "UniOLEDB Functionality."

COM Objects (Continued)

Object Interactions

The following diagram shows how the COM objects could be used:

```
                          ┌──────────────┐
                          │  Enumerator  │
                          └──────┬───────┘
                                 │
                                 ▼
                        ┌────────────────────┐
                        │ Data Source Object │
                        └──┬──────────────┬──┘
                           │              │
  ┌───────┐                ▼              ▼
  │ Error │        ┌──────────────┐  ┌──────────────┐              ┌───────┐
  └───▲───┘        │  Session 1   │  │  Session 2   │              │ Error │
      │            └──────────────┘  └──────────────┘              └───▲───┘
  ┌───┴────┐                                                           │
  │ Rowset │    ┌──────────┐  ┌──────────┐  ┌──────────┐   ┌─────────────┐
  └────────┘    │ Command  │  │ Command  │  │  Rowset  │   │ Transaction │
  ┌────────┐    └──────────┘  └──────────┘  └──────────┘   └─────────────┘
  │ Rowset │    ┌──────────┐
  └────────┘    │  Rowset  │
  ┌────────┐    └──────────┘
  │ Rowset │    ┌──────────┐
  └────────┘    │  Rowset  │
                └──────────┘
```
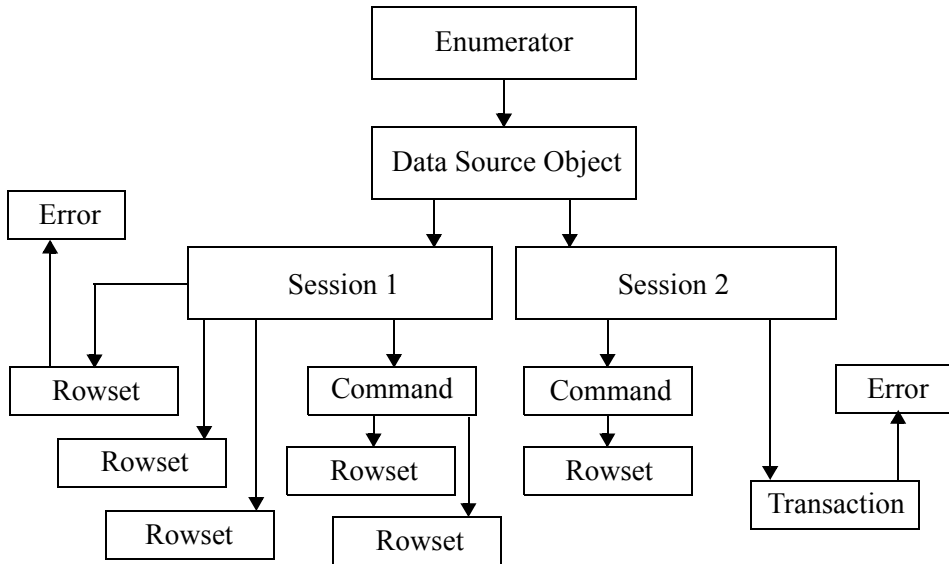
This example illustrates the standard relationships between the various objects. The enumerator identifies a data source object for a data source. The data source object generates two sessions, each representing a connection to the data source.

From the first session, three rowsets are created by using the IOpenRowset interface directly from the session (one rowset generates an error object), and two additional rowsets are created by a text command that is enclosed in a command object. The error object contains error information beyond the return codes and status information COM objects generally provide.

From the second session, a rowset is created by using an enclosed text command, and a transaction object is created that generates an error object.

Overview of UniOLEDB

OLE DB technology enables consumers to access and manipulate data from UniData and UniVerse, which are extended relational databases that can store multiple values in the column of a row. Because UniOLEDB expects data to be organized relationally in first normal form (1NF), data in UniData and UniVerse must be presented in a standard non-extended table format (normalized).

Although OLE DB 2.1 supports chaptered rowsets, UniOLEDB does not support them at this time.

UniVerse dynamically normalizes its own data. For more information, see Chapter 4, "Accessing UniVerse Data."

UniData also normalizes its own data, but you must prepare the data for it to do so. To prepare the data, you must use VSG, a GUI tool that generates schema on UniData files, or the Schema API. For more information about using VSG or the Schema API, see *Using VSG and the Schema API*.

UniOLEDB Architecture

The following diagram shows a high-level view of a consumer application accessing a data source through UniOLEDB:
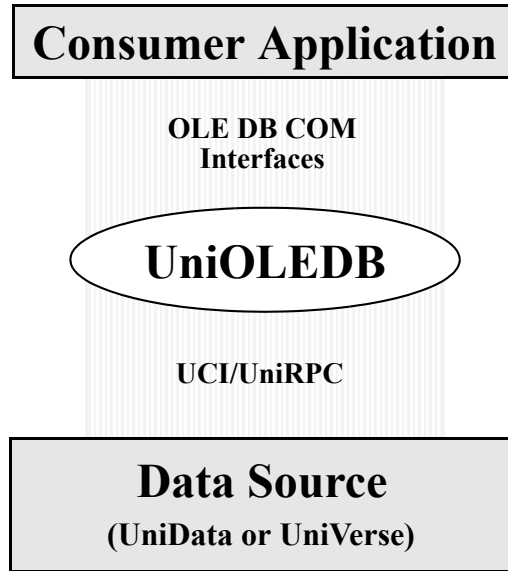
UniOLEDB exposes the data in a UniData or UniVerse data store. It uses UCI, which is an application programming interface (API) that UniOLEDB uses to access data in UniData and UniVerse databases. UCI uses UniRPC, a library of calls developed by IBM, to implement remote procedure calls. UniOLEDB passes the results back to the consumer application in the form of a rowset.

The following diagram shows a client accessing data from either a UniData or
UniVerse server. It reveals in greater detail how UniOLEDB fits in with other compo-
nents in a client/server configuration. UniOLEDB also can access UniData and
UniVerse servers concurrently:

Client

Consumer Application

OLE DB COM

OLE DB Service Component

OLE DB COM

UniOLEDB
UCI/UniRPC

udserver		uvserver
:---:		:---:
UniData		UniVerse

Server 1   **Server 2**
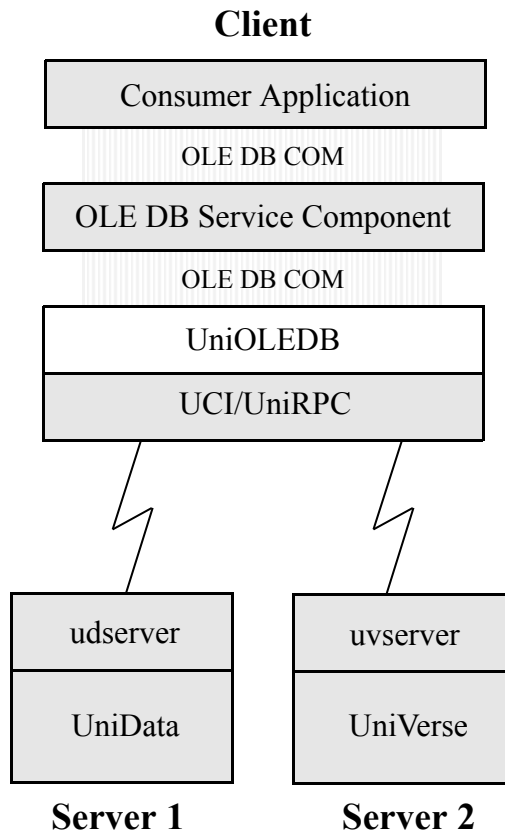
The following table describes each component in the diagram.

Component	Description
Consumer Application	An OLE DB–compliant application that accesses and manipulates OLE DB–accessible data sources. A UniOLEDB consumer application can access either UniData or UniVerse data, or both concurrently.
OLE DB COM	A set of object-oriented interfaces that are based on the Component Object Model (COM). It consists of objects that encapsulate various aspects of traditional database functionality. OLE DB COM enables OLE DB components (consumers, providers, and service components) to interoperate and produce, share, and consume data in the form of rowsets.
OLE DB Service Component	An OLE DB component that encapsulates a service that creates and consumes data through OLE DB interfaces. Third-party OLE DB service components are consumers and providers of data. They lay on top of UniOLEDB and provide enhanced and specialized functionality. For example, the Microsoft Cursor Service component provides scrollable cursor functionality for UniOLEDB.
UniOLEDB	The OLE DB provider, which exposes UniData and UniVerse data to consumer applications.
UCI/UniRPC	UCI is an application programming interface (API) that UniOLEDB uses to access data in UniData and UniVerse databases. UCI uses UniRPC, a library of calls developed by IBM, to implement remote procedure calls.
udserver	A UniData host-based server that interprets and processes data requests from UniOLEDB by using UCI. The udserver service can handle multiple consumer requests efficiently.
UniData	An extended relational database.
uvserver	A UniVerse host-based server that interprets and processes data requests from UniOLEDB by using UCI. The uvserver service can handle multiple consumer requests efficiently.
UniVerse	An extended relational database.

Example Enterprise Network Components

Supported OLE DB Functionality

UniOLEDB supports minimum and base provider functionality, as defined in Microsoft OLE DB provider leveling specifications, and several extended OLE DB interfaces. For detailed information about UniOLEDB functionality, including supported OLE DB interfaces and descriptions of their properties and behaviors, see Chapter 5, "UniOLEDB Functionality."

Before You Use UniOLEDB

Before consumers can access UniData or UniVerse through UniOLEDB, you must:

- Meet the hardware and software requirements for the client and server machines as provided under "Hardware and Software Requirements" on page 1-21.

- Install UniOLEDB on the client machine from the appropriate CD-ROM by using the standard Microsoft Windows installation procedure.

- Set up the UCI configuration file (uci.config) on the client machine. For information, see Chapter 2, "Setting Up UniOLEDB."

- Make UniData or UniVerse accounts and their data accessible to OLE DB consumers. For information, see Chapter 3, "Accessing UniData Data," for UniData, or Chapter 4, "Accessing UniVerse Data," for UniVerse.

Hardware and Software Requirements

The following lists provide the hardware and software requirements for client and server machines that use UniOLEDB.

Client Requirements

- IBM-compatible personal computer (PC) attached to a network.
- Either of the following operating systems:
    - Microsoft Windows 95 or later.
    - Microsoft Windows NT 4.0 (Service Pack #3) or later.
- Approximately 21 MB of free disk space.
- A minimum of 32 MB of random access memory (RAM).
- TCP/IP.

Server Requirements

- UNIX, Windows platform.
- TCP/IP.

- Either of the following IBM DBMSs:
  - UniData 5.1 or later (to determine platform availability of UniData 5.1, contact your IBM account manager).
  - UniVerse 9.5.1D or later.

Setting Up UniOLEDB

This chapter describes how to prepare a client machine to use UniOLEDB.

Install UniOLEDB from the appropriate CD-ROM by using the standard Microsoft Windows installation procedure.

Setting Up the UCI Configuration File

An OLE DB consumer connects to UniOLEDB, which establishes connections to remote data sources and sends database requests to one or more UniData or UniVerse servers. Consumers access data sources that are mapped to UniData or UniVerse accounts through entries in the UCI configuration file (uci.config) on the client machine. This file contains connection parameters needed to route requests to the appropriate UniData or UniVerse server.

When a consumer attempts to connect to a data source, UniOLEDB reads the UCI configuration file to determine the host system, DBMS type, server name, and other optional information.

The UCI configuration file that UniOLEDB uses is specified in the UciCfgFile key in the system registry under \HKEY_LOCAL_MACHINE\SOFTWARE\IBM\UCI.

A consumer can access UniData or UniVerse databases residing on various operating systems. Each configuration entry describes the physical attributes of a database in sufficient detail to perform three tasks:

- Establish communications.
- Start a UniData or UniVerse server process.
- Route query and update requests.

In uci.config on the client machine, you must define the UCI data sources to which you want OLE DB consumers to connect. You can define the UCI data sources in either of the following ways:

- Use any text editor to modify the file.
- Use the UCI Config Editor to manage data sources. For information about defining data sources with the UCI Config Editor, and for information about parameters in the UCI configuration file, see the *Administrative Supplement for Common APIs*.

The default configuration file shipped with UniData or UniVerse looks like this:

```
[ODBC DATA SOURCES]
<localuv>
DBMSTYPE = UNIVERSE
NETWORK = TCP/IP
SERVICE = uvserver
HOST = localhost

<localud>
DBMSTYPE = UNIDATA
NETWORK = TCP/IP
SERVICE = udserver
HOST = localhost
```

This default configuration file enables you to access a UniVerse or UniData database on the same hardware platform as the one on which your application is running.

You can add as many of these entries as you want, each with a different data source name.

To access a remote database on a different platform, you must add an entry to the configuration file. For example, if the remote system you want to access is named hq1 and the account path is /usr/myacct, make up a data source name such as corp1 and add the data source definition to uci.config as follows:

```
<corp1>
DBMSTYPE = UNIDATA
NETWORK = TCP/IP
SERVICE = udserver
HOST = hq1
ACCOUNT = /usr/myacct
USERNAME = myloginname
```

The ACCOUNT parameter can be set to any one of the following:

- The full path to a UniData or UniVerse account.
- A valid UniVerse schema name.
- A valid UniData database name.

A UniData database name is valid if it appears as an entry in the ud_database file. For UNIX systems, this file is located in the /usr/ud*nn*/include path, where *nn* is the release of UniData you are running. For Windows platforms, it is located in *udthome*\include. For examples of database entries in the ud_database file, see Chapter 3, "Accessing UniData Data."

Note: If you modify uci.config by using a text editor, make sure you surround the equal signs with spaces.

Accessing UniData Data

This chapter describes how to access data in UniData tables and files. You must perform the following tasks:

- Verify that the UniRPC daemon (for UNIX systems) or the UniRPC service (for Windows platforms) is running.
- Make UniData accounts accessible to consumers.
- Present the data in UniData in a format that is accessible to consumers.

Verifying That UniRPC Is Running

When UniOLEDB requests a connection to the UniData UCI server, the UniRPC facility starts the server (udserver) process, which runs on the machine where the database resides. Before you use UniOLEDB, make sure that UniRPC is running.

On UNIX systems, you can verify whether UniRPC is running by entering the following command:

    ps -ef | grep unirpc

If you need to start UniRPC for UNIX, use the UniData startud command.

On Windows platforms, you can verify whether UniRPC is running by double-clicking the Services icon on the Control Panel. The Services dialog box appears. If UniRPC is running, it appears in the list of services with the status "Started." If UniRPC is not running, you can start it from the Services dialog box.

For both UNIX and Windows platforms, UniRPC is installed automatically when you install UniData.

On UNIX systems, the UniRPC services file (unirpcservices) contains an entry that is similar to the following:

    udserver /usr/ud71/bin/udsrvd * TCP/IP 0 3600

The unirpcservices file is located, by default, on the UCI server in the .../unishared/unirpc directory. This directory is located in a path that is parallel with the *udthome* directory. For example, if the path to *udthome* is /disk1/*udthome*, the path to the unirpcservices file is /disk1/unishared/unirpc.

On Windows platforms, the unirpcservices file is located on the UCI server in the following default path:

    x:\IBM\unishared\unirpc

where *x* is the drive on which the software is installed. It contains an entry that is similar to the following:

    udserver C:\usr\ud71\bin\udsrvd.exe * TCP/IP 0 3600

Your unishared directory might not be located in the default path. To determine its actual location on UNIX systems, enter:

    cat /.unishared

This provides the path for the unishared directory.

For Windows platforms, you can find the path for unishared by looking in the system registry under \HKEY_LOCAL_MACHINE\SOFTWARE\IBM\unishared.

When the client system requests a connection to a UCI server, the local UniRPC daemon or service uses the unirpcservices file to verify that the client can start the requested server (in this case, udserver).

With the Server edition of UniData, each UniOLEDB connection to a UniData UCI server consumes one UniData license. With the Workstation, Workgroup, or Enterprise edition of UniData, device licensing enables each client system to establish up to ten connections to the UCI server from one device while consuming only one database license — regardless of whether more than one user logon is used to make the connections. For more information about device licensing, see *Installing and Licensing UniData Products* and the *Administrative Supplement for Common APIs*.

UCI Connection Timeout Configuration

The sixty-minute default inactivity timeout value (3600) for UCI connections can be too short. If users leave UniOLEDB connections open, but the connections remain inactive for longer than sixty minutes, they could receive UniRPC error code 81015. To increase this timeout value, use any text editor to modify the unirpcservices file (for the path to this file, see "Verifying That UniRPC Is Running" on page 3-3).

Note: To edit the unirpcservices file, you must have root privileges on UNIX or Administrator privileges on Windows platforms.

For example, to set the connection timeout delay to 24 hours, in the line starting with udserver, change the right-most number to 864000 (the number of seconds in 24 hours). The line could appear as follows:

    udserver /usr/ud72/bin/udsrvd * TCP/IP 0 864000

Making UniData Accounts Accessible

UniData databases are organized into accounts. A consumer connects to a UniData account and can access the files there. You optionally can define the account as a database in the ud_database file on the server machine. You also can include the account path or database name in the UCI data source definition in the UCI configuration file (uci.config). For information about setting up the UCI configuration file, see Chapter 2, "Setting Up UniOLEDB."

You also can specify the account path or database name each time you attempt to connect to the account. In this case, you would not need to include the account path or database name in the UCI configuration file. When you attempt to connect, you are prompted to specify the full path to the account or the database name.

If you want to access an account that has a UDTHOME directory different than the default UDTHOME directory, you must include a definition for that account in the ud_database file on the server machine. For UNIX systems, this file is located in the /usr/ud*nn*/include path, where *nn* is the release of UniData you are running. For Windows platforms, it is located in *udthome*\include. You can find the path for *udthome* by looking in the system registry under \HKEY_LOCAL_MACHINE\SOFTWARE\IBM\UniData\7.1. Use any text editor to modify the ud_database file.

Note: To determine your default UniData home directory, use the UNIX env command. The results of this command include the default setting for the UDTHOME environment variable.

The following Windows example shows an entry in the ud_database file for a database named db2:

```
DATABASE=db2
UDTHOME=d:\disk2\test71
UDTACCT=d:\disk2\test71\testacct.
```

In the ud_database file entry, the UDTHOME parameter is optional. You should include it only when the UDTHOME directory is different than the default UDTHOME directory.

Tracing Events

By using the tracing feature, you can create logs of events between clients and the database through the server. Logs enable IBM support personnel to help troubleshoot problems. You can define trace levels for database entries in the ud_database file.

The following table describes the valid trace levels and the associated information that is written to the trace log.

Trace Level	Description
0	Includes all fatal error information.
1	Includes all UCI commands in addition to the information provided by trace level 0.
2	Includes parameter information and column descriptions in addition to the information provided by trace levels 0 and 1.
3	Includes data values in addition to the information provided by trace levels 0, 1, and 2.

Trace Levels

The trace log is named udsrv_*database.processID* where *database* is the the database name (for example, db2) and *processID* is the process number ID. By default, it is located in your temporary directory (typically, /tmp for UNIX; *x*:\temp for Windows platforms). For UNIX, you can find the location of the trace log file in the TMP parameter in the udtconfig file under /usr/ud61/include.

The following UNIX example shows a tracing level setting for a database named dbase3:

```
DATABASE=dbase3
UDTHOME=/disk1/ud71
UDTACCT=/home/bobm/bmtest
TRACE_LEVEL=3
```

Presenting Data in OLE DB–Accessible Format

Data in UniData is organized differently from the way UniOLEDB expects it to be organized. Two areas in which the data differs from what UniOLEDB expects are:

- Data types
- Multivalued and multi-subvalued data

Although OLE DB 2.1 supports chaptered rowsets, UniOLEDB does not support them at this time.

Data Types

UniData does not define data types for data contained in its files. On the other hand, UniOLEDB expects data types for all data. In addition, data in UniData can be of variable length, but UniOLEDB expects data to have either a fixed or a maximum length. To make the data look more like what UniOLEDB expects, you must use VSG or Schema API.

Multivalued and Multi-Subvalued Data

UniOLEDB expects data to be organized in first normal form (1NF) format. Although some files could be in 1NF format, which means that only one value is stored in each column of each row, many UniData files have columns that store multiple values in the columns of a row (NF^2 format). To instruct UniData SQL to present data in 1NF format, you must use VSG or Schema API.

VSG or Schema API creates views and subtables that present multivalued and multi-subvalued data in 1NF format. These views and subtables do not duplicate data, but merely instruct UniData SQL to normalize data before returning it to the application. UniOLEDB passes the 1NF data to consumers in the form of rowsets.

Missing Values

To determine how UniData treats missing values for character-type data, set the VCHAREMPTY variable in the uci.config file.

If VCHAREMPTY=ON, UniData does not convert empty (missing) values to NULL. If VCHAREMPTY=OFF, UniData converts empty (missing) values to NULL.

Visual Schema Generator and the Schema API

There are two ways to create 1NF views and subtables to instruct UniData SQL to present data in UniData in an OLE DB–accessible format:

- VSG
- Schema API

VSG is a Windows-based graphical user interface (GUI) tool that makes UniData files accessible to consumers through UniOLEDB. It enables you to create, delete, and modify UniData SQL subtables and views. VSG also enables you to set ANSI privileges (security) for tables, subtables, and views.

VSG guides data administrators through the process of defining the 1NF subtables and views that represent the extended relations by visually presenting all available configuration options. It also translates conversion specifications for UniData to SQL data types. VSG performs logical error checking through every step of the schema generation process.

The Schema API consists of a series of UniBasic subroutines designed to accomplish the same normalization tasks as VSG.

IBM recommends that you use VSG to prepare files because it is easier and faster to use than the Schema API. It performs several processes automatically. You might want to use the Schema API if you have a large number of files (in this case, several VSG processes take several minutes to complete) or you deploy data files with your applications.

For more information about using VSG or the Schema API, see the *Using VSG and the Schema API* manual.

Accessing UniVerse Data

This chapter describes how to access data in UniVerse tables and files. The following data is always accessible to a UniOLEDB client connected to a UniVerse account:

- All UniVerse tables on the system.
- All non-SQL UniVerse files defined in the VOC file of the current account.

In this chapter, the word *tables* refers to UniVerse tables defined by the CREATE TABLE statement and views defined by the CREATE VIEW statement. The word *files* refers to non-SQL UniVerse files that are defined by the CREATE.FILE command. The term *UniOLEDB table* refers to any table or file, real or virtual, that is accessible to UniOLEDB.

Before a consumer attempts to connect to a UniVerse data source, the UniRPC daemon (for UNIX systems) or the UniRPC service (for Windows platforms) must be running on the server system.

The UniVerse system must be defined as a data source on the client machine in the UCI configuration file. For information about setting up data sources, see Chapter 2, "Setting Up UniOLEDB"

Accessing UniVerse Tables and Files

An OLE DB consumer connects to a UniVerse account (which can be a schema) and can access all tables on the system and all files defined in the VOC file of that account.

Sometimes an OLE DB consumer connected to UniVerse gets a list of UniOLEDB tables by requesting a TABLES rowset. This rowset lists:

- All tables on the system.
- Files defined in the current account's VOC file that have been made visible to OLE DB.
- Virtual tables (dynamically normalized multivalued data) within the above.

We will refer to the UniOLEDB tables listed in the TABLES rowset as "visible" to OLE DB consumers. Although all SQL tables on the system are by definition visible, non-SQL files that have not been made visible are not included in the TABLES rowset. For information about how to make files visible in an account, see "Making Files Visible to UniOLEDB Consumers" on page 4-12.

Tables

When connected to a UniVerse account, an OLE DB consumer can access all tables on the system. If connected to a UniVerse schema, the consumer can reference tables in that schema by using unqualified table names in SQL statements. All other tables on the system can be referenced using table names qualified by the appropriate schema name.

Example 1

The consumer is connected to schema HR. If table EMPLOYEES is in schema HR, it can be referenced simply as EMPLOYEES. If view EMPLOYEES is in schema DENVER, it must be referenced as DENVER.EMPLOYEES.

Example 2

The consumer is connected to account SALES that is not a schema. In this case, table EMPLOYEES (in schema HR) and view EMPLOYEES (in schema DENVER) must be referenced as HR.EMPLOYEES and DENVER.EMPLOYEES respectively.

Files

An OLE DB consumer can access files that are defined in the VOC file of the account to which the consumer is connected. Because some OLE DB applications are written to access only tables listed in the TABLES rowset, as a practical matter UniVerse files may not be usable unless they have been made visible for that account.

For example, if a consumer connects to an account whose files have been made visible, all files defined in the account's VOC file, except for system files (such as &SAVEDLISTS&) and UV/Net files, are visible. The consumer can reference the files using the file names in the VOC. The account does not need to be a schema.

On the other hand, if a consumer connects to an account whose files have not been made visible, no files appear in the TABLES rowset. However, the consumer can still access files defined in the account's VOC file if the consumer is programmed to bypass the contents of the TABLES rowset.

Multivalued Data

UniOLEDB expects data to be organized relationally in first normal form (1NF). Although some UniVerse tables and files are in first normal form, which means only one value is stored in each column of each row, many UniVerse tables and files have columns that store multiple values.

UniVerse always presents multivalued data to UniOLEDB in first normal form (1NF), and UniOLEDB passes it to consumers in the form of rowsets. UniVerse automatically normalizes its tables and files by a process called "dynamic normalization." This means that a file containing multivalued data appears to UniOLEDB as several 1NF tables, each consisting of singlevalued data only.

Example

The table ORDERS is defined with columns ORDERNUM, CUSTNUM, DATE, PART, and QTY. PART and QTY are multivalued and make up an association called ITEMS. The association key is PART. Suppose this table contains the following orders.

ORDERNUM	CUSTNUM	DATE	PART	QTY
99101	12345	5/25/99	HINGE	200
			BOLT	650
99102	98765	6/10/99	BOLT	50

ORDERS Table

This file appears to OLE DB as two 1NF tables called ORDERS and ORDERS_ITEMS. The ORDERNUM column is the ORDERS key.

ORDERNUM	CUSTNUM	DATE
99101	12345	5/25/99
99102	98765	6/10/99

ORDERS 1NF Table

The ORDERNUM and PART columns are the two columns that make up the ORDERS_ITEMS key.

ORDERNUM	PART	QTY
99101	HINGE	200
99101	BOLT	650
99102	BOLT	50

ORDERS 1NF Table

The TABLES Rowset

An OLE DB consumer can request a TABLES rowset that includes all tables and visible files. For each table and visible file, this rowset includes schema name, table name, and table type. Table type can be TABLE, VIEW, or SYSTEM TABLE.

Example 1

Suppose the consumer is connected to a UniVerse account that is not a schema, and suppose the account contains two files called EMPS and DEPTS. EMPS and DEPTS are not tables. EMPS has one multivalued column called DEPENDENTS, and DEPTS has no multivalued columns. Files in this account have been made visible.

Suppose there is another UniVerse account whose files have been made visible, which contains a file called OTHERFILE.

If the consumer requests the TABLES rowset, it includes the following information.

Schema Name	Table Name	Table Type
<null value>	EMPS	TABLE
<null value>	EMPS_DEPENDENTS	TABLE
<null value>	DEPTS	TABLE

Example 1: TABLES Rowset Information

This shows that:

- Any files listed in the TABLES rowset must be in the account to which the consumer is connected (OTHERFILE does not show up).

- Files listed in the TABLES rowset have a null value for the schema name.

- A multivalued column appears as a separate dynamically-normalized table whose type is TABLE and whose name is composed of the original file's name followed by an underscore and the name of the column.

Example 2

Suppose the above account is made into a schema named USA. Suppose two tables called CITIES and STATES are then created where CITIES has only singlevalued columns and STATES has an association called COUNTIES.

The TABLES rowset now includes the following returned information.

Schema Name	Table Name	Table Type
<null value>	EMPS	TABLE
<null value>	EMPS_DEPENDENTS	TABLE
<null value>	DEPTS	TABLE
USA	CITIES	TABLE
USA	STATES	TABLE
USA	STATES_COUNTIES	TABLE

Example 2: TABLES Rowset Information

This shows that:

■ Tables have a schema name, unlike files.

■ An association appears as a dynamically-normalized table whose type is TABLE and whose name is composed of the original file's name followed by an underscore and the name of the association.

Example 3

Suppose an SQL view called STATEVIEW is created in another schema (called JOESCHEMA) with the following SQL statement:

```
CREATE VIEW STATEVIEW AS SELECT * FROM USA.STATES;
```

While still connected to the original account, the consumer sees the following information returned in the TABLES rowset.

Schema Name	Table Name	Table Type
<null value>	EMPS	TABLE
<null value>	EMPS_DEPENDENTS	TABLE
<null value>	DEPTS	TABLE
USA	CITIES	TABLE

Example 3: TABLES Rowset Information

Schema Name	Table Name	Table Type
USA	STATES	TABLE
USA	STATES_COUNTIES	TABLE
JOESCHEMA	STATEVIEW	VIEW
JOESCHEMA	STATEVIEW_COUNTIES	VIEW

Example 3: TABLES Rowset Information (Continued)

This shows that:

- Tables and views in other schemas appear in the TABLES rowset.
- Views and their dynamically-normalized associations and multivalued columns have the table type VIEW.

Example 4

Suppose the account JOESCHEMA mentioned above contains a UniVerse file called JOEFILE that has not been made visible. Suppose an OLE DB consumer connects to JOESCHEMA and requests the TABLES rowset. This rowset includes the following information.

Schema Name	Table Name	Table Type
USA	CITIES	TABLE
USA	STATES	TABLE
USA	STATES_COUNTIES	TABLE
JOESCHEMA	STATEVIEW	VIEW
JOESCHEMA	STATEVIEW_COUNTIES	VIEW

Example 4: TABLES Rowset Information

This shows:

- Files in other accounts (such as EMPS) do not appear in the TABLES rowset.
- Files in the account the consumer is connected to (such as JOEFILE) do not appear if they have not been made visible.

Example 5

The UniVerse SQL catalog tables also appear in the TABLES rowset. Their table type is SYSTEM TABLE.

Schema Name	Table Name	Table Type
CATALOG	UV_ASSOC	SYSTEM TABLE
CATALOG	UV_COLUMNS	SYSTEM TABLE
CATALOG	UV_COLUMNS_ACOL_NO	SYSTEM TABLE
CATALOG	UV_COLUMNS_AMC	SYSTEM TABLE
CATALOG	UV_SCHEMA	SYSTEM TABLE
CATALOG	UV_TABLES	SYSTEM TABLE

Example 5: TABLES Rowset Information

The UniVerse Server Administration Menu

You can use the UniVerse Server Administration menu on the server system to perform certain administrative functions that may facilitate access to your UniVerse data from UniOLEDB consumers. These functions include making UniVerse files in an account visible to UniOLEDB, and detecting data anomalies in a table or file. Tasks described later in this chapter use selections that appear on this menu.

To display the menu:

1. Log on to the server system as root on UNIX or as an Administrator on Windows platforms.

2. Invoke UniVerse and log to the HS.ADMIN account. This account is in a subdirectory named HS.ADMIN in the UV account directory.

3. Type the command: HS.ADMIN.

The UniVerse Server Administration menu appears.

```
UniVerse Server Administration


 1. List activated accounts
 2. Show UniVerse ODBC Config configuration for an account
 3. Activate access to files in an account
 4. Deactivate access to files in an account
 5. Run HS.SCRUB on a File/Table
 6. Update File Information Cache in an account


Which would you like? ( 1 - 6 )
```

1.   To exit the menu, press ENTER.

The following table describes the selections that appear on the UniVerse Server Administration menu.

Menu Selection	Description
1. List activated accounts	Lists UniVerse accounts whose files have been made visible to UniOLEDB. When a consumer is connected to such an account, the account's files are included in the TABLES rowset.
2. Show UniVerse ODBC Config configuration for an account	Does not apply to UniOLEDB.
3. Activate access to files in an account	Makes files in a UniVerse account visible to UniOLEDB so that when a consumer is connected to this account, its files are included in the TABLES rowset.
4. Deactivate access to files in an account	Removes UniOLEDB-visibility from files in a UniVerse account. This reverses the actions of making files visible.
5. Run HS.SCRUB on a File/Table	Analyzes the data in a table or file, and lets you optionally correct anomalous data.
6. Update File Information Cache in an account	Updates this account's .hs_fileinfo file to reflect the current state of all files in the account.

UniVerse Server Administration Menu Selections

Making Files Visible to UniOLEDB Consumers

There are several types of UniOLEDB consumers, including:

- Consumers programmed to deal with known tables and data on a specific UniVerse server.
- Consumers that let an interactive user specify what tables to access on the server, expecting the user to know the names of the tables to process.
- Consumers that use the TABLES rowset to determine what tables can be accessed. An example of this type of consumer might be a general-purpose tool that lists the TABLES rowset on users' screens and lets them choose one of the listed tables to process.

For consumers in the first two categories, there is no need to make files visible in UniVerse accounts on the server.

For consumers in the last category, it is necessary to make files visible unless all of the data of interest is stored in UniVerse tables. This is because files that are not tables are not included in the TABLES rowset unless the consumer is connected to an account whose files have been made visible.

Making Files Visible

To make files in a UniVerse account visible to UniOLEDB consumers:

1. On the UniVerse Server Administration menu, choose the third selection (Activate access to files in an account).
2. When prompted, enter either the full path of the UniVerse account (on Windows platforms, start with the drive letter, such as D:) or the account name as listed in the UV.ACCOUNT file.
3. Repeat steps 1 and 2 for each account whose files you want to make visible.
4. To exit the UniVerse Server Administration menu, press ENTER.

The process of making files visible does the following:

- Scans the dictionaries of all nonsystem files named in the VOC, finding all associations and unassociated multivalued columns.
- Writes an @EMPTY.NULL X-record in each file dictionary.

- Writes S or M in field 5 of A- and S-descriptors.
- Creates Q-pointers in the VOC for files comprising multiple data files.
- Creates an HS_FILE_ACCESS file in the account.
- Creates a file information cache (.hs_fileinfo) under the account's directory, which contains a compressed list of all 1NF file names in the account and is used by the UniOLEDB provider to rapidly construct the TABLES rowset whenever a consumer requests it to do so.
- Updates the UV.ACCOUNT file to indicate that the files in the account have been made visible.

Restricting File Visibility

When you make an account's files visible to UniOLEDB, the HS_FILE_ACCESS file is created in the account, as stated in the previous section. This file contains records for non-SQL files referenced by F- and Q-pointers in the account's VOC file. the following table shows the format and initial contents of the HS_FILE_ACCESS file.

FILENAME (Record ID)	ACCESS (Field 1)
HS_DEFAULT	READ_WRITE
&DEVICE&	NONE
.	.
.	.
.	.
VOC	NONE
VOCLIB	NONE

Initial Contents of HS_FILE_ACCESS

You can edit the HS_FILE_ACCESS file to control which files in the account are visible to UniOLEDB consumers.

The IDs of records in the HS_FILE_ACCESS file are the names of the files whose access you want to control. Each record has one field (ACCESS), which contains one of the following values:

- READ_WRITE
- READ

- NONE

If the ACCESS field for some file is set to READ, this is treated the same as READ_WRITE for UniOLEDB.

The HS_FILE_ACCESS file contains a special record called HS_DEFAULT that controls default access to all files in the account. When you first make files visible, the HS_DEFAULT record is set to READ_WRITE and the records for UniVerse system files (APP.PROGS, BASIC.HELP, ERRMSG, UV.ACCOUNT, DICT.DICT, NEWACC, and so forth) are set to NONE (no access).

To remove visibility from a few selected files and leave all the rest visible to UniOLEDB, add a record to HS_FILE_ACCESS for each restricted file, with the ACCESS field set to NONE.

To make just a few files in an account visible to UniOLEDB, change the ACCESS field in the HS_DEFAULT record from READ_WRITE to NONE, and then add selected files whose ACCESS field is READ_WRITE.

If you change the contents of the HS_FILE_ACCESS file, you must then update the file information cache in order for your changes to take effect. For information about updating the file information cache, see the next section.

Updating the File Information Cache

After an account's files have been made visible, if the following types of change are made to the account's files, the changes are not automatically reflected in the TABLES rowset:

- Adding, changing, or deleting F- or Q-pointers in the VOC file.
- Creating or deleting UniVerse files.
- Adding, changing, or deleting associations or unassociated multivalued column definitions in file dictionaries.
- Restricting access to selected files by changing the contents of the HS_FILE_ACCESS file.

These types of change are not reflected in the TABLES rowset until the account's file information cache is updated.

You can update the file information cache using the UniVerse Server Administration menu or the HS.UPDATE.FILEINFO command.

To update the file information cache using the menu:

1. On the UniVerse Server Administration menu, choose the sixth selection (Update File Information Cache in an account).

2. When prompted, enter either the full path of the UniVerse account (on Windows platforms, start with the drive letter, for example, D:) or enter the account name as listed in the UV.ACCOUNT file.

3. Repeat steps 1 and 2 for each account you want to update.

4. To exit the UniVerse Server Administration menu, press ENTER.

To update the file information cache using the HS.UPDATE.FILEINFO command:

1. Log to the desired UniVerse account.

2. At the system prompt, enter the command HS.UPDATE.FILEINFO.

    The process of updating an account's file information cache does the following:

    ■ Scans the dictionaries of all nonsystem files named in the VOC, finding all associations and unassociated multivalued columns.

    ■ Rewrites the file information cache (.hs_fileinfo) under the account's directory, based on the above dictionary information and on the contents of the HS_FILE_ACCESS file.

The file information cache is for internal use only and should not be modified. Any changes to the cache can cause unpredictable behavior and can make UniVerse files in the account inaccessible to consumers.

Example

Suppose account MYACCOUNT contains the following files:

■ STATES, which has an association called CITYZIPS.

■ EMPS, which has multivalued (unassociated) columns DEPENDENTS and PHONES.

■ OCEANS, which has only singlevalued columns.

After the files in this account are made visible, the TABLES rowset lists the following tables.

Schema Name	Table Name	Table Type
<null value>	STATES	TABLE
<null value>	STATES_CITYZIPS	TABLE
<null value>	EMPS	TABLE
<null value>	EMPS_DEPENDENTS	TABLE
<null value>	EMPS_PHONES	TABLE
<null value>	OCEANS	TABLE

TABLES Rowset for MYACCOUNT

Suppose you now make the following changes in MYACCOUNT:

- Delete file STATES.

- Remove column DEPENDENTS from the dictionary of EMPS.

- Create a VOC entry called EMPLOYEES that refers to the same data file and file dictionary as does EMPS.

- Create a new file CONTINENTS that has only singlevalued columns.

- Add a record to HS_FILE_ACCESS, whose key is OCEANS and whose ACCESS field is set to NONE.

These changes are not immediately reflected in the TABLES rowset, but after MYACCOUNT's file information cache is updated, the TABLES rowset lists the following information.

Schema Name	Table Name	Table Type
<null value>	EMPS	TABLE
<null value>	EMPS_PHONES	TABLE
<null value>	EMPLOYEES	TABLE
<null value>	EMPLOYEES_PHONES	TABLE
<null value>	CONTINENTS	TABLE

Updated TABLES Rowset for MYACCOUNT

Removing File Visibility

To make files in a UniVerse account invisible to UniOLEDB:

1. On the UniVerse Server Administration menu, choose the fourth selection (Deactivate access to files in an account).

2. When prompted, enter either the full path of the UniVerse account (on Windows platforms, start with the drive letter, for example, D:) or the account name as listed in the UV.ACCOUNT file.

3. Repeat steps 1 and 2 for each account whose files you want to make invisible.

4. To exit the UniVerse Server Administration menu, press ENTER.

Making files invisible does the following:

■ Deletes the HS_FILE_ACCESS file in the account.

■ Deletes the file information cache (.hs_fileinfo) under the account's directory.

■ Updates the UV.ACCOUNT file to indicate that files in this account are not visible.

Accessing Columns in UniVerse Tables and Files

As explained earlier, NF^2 data in UniVerse appears as 1NF tables to an OLE DB consumer. All columns defined in the dictionaries of these NF^2 files (including tables and views) can be accessed. In addition to stored-data columns, I-descriptors and correlatives can be accessed (as read-only columns) by an OLE DB consumer.

Some OLE DB consumers are written as general-purpose programs that determine a list of visible columns by requesting a COLUMNS rowset. The COLUMNS rowset for any visible 1NF table describes all columns returned by the following statement:

```
SELECT * FROM tablename;
```

Columns other than those returned in the COLUMNS rowset may be accessible by OLE DB consumers because "SELECT *" does not always return all columns defined in the dictionary.

COLUMNS Rowset

A UniOLEDB consumer can request a COLUMNS rowset, which provides a list of column characteristics. Each row in the COLUMNS rowset includes the following information:

- Schema name
- Table name
- Column name
- Column position
- Data type
- Precision
- Scale

For UniVerse tables, and for associations and unassociated multivalued columns in them, columns listed in the COLUMNS rowset are defined by the dictionary's @SELECT phrase (if it exists), or are defined as the columns created by CREATE TABLE (and possibly modified by ALTER TABLE) if there is no @SELECT phrase.

For files, and for associations and unassociated multivalued columns within them, columns listed in the COLUMNS rowset are defined by the dictionary's @SELECT phrase (if it exists). If there is no @SELECT phrase, the contents of the COLUMNS rowset are defined by the dictionary's @ phrase (if it exists). If neither an @SELECT phrase nor an @ phrase exists, only the record ID (@ID) appears in the COLUMNS rowset.

For more information about the @SELECT and @ phrases, see *UniVerse Administration for DBAs*.

Association Keys

UniVerse tables have primary keys. UniVerse files have record IDs. Primary keys and record IDs are unique identifiers for each row (record) of data in a table or file.

Because UniOLEDB shows associations and unassociated multivalued columns as 1NF tables, they also require a set of unique identifiers that serve as primary keys. These keys are called association keys.

When you create a UniVerse table or file, you can define one or more association columns as the association key, but you do not have to. If you do not, UniVerse SQL generates a virtual column called @ASSOC_ROW containing unique values that, combined with the primary keys or record IDs of the base table, become the association keys for the 1NF table generated from the association.

For detailed information about defining association keys, see *UniVerse SQL Administration for DBAs* and the *UniVerse SQL Reference*.

Example

The file EMPS is defined with columns EMPNUM, NAME, DEPTNUM, DEPENDENTS, and PHONES, where DEPENDENTS and PHONES are unassociated multivalued columns. Suppose this file contains the following employees.

EMPNUM	NAME	DEPTNUM	DEPENDENTS	PHONES
4456	GONZALES	97	SUSAN	555-876-4041
			FREDERICA	
4901	HURLBUT	58		555-245-1000

EMPS File

EMPNUM	NAME	DEPTNUM	DEPENDENTS	PHONES
				555-245-1456
6511	RICHARDS	58	HELEN	400-765-4321
			ALFRED	400-765-9010
			SAMUEL	

EMPS File

This table will appear to OLE DB as three 1NF tables called EMPS, EMPS_DEPENDENTS, and EMPS_PHONES.

The EMPS 1NF table has column EMPNUM as its key.

EMPNUM	NAME	DEPTNUM
4456	GONZALES	97
4901	HURLBUT	58
6511	RICHARDS	58

EMPS NF1 Table

The EMPS_DEPENDENTS 1NF table has a key consisting of two columns: EMPNUM and @ASSOC_ROW.

EMPNUM	DEPENDENTS	@ASSOC_ROW
4456	SUSAN	1
4456	FREDERICA	2
6511	HELEN	1
6511	ALFRED	2
6511	SAMUEL	3

EMPS_DEPENDENTS 1NF Table

The EMPS_PHONES 1NF table also has a two-column key consisting of EMPNUM and @ASSOC_ROW.

EMPNUM	PHONES	@ASSOC_ROW
4456	555-876-4041	1
4901	555-245-1000	1
4901	555-245-1456	2
6511	400-765-4321	1
6511	400-765-9010	2

EMPS_PHONES 1NF Table

Table and Column Names Containing Special Characters

Some UniVerse table and column names can contain special characters such as period (.) or at-sign (@). These names can cause difficulty with some OLE DB consumers unless they are enclosed in double quotation marks.

Delimited Identifiers

UniVerse supports an ANSI-SQL feature called "delimited identifiers." This means that any identifier (table name, column name, index name, constraint name, and so forth) can be enclosed in double quotation marks to avoid ambiguity in the syntax of an SQL statement. This is particularly useful in the case of UniVerse table and column names that contain the period (.) character.

The following example shows a SELECT statement that contains column and table names delimited by double-quotation marks:

```
SELECT "MY.COLUMN" FROM SCHEMAX."MY.TABLE";
```

Making UniVerse Data Meaningful to UniOLEDB

This section describes things you may need to do on the UniVerse server to make particular kinds of UniVerse data meaningful to UniOLEDB consumers:

- Define SQL data types for data in UniVerse files.
- Define the length of character string data in UniVerse files.
- Set up empty-null mapping.
- Validate and fix data in data files and dictionaries that cause SQL and UniOLEDB problems.

SQL Data Types

To fine-tune or define data type, precision, and scale values for columns and I-descriptors in files, you can edit the DATATYPE field of the corresponding dictionary entry (field 8 in D- and I-descriptors, field 6 in A- and S-descriptors). This is especially important for character data in which the display width defined in the dictionary may be much larger or smaller than the largest data values in the file. The HS.SCRUB utility can automatically make these adjustments based on the data found. For more information about HS.SCRUB, see

If the DATATYPE field is empty, UniVerse determines a column's SQL data type by examining its conversion code and format specifications. The UniVerse server reports this SQL data type to OLE DB consumers in the COLUMNS rowset. If the UniVerse-generated SQL data type is inappropriate for the actual data in the column, you can specify the correct SQL data type in the DATATYPE field of the column's dictionary entry. For some data types, the SQL data type syntax is different between dictionary specifications and UniVerse SQL statements (for example, CREATE TABLE) as noted in the following table. Square brackets indicate optional parameters.

Dictionary Syntax	UniVerse SQL Syntax	Notes
CHAR[ACTER][,n]	CHAR[ACTER][(n)]	n = number of characters
DEC[IMAL][,p[,s]]	DEC[IMAL][(p[,s])]	p = precision s = scale
FLOAT[,p]	FLOAT[(p)]	p = precision
NUMERIC[,p[,s]]	NUMERIC[(p[,s])]	p = precision s = scale
VARCHAR[, n]	VARCHAR[(n)]	n = number of characters

SQL Data Type Syntax

Note the syntactic differences regarding the use of parentheses and commas.

The DATE, DOUBLE PRECISION, INT[EGER], REAL, SMALLINT, and TIME data type syntax is identical for dictionaries and UniVerse SQL.

You can specify the SQL data type for any column, real or virtual, in a UniVerse file. You need not specify the SQL data type for any column of a table defined by the CREATE TABLE or CREATE VIEW statement, but you may want to specify the SQL data type for other columns in the table (such as I-descriptors) that are not defined in the SICA. You cannot modify the SQL data type for columns defined in the SICA, and UniVerse ignores the dictionary definitions for these columns. For more information about the SICA and UniVerse tables, see *UniVerse SQL Administration for DBAs* manual and the *UniVerse SQL User Guide*.

Length of Character String Data

In UniOLEDB, every character column has either a fixed length or a maximum length. This column length is called its precision and is reported in the COLUMNS rowset.

The precision of a character column is determined from one of the following:

- For a columns in a UniVerse file, the precision is defined by the data type specified in the DATATYPE field of the column's dictionary definition. If the data type is not defined, the FORMAT field of the dictionary defines the precision.

- For a column defined by a CREATE TABLE or ALTER TABLE statement, the precision is defined by the column definition, which is stored in the table's SICA.

- For a column in a view, the precision is defined by the CREATE VIEW statement or by the precision of the column specified by the SELECT statement that creates the view.

- For a column added to a table's dictionary (such as an I-descriptor), the precision is defined by the data type specified in the DATATYPE field of the column's dictionary definition. If the data type is not defined, the FORMAT field of the dictionary defines the precision.

You can use the HS.SCRUB utility to examine a file's data and write appropriate data types in the DATATYPE field of the dictionary definitions for character-string columns. For information about HS.SCRUB, see "Validating and Fixing Tables and Files" on page 4-26.

The actual number of characters in a UniVerse character column can be greater than its precision. UniOLEDB retrieves such extra characters, up to a limit. The number of bytes of character data that UniOLEDB retrieves from a character column is the smallest of:

- The number of bytes of data the column actually contains.
- The number of bytes of data that UCI can fetch. This is controlled by the UCI configuration parameters MAXFETCHBUFF and MAXFETCHCOLS, defined in the uci.config file.

If your fetch buffer is not large enough to hold all the character data that UniOLEDB retrieves, UniOLEDB generates a truncation warning.

The actual number of characters in a UniVerse character column can be less than its precision. Unlike some DBMSs, UniVerse does not automatically pad CHAR(*n*) columns on the right with spaces. If you insert the value "abc " (with two trailing spaces) into a CHAR(10) column, the column contains only five characters, not 10.

Your application and data source must agree on a consistent way to treat trailing spaces in a CHAR(*n*) column. Generally, it is better to treat CHAR(*n*) columns as if they were VARCHAR columns with no space padding.

Empty-Null Mapping

UniVerse files use empty strings in much the same way tables use null values. Unfortunately, empty strings in numeric or date columns cause data conversion errors in UniOLEDB, making these columns almost inaccessible to some consumers. To make files with empty values accessible to consumers, the UniVerse server provides empty-null mapping, converting empty values in UniVerse files to null values in UniOLEDB application buffers, and vice versa.

To turn on empty-null mapping for a UniVerse table or file, add an X-descriptor named @EMPTY.NULL to the dictionary. To turn off empty-null mapping, delete the @EMPTY.NULL entry from the dictionary.

The third selection (Activate access to files in an account) on the UniVerse Server Administration menu enables empty-null mapping in all files in the account by creating @EMPTY.NULL dictionary entries.

Validating and Fixing Tables and Files

Use the HS.SCRUB utility to scan data in a table or file and fix data and dictionary problems that can cause SQL and UniOLEDB difficulties. The HS.SCRUB utility does the following:

- Reports data anomalies.

- Saves a select list of record IDs of problem records.

- Writes appropriate data types in the DATATYPE field of the dictionary's column definitions.

- Adjusts dictionary entries to accommodate bad data, such as nonnumeric values in numeric, date, and time columns, which can lead to adjusting the column type to CHARACTER.

- Adds an @EMPTY.NULL record to the dictionary.

- Adds an @SELECT record to the dictionary.

- Fixes the data in the file, optionally saving a copy of the original file.

The @EMPTY.NULL Record

HS.SCRUB adds an @EMPTY.NULL record to the table or file dictionary if all the following conditions are met:

- The dictionary does not already include an @EMPTY.NULL record.
- The data contains empty values.
- Modification of the dictionary is enabled (FIX, AUTOFIX, AUTOFIX DICT).

For information about empty-null mapping, see "Empty-Null Mapping" on page 4-26.

The @SELECT Record

HS.SCRUB adds an @SELECT record to the file dictionary (but not to a table dictionary) if both of the following conditions are met:

- The dictionary does not already include an @ or @SELECT record.
- Modification of the dictionary is enabled (FIX, AUTOFIX, AUTOFIX DICT).

For information about @SELECT records, see *UniVerse SQL Administration for DBAs*.

Running the HS.SCRUB Utility

To run the HS.SCRUB utility, choose the fifth selection (Run HS.SCRUB on a File/Table) on the UniVerse Server Administration menu, or use the HS.SCRUB command described in the next section.

If you use the Run HS.SCRUB on a File/Table selection, you are prompted to enter either the full path of the UniVerse account (for Windows platforms, start with the drive letter, such as D:) or the account name as listed in the UV.ACCOUNT file. Press ENTER to see a list of UniVerse accounts whose files have been made visible to OLE DB.

Next you are prompted to enter the name of the table or file you want to analyze or change. After you enter a file name, you are prompted to enter the mode of operation. Enter one of the following at the Mode prompt:

- To generate a report without modifying the table or file, press ENTER

- FIX
- AUTOFIX
- AUTOFIX DICT
- AUTOFIX DATA

The next subsection describes these modes.

Using the HS.SCRUB Command

To use the HS.SCRUB command, log to the UniVerse account containing the table or file you want to analyze.

The syntax of the HS.SCRUB command is as follows:

$$\text{HS.SCRUB } \textit{filename } \left[\text{FIX} \mid \text{AUTOFIX} \left[\text{DICT} \mid \text{DATA}\right]\right]$$

where *filename* is the name of the file or table to analyze.

The following table describes each parameter of the syntax.

Parameter	Description
FIX	Puts HS.SCRUB in interactive mode, in which you are prompted to resolve any anomalies following the analysis.
AUTOFIX	Puts HS.SCRUB in automatic mode; anomalies are corrected with the default action that would have been presented to the user. If you do not specify DICT or DATA with the AUTOFIX option, HS.SCRUB resolves anomalies in both the data file or table and its dictionary.
DICT	Indicates that HS.SCRUB resolves only those anomalies associated with the file's or table's dictionary.
DATA	Indicates that HS.SCRUB resolves only those anomalies associated with the file's or table's data.

HS.SCRUB Parameters

When neither FIX nor AUTOFIX is specified, HS.SCRUB only reports anomalies. No data or dictionary items are modified.

UniOLEDB Functionality

This chapter describes various aspects of UniOLEDB functionality. For more information about functionality described in this chapter, such as specific interfaces or properties, and how to develop applications with OLE DB, see the *Microsoft OLE DB 2.0 Programmer's Reference and Software Development Kit* manual.

UniOLEDB Implementation Notes

The following notes describe various aspects of how UniOLEDB is implemented.

General

- UniOLEDB is a read/write release that supports minimum OLE DB provider functionality and some extended interfaces. UniOLEDB exposes interfaces to consumers that connect to and access data from one or more UniData or UniVerse databases. It supports creating sessions, commands, and rowsets.

- UniOLEDB is deployed as an in-process server dynamic link library (DLL) that loads into the address space of the consumer.

- UniOLEDB supports the COM free-threading model, which provides for optimum speed and safety. The threading model specified in the registry for UniOLEDB is "Both," which allows single, apartment, and free-threaded consumers to access data directly.

- UniOLEDB does not support index and table definitions, which enable consumers to create, modify, and drop tables and indexes on the data source.

Data Source Objects

- UniOLEDB supports multiple instances of data source objects. It can work simultaneously with an arbitrary combination of sessions in which each session instance encapsulates a connection to either a UniData or UniVerse account.

- To create an instance of the data source object, the consumer uses the UniOLEDB class ID (IBM.UniOLEDB) with the OLE CoCreateInstance function. The resulting data source object exposes the OLE DB initialization interfaces that enable the consumer to connect to UniData and UniVerse databases.

- A connection continues until the consumer calls IDBInitialize::Uninitialize or all references to data source initialization interfaces are released.

- UniOLEDB does not support the persistence of data source objects.

Session Objects

- Each data source allows multiple session objects to be created. Each session represents a logical connection to the database and serves as the single-level transaction context.

- All command objects that UniOLEDB creates from a specific session participate in the local transaction of the session. UniOLEDB does not support coordinated or distributed transactions across multiple sessions. It also does not support nested transactions.

- A session continues until the consumer releases all references to the session object. When a session terminates, the connection closes.

- Rather than creating and releasing session objects continually, it is more efficient for the consumer to maintain a reference to at least one interface of the session object, which causes it to remain active at a minimal level.

Command Objects

- Sessions support multiple command object instances. Any SQL statement the UniData or UniVerse engine (whichever one to which you are connected) supports can be executed as a command object. In particular, both UniData and UniVerse support at least the ANSI92_ENTRY level definition of SQL. For more information about SQL support, see "SQL Support" on page 5-62. In addition, UniOLEDB supports CALL commands.

Note: UniOLEDB does not support data definition language (DDL) statements sent to UniData.

- UniOLEDB supports prepared commands and parameters, which are variables in text commands that can be used in conjunction with prepared commands. A consumer prepares a command to execute it multiple times. Because prepared commands are resource intensive, IBM recommends that a consumer should use them only if the command will be executed multiple times.

- UniOLEDB does not support multiple-rowset results.

- UniOLEDB does not support the use of catalog names in text commands. A catalog is a database that contains one or more schemas. A schema is a group of related tables and files contained in a UniVerse account directory and listed in the SQL catalog. Schemas do not apply to UniData.

Rowset Objects

- UniOLEDB supports the IOpenRowset::OpenRowset interface.

- UniOLEDB can update rowsets only through the Microsoft Cursor Engine service component, which converts rowset updates to SQL statements. The ICommand interface runs the SQL statements. Consumers also can update UniData or UniVerse data directly by using SQL through the ICommand interface.

- The Microsoft Cursor Engine service component supports scrollable UniOLEDB rowsets.

- UniOLEDB does not expose index rowsets through the IRowsetIndex interface. For optimum efficiency, IBM follows Microsoft's recommendation that consumers of SQL providers primarily should use the ICommand interface to access data and rely on the query processor in the SQL provider to optimize data access.

- UniOLEDB supports the following types of schema rowsets:

  - COLUMNS
  - PROVIDER_TYPES
  - SCHEMATA (for UniVerse only)
  - TABLES

- UniOLEDB does not support resynchronizing rows in a rowset with those in the data source.

- UniOLEDB does not support bulk-copy rowsets.

- UniOLEDB does not support chaptered rowsets.

- UniOLEDB does not support consumer access to binary large objects (BLOBs) that are retrieved as storage objects through the ISequentialStream, IStream, IStorage, and ILockBytes interfaces.

- UniOLEDB does not notify a consumer about any changes to rowsets.

Error Objects

- UniOLEDB data source, session, command, and rowset objects support the ISupportErrorInfo interface because they expose interfaces that return error objects. For information about how UniOLEDB implements error objects, see "Error Support" on page 5-60.

Supported Interfaces

The following table describes the interfaces that UniOLEDB supports. Interfaces are grouped according to COM object type (for example, data source objects).

Each interface listed in the table includes one or more methods to perform the tasks that are described in the Description column. For more detailed descriptions of the interfaces and how they can be used, see the *Microsoft OLE DB 2.0 Programmer's Reference and Software Development Kit* manual.

Interface	Description
Interfaces for data source objects:	
IDBCreateSession	Creates a new session from a data source object.
IDBInitialize	Initializes a data source object, or uninitializes it.
IDBProperties	Enumerates, sets, or gets values of properties on a data source object, or gets a list of all properties UniOLEDB supports.
IPersist	Retrieves the persisted information on a data source object.
ISupportErrorInfo	Indicates whether a specific OLE DB interface that is exposed by a data source object supports OLE DB error objects.
Interfaces for session objects:	
IDBCreateCommand	Creates a new command from a session.
IDBSchemaRowset	Exposes information about the structure of a data source (metadata).
IGetDataSource	Obtains an interface pointer to the data source object that created the session.
IOpenRowset	Creates a simple rowset of all rows in a table.
ISessionProperties	Sets properties in the Session property group, or returns information about the properties a session supports, including their current settings.

Interfaces Supported by UniOLEDB

Interface	Description
ISupportErrorInfo	Indicates whether a specific OLE DB interface that is exposed by a session object supports OLE DB error objects.
ITransaction	Commits, aborts, or obtains information about manual-commit transactions.
ITransactionLocal	Starts, commits, or aborts local manual-commit transactions. ITransactionLocal inherits from ITransaction.
Interfaces for command objects:	
IAccessor	Provides methods for managing accessors on a command, including methods for adding a reference count to an existing accessor, creating an accessor from a set of bindings, and releasing an accessor.
IColumnsInfo	Exposes certain information about columns of a rowset or a prepared command, such as column metadata and ordinals of columns.
IColumnsRowset	Exposes complete information about columns in a rowset. UniOLEDB supports this interface through the Microsoft Cursor Engine service component.
ICommand	Provides methods for managing commands, including methods for cancelling a command, executing a command, and returning an interface pointer to a session that created the command.
ICommandPrepare	Validates and optimizes a current command, which results in a command execution plan, or discards a current command execution plan.
ICommandProperties	Sets properties in the Rowset property group, or returns a list of rowset properties that currently are requested for a rowset.
ICommandText	Sets or returns the text command of a command object.

Interfaces Supported by UniOLEDB (Continued)

Interface	Description
ICommandWithParameters	Encapsulates the parameters of a command, or sets and obtains information about the parameters and their native data types.
IConvertType	Exposes information about the types of data conversions a command supports.
ISupportErrorInfo	Indicates whether a specific OLE DB interface that is exposed by a command object supports OLE DB error objects.
Interfaces for rowset objects:	
IAccessor	Provides methods for managing accessors on a rowset, including methods for adding a reference count to an existing accessor, creating an accessor from a set of bindings, and releasing an accessor.
IColumnsInfo	Exposes certain information about rowset columns, such as column metadata and ordinals of columns.
IConvertType	Exposes information about the types of data conversions a rowset supports.
IRowset	Provides methods for managing rowsets, including methods for adding a reference count to an existing row handle, fetching rows sequentially, getting the data from those rows, and releasing rows.
IRowsetChange	Provides methods for managing rowsets, including methods for updating values of columns in existing rows, deleting existing rows, and inserting new rows. UniOLEDB supports this interface through the Microsoft Cursor Engine service component.
IRowsetFind	Finds a row in a rowset that matches the criteria a consumer specifies. UniOLEDB supports this interface through the Microsoft Cursor Engine service component.
IRowsetIdentity	Indicates the identity of a row instance, and tests by comparison whether two row handles refer to the same row instance.

Interfaces Supported by UniOLEDB (Continued)

Interface	Description
IRowsetInfo	Returns information about the properties a rowset supports (including their current settings) or obtains an interface pointer to the command session that created the rowset.
IRowsetLocate	Fetches arbitrary rows of a rowset. Methods also enable a consumer to compare bookmarks and retrieve hash values for bookmarks the consumer specifies. UniOLEDB supports this interface through the Microsoft Cursor Engine service component.
IRowsetScroll	Locates the approximate position of a row in a rowset, or fetches rows starting from a fractional position in a rowset. UniOLEDB supports this interface through the Microsoft Cursor Engine service component.
IRowsetUpdate	Indicates that the rowset is in delayed update mode. If a rowset exposes IRowsetUpdate, it is in delayed update mode. The methods in IRowsetChange do not transmit changes to the data source immediately, and the changes are pending until the consumer calls IRowsetUpdate::Update. If a rowset does not expose IRowsetUpdate, it is in immediate update mode. UniOLEDB supports this interface through the Microsoft Cursor Engine service component.
ISupportErrorInfo	Indicates whether a specific OLE DB interface that is exposed by a rowset object supports OLE DB error objects.
Interfaces for error objects:	
IErrorLookup	Provides methods for retrieving error information based on the return code and the error number UniOLEDB specifies. Information can include error message text and source, the path of the Help file, and the context ID of the topic that explains the error.

Interfaces Supported by UniOLEDB (Continued)

UniOLEDB Properties

UniOLEDB defines properties for the following OLE DB property groups:

- Data Source Information (DBPROPSET_DATASOURCEINFO)
- Initialization (DBPROPSET_DBINIT)
- Rowset (DBPROPSET_ROWSET)
- Session (DBPROPSET_SESSION)

The tables in this section describe each property UniOLEDB supports. Each description includes a general OLE DB description of the property and information about how UniOLEDB defines it. For more detailed descriptions of the properties and how a provider can use them, see the *Microsoft OLE DB 2.0 Programmer's Reference and Software Development Kit* manual.

UniOLEDB does not support any properties other than those described in the following tables.

Data Source Information

The following table describes the properties UniOLEDB supports in the Data Source Information property group.

Property	Variant Type	Description
DBPROP_ACTIVESESSIONS	VT_I4	**General:** The Active Sessions property indicates the maximum number of sessions that can exist concurrently.
		UniOLEDB: The property is set to zero (0), which means that the number of sessions that can exist concurrently are limited by server licensing.
		R/W: Read only.
DBPROP_ASYNCTXNABORT	VT_BOOL	**General:** The Asynchable Abort property indicates whether a consumer can terminate transactions asynchronously.
		UniOLEDB: The property is set to VARIANT_ FALSE.
		R/W: Read only.
DBPROP_ASYNCTXNCOMMIT	VT_BOOL	**General:** The Asynchable Commit property indicates whether a consumer can commit transactions asynchronously.
		UniOLEDB: The property is set to VARIANT_FALSE.
		R/W: Read only.

Data Source Information Properties

Property	Variant Type	Description
DBPROP_BYREFACCESSORS	VT_BOOL	**General:** The Pass By Ref Accessors property indicates whether the provider supports reference accessors for rows and parameters. The provider creates them from a set of bindings if the DBACCESSOR_PASSBYREF flag is set in IAccessor::CreateAccessor. **UniOLEDB:** The property is set to VARIANT_FALSE. **R/W:** Read only.
DBPROP_COLUMNDEFINITION	VT_I4	**General:** The Column Definition property indicates the valid clauses for the column definition. **UniOLEDB:** The property is set to DBPROPVAL_CD_NOTNULL, which means that UniOLEDB can create columns that are not NULL valued. **R/W:** Read only.
DBPROP_CONCATNULLBEHAVIOR	VT_I4	**General:** The NULL Concate-nation Behavior property indicates how the data source handles concatenating character data type columns that are NULL valued with those that are not NULL valued. **UniOLEDB:** The property is set to DBPROPVAL_CB_NULL, which means that the result is NULL valued. **R/W:** Read only.

Data Source Information Properties (Continued)

Property	Variant Type	Description
DBPROP_DATASOURCENAME	VT_BSTR	**General:** The Data Source Name property indicates the name of the data source, which the provider uses during the connection process. **UniOLEDB:** The property is set to the name of the data source. **B/W:** Read only.
DBPROP_DATASOURCEREADONLY	VT_BOOL	**General:** The Read-Only Data Source property indicates whether the data source is read-only. **UniOLEDB:** The property is set to VARIANT_FALSE, which means that the UniData or UniVerse database is updatable. **R/W:** Read only.
DBPROP_DBMSNAME	VT_BSTR	**General:** The DBMS Name property indicates the name of the data management product the provider has accessed. **UniOLEDB:** For UniData, the property returns "UniData." For UniVerse, the property returns "UniVerse." **R/W:** Read only.

Data Source Information Properties (Continued)

Property	Variant Type	Description
DBPROP_DBMSVER	VT_BSTR	**General:** The DBMS Version property indicates the version of the data management product the provider has accessed.
		UniOLEDB: The property returns the version in the form ##.##.#### in which the first two digits are the major version, the second two digits are the minor version, and the remaining four digits are the release version.
		For example, the value returned for UniData release 5.2 would be "05.02.0000". The value returned for UniVerse release 9.6 would be "09.06.0000".
		R/W: Read only.
DBPROP_DSOTHREADMODEL	VT_I4	**General:** The Data Source Object Threading Model property indicates the threading model the provider supports for the data source object.
		UniOLEDB: The property is set to DBPROPVAL_RT_FREETHRE ADwhich means that the data source object supports the free-threading model.
		R/W: Read only.

Data Source Information Properties (Continued)

Property	Variant Type	Description
DBPROP_GROUPBY	VT_I4	**General:** The GROUP BY Support property indicates how columns in a GROUP BY clause relate to the nonaggregated columns in the select list.
		UniOLEDB: For UniData, the property is set to DBPROPVAL_GB_EQUALS_ SELECT, which means that the GROUP BY clause must include all the nonaggregated columns that are in the select list. It cannot include any other columns.
		For UniVerse, the property is set to DBPROPVAL_GB_CONTAINS _ SELECT, which means that the GROUP BY clause must include all the nonaggregated columns that are in the select list. It can include columns that are not in the select list.
		R/W: Read only.
DBPROP_IDENTIFIERCASE	VT_I4	**General:** The Identifier Case Sensitivity property indicates how SQL identifiers handle case types (uppercase, lowercase, and mixed case characters).
		UniOLEDB: The property is set to DBPROPVAL_IC_SENSITIVE, which means that SQL identifiers are case sensitive and are stored in the system catalog in mixed case.
		R/W: Read only.

Data Source Information Properties (Continued)

Property	Variant Type	Description
DBPROP_MAXINDEXSIZE	VT_I4	**General:** The Maximum Index Size property indicates the maximum number of bytes the provider allows in the combined columns of an index.
		UniOLEDB: For UniData, the property is set to zero (0) because no specific limit exists. For UniVerse, it is set to 254.
		R/W: Read only.
DBPROP_MAXROWSIZE	VT_I4	**General:** The Maximum Row Size property indicates the maximum length of a single table row.
		UniOLEDB: The property is set to zero (0) because no specific limit exists.
		R/W: Read only.
DBPROP_ MAXROWSIZEINCLUDESBLOB	VT_BOOL	**General:** The Maximum Row Size Includes BLOB property indicates whether the maximum row size the DBPROP_ MAXROWSIZE property returns includes the length of all BLOB data.
		UniOLEDB: The property is set to VARIANT_FALSE.
		R/W: Read only.

Data Source Information Properties (Continued)

Property	Variant Type	Description
DBPROP_MAXTABLESINSELECT	VT_I4	**General:** The Maximum Tables In SELECT property indicates the maximum number of tables the provider allows in the FROM clause of a SELECT statement. **UniOLEDB:** For UniData, the property is set to 7. For UniVerse, it is set to zero (0) because no specific limit exists. **R/W:** Read only.
DBPROP_MULTIPLEPARAMSETS	VT_BOOL	**General:** The Multiple Parameter Sets property indicates whether the provider supports multiple parameter sets. **UniOLEDB:** The property is set to VARIANT_FALSE. **R/W:** Read only.
DBPROP_MULTIPLERESULTS	VT_I4	**General:** The Multiple Results property indicates whether the provider supports multiple results objects, which are used to retrieve multiple results. It also indicates the restrictions the provider places on them. **UniOLEDB:** The property is set to DBPROPVAL_MR_ NOTSUPPORTED, which means that UniOLEDB does not support them. **R/W:** Read only.

Data Source Information Properties (Continued)

Property	Variant Type	Description
DBPROP_MULTITABLEUPDATE	VT_BOOL	**General:** The Multi-Table Update property indicates whether the provider can update rowsets obtained from multiple tables.
		UniOLEDB: The property is set to VARIANT_FALSE.
		R/W: Read only.
DBPROP_NULLCOLLATION	VT_I4	**General:** The NULL Collation Order property indicates the position in a list at which the NULLs are sorted.
		UniOLEDB: For UniData, the property is set to DBPROPVAL_NC_LOW, which means that NULLs are sorted in the lower portion of the list. For UniVerse, it is set to DBPROPVAL_NC_HIGH, which means that NULLs are sorted in the higher portion of the list.
		R/W: Read only.
DBPROP_ ORDERBYCOLUMNSINSELECT	VT_BOOL	**General:** The ORDER BY Columns in Select List property indicates whether the consumer must include the columns in an ORDER BY clause in the select list.
		UniOLEDB: The property is set to VARIANT_FALSE.
		R/W: Read only.

Data Source Information Properties (Continued)

Property	Variant Type	Description
DBPROP_OUTPUTPARAMETERAVAILABILITY	VT_I4	**General:** The Output Parameter Availability property indicates the time at which output parameter data becomes available to the consumer.
		UniOLEDB: The property is set to DBPROPVAL_OA_ ATEXECUTE, which means that output parameter data is available immediately after ICommand::Execute returns.
		R/W: Read only.
DBPROP_PREPAREABORTBEHAVIOR	VT_I4	**General:** The Prepare Abort Behavior property indicates how terminating a transaction affects prepared commands.
		UniOLEDB: The property is set to DBPROPVAL_CB_ PRESERVE, which means that terminating a transaction preserves prepared commands. The consumer does not need to prepare the commands again to run them again.
		R/W: Read only.

Data Source Information Properties (Continued)

Property	Variant Type	Description
DBPROP_ PREPARECOMMITBEHAVIOR	VT_I4	**General:** The Prepare Commit Behavior property indicates how committing a transaction affects prepared commands.
		UniOLEDB: The property is set to DBPROPVAL_CB_PRESERVE, which means that committing a transaction preserves prepared commands. The consumer does not need to prepare the commands again to run them again.
		R/W: Read only.
DBPROP_PROCEDURETERM	VT_BSTR	**General:** The Procedure Term property indicates the name the data source vendor uses for the term "procedure". Application development tools use this term to build user interfaces.
		UniOLEDB: The property is set to "Procedure".
		R/W: Read only.
DBPROP_PROVIDERNAME	VT_BSTR	**General:** The Provider Name property indicates the full file name of the provider.
		UniOLEDB: The property is set to UNIOLEDB.DLL.
		R/W: Read only.

Data Source Information Properties (Continued)

Property	Variant Type	Description
DBPROP_PROVIDEROLEDBVER	VT_BSTR	**General:** The OLE DB Version property indicates the version of OLE DB the provider supports.
		UniOLEDB: The property returns the version in the form ##.## in which the first two digits are the major version and the final two digits are the minor version. For example, UniOLEDB supports OLE DB release 2.1, so the value returned would be "02.01".
		R/W: Read only.
DBPROP_PROVIDERVER	VT_BSTR	**General:** The Provider Version property indicates the version of the provider.
		UniOLEDB: The property returns the version in the form ##.##.#### in which the first two digits are the major version, the second two digits are the minor version, and the remaining four digits are the release version. For example, the value returned for UniOLEDB release 1.0 could be "01.00.0000".
		R/W: Read only.

Data Source Information Properties (Continued)

Property	Variant Type	Description
DBPROP_QUOTEDIDENTIFIERCASE	VT_I4	**General:** The Quoted Identifier Sensitivity property indicates how SQL quoted identifiers handle case types (uppercase, lowercase, and mixed case characters).
		UniOLEDB: The property is set to DBPROPVAL_IC_ SENSITIVE, which means that SQL quoted identifiers are case sensitive.
		R/W: Read only.
DBPROP_ ROWSETCONVERSIONSONCOMMAND	VT_BOOL	**General:** The Rowset Conversions On Command property indicates whether a consumer that calls IConvertType:: CanConvert can inquire on a command about conversions supported on rowsets the command creates.
		UniOLEDB: The property is set to VARIANT_TRUE.
		R/W: Read only.
DBPROP_SCHEMATERM	VT_BSTR	**General:** The Schema Term property indicates the name the data source vendor uses for the term "schema". Application development tools use this term to build user interfaces.
		UniOLEDB: The property is set to "Schema".
		R/W: Read only.

Data Source Information Properties (Continued)

Property	Variant Type	Description
DBPROP_SCHEMAUSAGE	VT_I4	**General:** The Schema Usage property indicates the types of text commands that will support schema names.
		UniOLEDB: For UniData, the property is set to zero (0) because UniData does not support schema names in text commands. For UniVerse, it is set to DBPROPVAL_SU_DML_ STATEMENTS, which means that schema names are supported in DML statements only.
		R/W: Read only.
DBPROP_SORTONINDEX	VT_BOOL	**General:** The Sort On Index property indicates whether the provider supports SetSortOrder.
		UniOLEDB: The property is set to VARIANT_FALSE.
		R/W: Read only.
DBPROP_SQLSUPPORT	VT_I4	**General:** The SQL Support property indicates the level at which the provider supports SQL.
		UniOLEDB: The property is set to DBPROPVAL_SQL_ ANSI92_ENTRY.
		R/W: Read only.

Data Source Information Properties (Continued)

Property	Variant Type	Description
DBPROP_SUBQUERIES	VT_I4	**General:** The Subquery Support property indicates the predicates in text commands that support subqueries.
		UniOLEDB: The property is set to include all of the following DBPROPVAL_SQ_ values:
		COMPARISON
		EXISTS
		IN
		QUANTIFIED
		CORRELATEDSUBQUERIES
		CORRELATEDSUBQUERIES indicates that all predicates that support subqueries also support correlated subqueries.
		R/W: Read only.
DBPROP_SUPPORTEDTXNDDL	VT_I4	**General:** The Transaction DDL property indicates whether the provider supports DDL statements in transactions and to what extent it supports them.
		UniOLEDB: The property is set to DBPROPVAL_TC_DML, which means that transactions can contain Data Manipulation Language (DML) statements only. Transactions that contain DDL statements cause errors.
		R/W: Read only.

Data Source Information Properties (Continued)

Property	Variant Type	Description
DBPROP_SUPPORTEDTXNISOLEVELS	VT_I4	**General:** The Isolation Levels property indicates the transaction isolation levels the provider supports.
		UniOLEDB: The property is set to include all of the following DBPROPVAL_TI_ values:
		READUNCOMMITTED
		BROWSE
		CURSORSTABILITY
		READCOMMITTED
		REPEATABLEREAD
		SERIALIZABLE
		ISOLATED
		R/W: Read only.
DBPROP_SUPPORTEDTXNISORETAIN	VT_I4	**General:** The Isolation Retention property indicates the transaction isolation retention levels the provider supports.
		UniOLEDB: The property is set to DBPROPVAL_TR_ DONTCARE, which means that the transaction can preserve or dispose of isolation context across a retaining commit or abort.
		R/W: Read only.
DBPROP_TABLETERM	VT_BSTR	**General:** The Table Term property indicates the name the data source vendor uses for the term "table". Application development tools use this term to build user interfaces.
		UniOLEDB: The property is set to "Table".
		R/W: Read only.

Data Source Information Properties (Continued)

Initialization

The following table describes the properties UniOLEDB supports in the Initialization property group (including authentication and initialization properties).

Property	Variant Type	Description
DBPROP_AUTH_CACHE_AUT HINFO	VT_BOOL	**General:** The Cache Authentication authentication property indicates whether the data source object can cache sensitive authentication infor-mation, such as a password. **UniOLEDB:** The property is set to VARIANT_FALSE. **R/W:** Read only.
DBPROP_AUTH_ENCRYPT_PA SSWORD	VT_BOOL	**General:** The Encrypt Password authentication property indicates whether the consumer requires the provider to send the password to the data source object in an encrypted form. This property specifies a stronger form of masking than DBPROP_AUTH_MASK_ PASSWORD specifies. **UniOLEDB:** The property is set to VARIANT_FALSE because UniOLEDB does not support sending passwords in an encrypted form. **R/W:** Read only.
DBPROP_AUTH_INTEGRATED	VT_BSTR	**General:** The Integrated Security authentication property indicates the name of the authentication service the server uses. This service identifies the users who use identities an authenti-cation domain provides. **UniOLEDB:** The property is set to an empty string, which means that the server should use its own authentication process. **R/W:** Read only.

Initialization Properties

Property	Variant Type	Description
DBPROP_AUTH_MASK_PASS WORD	VT_BOOL	**General:** The Mask Password authentication property indicates whether the consumer requires the provider to send the password to the data source object in a masked form.
		UniOLEDB: The property is set to VARIANT_FALSE because UniOLEDB does not support sending passwords in a masked form.
		R/W: Read only.
DBPROP_AUTH_PASSWORD	VT_BSTR	**General:** The Password authentication property indicates the password the provider should use when it connects to the data source.
		UniOLEDB: The password must be set by using IDBProperties::SetProperties before calling IDBInitialize::Initialize.
		R/W: Read/write.
DBPROP_AUTH_PERSIST_ENC RYPTED	VT_BOOL	**General:** The Persist Encrypted authentication property indicates whether the consumer requires the data source object to persist sensitive authentication information, such as a password, in an encrypted form.
		UniOLEDB: the property is set to VARIANT_FALSE because UniOLEDB does not support persisting sensitive authentication information in an encrypted form.
		R/W: Read only.

Initialization Properties (Continued)

Property	Variant Type	Description
DBPROP_AUTH_PERSIST_SEN SITIVE_ AUTHINFO	VT_BOOL	**General:** The Persist Security Info authentication property indicates whether the data source object can persist sensitive authentication infor- mation, such as a password, with other authentication information.
		UniOLEDB: The property is set to VARIANT_FALSE, which means that the data source object will not persist sensitive authentication information.
		R/W: Read only.
DBPROP_AUTH_USERID	VT_BSTR	**General:** The User ID authentication property indicates the user ID the provider should use to connect to the data source.
		UniOLEDB: The user ID must be set by using IDBProperties::SetProperties before calling IDBInitialize::Initialize.
		R/W: Read/write.
DBPROP_INIT_ASYNCH	VT_I4	**General:** The Asynchronous Processing initialization property indicates the type of asynchronous processing that is performed on the data source.
		UniOLEDB: The property is not set, which means that IDBInitialize::Initialize does not return initialization information until the data source is initialized completely.
		R/W: Read only.

Initialization Properties (Continued)

Property	Variant Type	Description
DBPROP_INIT_DATASOURCE	VT_BSTR	**General:** The Data Source initial-ization property indicates the name of the data source to which to connect.
		UniOLEDB: The data source name must be set by using IDBProperties:: SetProperties before calling IDBInitialize::Initialize. The data source name must be the name of a UCI data source in the UCI configuration file (uci.config).
		R/W: Read/write.
DBPROP_INIT_HWND	VT_I4	**General:** The Window Handle initial-ization property indicates the window handle to use if the data source object needs to prompt for additional information.
		UniOLEDB: The window handle must be set by using IDBProperties:: SetProperties before calling IDBInitialize::Initialize.
		R/W: Read/write.
DBPROP_INIT_IMPERSONATI ON_ LEVEL	VT_I4	**General:** The Impersonation Level initialization property indicates the level of impersonation the server can use to impersonate the client.
		UniOLEDB: The property is set to zero (0) because UniOLEDB does not support impersonation levels.
		R/W: Read only.

Initialization Properties (Continued)

Property	Variant Type	Description
DBPROP_INIT_LCID	VT_I4	**General:** The Locale Identifier initialization property indicates the preferred locale ID (LCID) for the consumer.
		UniOLEDB: The property is set to MAKELANGID(LANG_ENGLISH, SUBLANG_ENGLISH_US), which means that the US English locale is preferred.
		R/W: Read only.
DBPROP_INIT_LOCATION	VT_BSTR	**General:** The Location initialization property indicates the location of the data source to which to connect. The location is the name of the account. If specified, the account name overrides the one that appears in the UCI configuration file (uci.config).
		UniOLEDB: The location must be set by using IDBProperties::SetProperties before calling IDBInitialize::Initialize.
		R/W: Read/write.
DBPROP_INIT_MODE	VT_I4	**General:** The Mode initialization property indicates access permissions.
		UniOLEDB: The mode must be set by using IDBProperties::SetProperties before calling IDBInitialize::Initialize.
		R/W: Read/write.

Initialization Properties (Continued)

Property	Variant Type	Description
DBPROP_INIT_PROMPT	VT_I2	**General:** The Prompt initialization property indicates whether to prompt the user during initialization and under what conditions.
		UniOLEDB: By default, the property is set to DBPROMPT_NOPROMPT, which means that the user should not be prompted. It can be reset to any of the following DBPROMPT_ values:
		PROMPT
		COMPLETE
		COMPLETEREQUIRED
		R/W: Read/write.

Initialization Properties (Continued)

Property	Variant Type	Description
DBPROP_INIT_PROTECTION_LEVEL	VT_I4	**General:** The Protection Level initialization property indicates the level of data protection as the data moves from the client to the server.
		UniOLEDB: The property is set to DB_PROT_LEVEL_NONE, which means that no data authentication occurs.
		R/W: Read only.
DBPROP_INIT_PROVIDERSTRING	VT_BSTR	**General:** The Extended Properties initialization property indicates extended connection information that is specific to the provider.
		UniOLEDB: The property is set to an empty string, which means that UniOLEDB does not have connection information it needs to specify.
		R/W: Read only.
DBPROP_INIT_TIMEOUT	VT_I4	**General:** The Connect Timeout initialization property indicates the amount of time, in seconds, to wait before initialization times out.
		UniOLEDB: Beginning at UniVerse 10.2, the UniOLEDB provider supports the connection timeout property.
		R/W: Read/write.

Initialization Properties (Continued)

Rowset

The following table describes the properties UniOLEDB supports in the Rowset property group.

Property	Variant Type	Description
DBPROP_ ABORTPRESERVE	VT_BOOL	**General:** The Preserve On Abort property indicates whether the rowset remains active after an abort that preserves occurs for a transaction. **UniOLEDB:** The property is set to VARIANT_FALSE, which means that the only operations allowed on a rowset if an abort or an abort that preserves occurs are: Releasing row and accessor handles. Releasing the rowset. **R/W:** Read only.
DBPROP_APPENDON LY	VT_BOOL	**General:** The Append-Only Rowset property indicates whether a rowset that is opened will be empty initially and, therefore, will be populated only by the rows that are inserted in it. **UniOLEDB:** The property is set to VARIANT_FALSE, which means that UniOLEDB does not support opening an append-only rowset. **R/W:** Read only.

Rowset Properties

Property	Variant Type	Description
DBPROP_ BLOCKINGSTORAGEOBJECTS	VT_BOOL	**General:** The Blocking Storage Objects property indicates whether storage objects could prevent using other methods on the rowset. **UniOLEDB:** The property is set to VARIANT_TRUE. After UniOLEDB creates a storage object, but before it releases the object, methods other than those used on the storage object could return E_UNEXPECTED. **R/W:** Read only.
DBPROP_BOOKMARKS	VT_BOOL	**General:** The Use Bookmarks property indicates whether the rowset supports bookmarks. UniOLEDB supports this property only if it is used with the Microsoft Cursor Engine service component. **UniOLEDB:** By default, the property is set to VARIANT_FALSE. It can be set to VARIANT_ TRUE. The property will be set automatically to VARIANT_TRUE if the consumer sets any of the following DBPROP_ values to VARIANT_TRUE: IROWSETLOCATE LITERALBOOKMARKS ORDEREDBOOKMARKS When the property is set to VARIANT_ TRUE, column zero (0) is the bookmark for the rows. Getting this column obtains a bookmark value that can be used to reposition to the row. **R/W:** Read/write.

Rowset Properties

Property	Variant Type	Description
DBPROP_BOOKMARKSKIPPED	VT_BOOL	**General:** The Skip Deleted Bookmarks property indicates whether the rowset allows IRowsetLocate::GetRowsAt, IRowsetScroll::Approximate-Position, or IRowsetFind::FindN-extRow to continue if any of the following conditions are true:
		A bookmark row was deleted.
		It is a row to which the consumer does not have access privileges.
		It is no longer a member of the rowset.
		UniOLEDB supports this property only if it is used with the Microsoft Cursor Engine service component.
		UniOLEDB: The property is set to VARIANT_FALSE, which means the IRowsetLocate::GetRowsAt, IRowsetScroll::Approximate-Position, or IRowsetFind::FindN-extRow interface returns DB_E_BADBOOKMARK if any of the above conditions are true.
		R/W: Read only.

Rowset Properties (Continued)

Property	Variant Type	Description
DBPROP_BOOKMARKTYPE	VT_I4	**General:** The Bookmark Type property indicates the type of bookmark (numeric or key value) the rowset supports. UniOLEDB supports this property only if it is used with the Microsoft Cursor Engine service component. **UniOLEDB:** The property is set to DBPROPVAL_BMK_NUMERIC, which means the rowset supports the numeric bookmark type. Numeric bookmarks are not based on the values of a row's columns, but on a row property, such as the absolute position of a row in a rowset or a row ID that the storage assigned to a tuple when it was created. **R/W:** Read only.
DBPROP_CACHEDEFERRED	VT_BOOL	**General:** The Cache Deferred Columns property indicates whether the provider caches the values of deferred columns. **UniOLEDB:** The property is set to VARIANT_FALSE. **R/W:** Read only.
DBPROP_ CANFETCHBACKWARDS	VT_BOOL	**General:** The Fetch Backwards property indicates whether the rowset can fetch backward from the specified row. **UniOLEDB:** The property is set to VARIANT_FALSE. The cRows parameter in IRowset::GetNextRows, IRowsetLocate::GetRowsAt, and IRowsetScroll::GetRowsAtRatio cannot be a negative value. **R/W:** Read only.

Rowset Properties (Continued)

Property	Variant Type	Description
DBPROP_CANHOLDROWS	VT_BOOL	**General:** The Hold Rows property indicates whether the rowset allows the consumer to fetch additional rows, or change the next fetch position, while holding previously fetched rows with pending changes.
		UniOLEDB: The property is set to VARIANT_FALSE, which means the rowset requires pending changes to be transmitted to the data source and all rows to be released before fetching additional rows, inserting new rows, or changing the next fetch position.
		R/W: Read only.
DBPROP_ CANSCROLLBACKWARDS	VT_BOOL	**General:** The Can Scroll Backward property indicates whether the provider supports backward scrolling for rowsets.
		UniOLEDB supports this property only if it is used with the Microsoft Cursor Engine service component.
		UniOLEDB: By default, the property is set to VARIANT_FALSE. It can be set to VARIANT_TRUE. The property will be set automatically to VARIANT_TRUE if DBPROP_IROWSETLOCATE is set to VARIANT_TRUE.
		R/W: Read/write.

Rowset Properties (Continued)

Property	Variant Type	Description
DBPROP_ CHANGEINSERTEDROWS	VT_BOOL	**General:** The Change Inserted Rows property indicates whether the consumer can set data values in row columns (IRowsetChange::SetData) or delete rows in newly inserted rows (IRowsetChange:: DeleteRows). A newly inserted row is one for which an insertion has been transmitted to the data source (not a pending insert row).
		UniOLEDB: The property is set to VARIANT_TRUE.
		UniOLEDB supports this property only if it is used with the Microsoft Cursor Engine service component.
		R/W: Read only.
DBPROP_COLUMNRESTRICT	VT_BOOL	**General:** The Column Privileges property indicates whether the provider restricts access to specific columns.
		UniOLEDB: The property is set to VARIANT_FALSE.
		R/W: Read only.
DBPROP_COMMANDTIMEOUT	VT_I4	**General:** The Command Time Out property indicates the amount of time in seconds to wait before a command times out.
		UniOLEDB: Beginning at UniVerse 10.2, UniOLEDB supports the command timeout property.
		R/W: Read/write.

Rowset Properties (Continued)

Property	Variant Type	Description
DBPROP_COMMITPRESERVE	VT_BOOL	**General:** The Preserve On Commit property indicates whether the rowset remains active after a commit that preserves occurs.
		UniOLEDB: The property is set to VARIANT_FALSE, which means that the only operations UniOLEDB allows on a rowset after a commit or commit that preserves occurs are:
		Releasing row and accessor handles.
		Releasing the rowset.
		R/W: Read only.
DBPROP_DEFERRED	VT_BOOL	**General:** The Defer Column property indicates whether the column data is fetched only when an accessor is used on the column.
		UniOLEDB: The property is set to VARIANT_FALSE, which means the column data is fetched when the row that contains it is fetched.
		R/W: Read only.
DBPROP_ DELAYSTORAGEOBJECTS	VT_BOOL	**General:** The Delay Storage Object Updates property indicates whether the storage objects also will be used in delayed update mode when the rowset is in delayed update mode.
		UniOLEDB: The property is set to VARIANT_FALSE because UniOLEDB does not support storage objects.
		R/W: Read only.

Rowset Properties (Continued)

Property	Variant Type	Description
DBPROP_IACCESSOR	VT_BOOL	**General:** The IAccessor property indicates whether the rowset supports the IAccessor interface. **UniOLEDB:** The property is set to VARIANT_TRUE. **R/W:** Read only.
DBPROP_ICOLUMNSINFO	VT_BOOL	**General:** The IColumnsInfo property indicates whether the rowset supports the IColumnsInfo interface. **UniOLEDB:** The property is set to VARIANT_TRUE. **R/W:** Read only.
DBPROP_ICONVERTTYPE	VT_BOOL	**General:** The IConvertType property indicates whether the rowset supports the IConvertType interface. **UniOLEDB:** The property is set to VARIANT_TRUE. **R/W:** Read only.
DBPROP_IMMOBILEROWS	VT_BOOL	**General:** The Immobile Rows property indicates whether the rowset will not reorder inserted or updated rows. **UniOLEDB:** The property is set to VARIANT_TRUE. For IRowsetChange:: InsertRow, inserted or updated rows will be positioned at the end of the rowset. **R/W:** Read only.

Rowset Properties (Continued)

Property	Variant Type	Description
DBPROP_IROWSET	VT_BOOL	**General:** The IRowset property indicates whether the rowset supports the IRowset interface.
		UniOLEDB: The property is set to VARIANT_TRUE.
		R/W: Read only.
DBPROP_IROWSETCHANGE	VT_BOOL	**General:** The IRowsetChange property indicates whether the rowset supports the IRowsetChange interface.
		UniOLEDB supports this property only if it is used with the Microsoft Cursor Engine service component.
		UniOLEDB: By default, the property is set to VARIANT_FALSE. It can be set to VARIANT_TRUE.
		R/W: Read/write.
DBPROP_IROWSETFIND	VT_BOOL	**General:** The IRowsetFind property indicates whether the rowset supports the IRowsetFind interface.
		UniOLEDB supports this property only if it is used with the Microsoft Cursor Engine service component.
		UniOLEDB: By default, the property is set to VARIANT_FALSE. It can be set to VARIANT_TRUE.
		R/W: Read/write.

Rowset Properties (Continued)

Property	Variant Type	Description
DBPROP_IROWSETIDENTITY	VT_BOOL	**General:** The IRowsetIdentity property indicates whether the rowset supports the IRowsetIdentity interface.
		UniOLEDB supports this property only if it is used with the Microsoft Cursor Engine service component.
		UniOLEDB: By default, the property is set to VARIANT_FALSE. It can be set to VARIANT_TRUE.
		R/W: Read/write.
DBPROP_IROWSETINFO	VT_BOOL	**General:** The IRowsetInfo property indicates whether the rowset supports the IRowsetInfo interface.
		UniOLEDB: The property is set to VARIANT_TRUE.
		R/W: Read only.
DBPROP_IROWSETLOCATE	VT_BOOL	**General:** The IRowsetLocate property indicates whether the rowset supports the IRowsetLocate interface.
		UniOLEDB supports this property only if it is used with the Microsoft Cursor Engine service component.
		UniOLEDB: By default, the property is set to VARIANT_FALSE. It can be set to VARIANT_TRUE.
		R/W: Read/write.

Rowset Properties (Continued)

Property	Variant Type	Description
DBPROP_IROWSETSCROLL	VT_BOOL	**General:** The IRowsetScroll property indicates whether the rowset supports the IRowsetScroll interface.
		UniOLEDB supports this property only if it is used with the Microsoft Cursor Engine service component.
		UniOLEDB: By default, the property is set to VARIANT_FALSE. It can be set to VARIANT_TRUE.
		R/W: Read/write.
DBPROP_IROWSETUPDATE	VT_BOOL	**General:** The IRowsetUpdate property indicates whether the rowset supports the IRowsetUpdate interface.
		UniOLEDB supports this property only if it is used with the Microsoft Cursor Engine service component.
		UniOLEDB: By default, the property is set to VARIANT_FALSE. It can be set to VARIANT_TRUE.
		R/W: Read/write.
DBPROP_ISUPPORTERRORINFO	VT_BOOL	**General:** The ISupportErrorInfo property indicates whether the rowset supports the ISupportErrorInfo interface.
		UniOLEDB: By default, the property is set to VARIANT_TRUE. It can be set to VARIANT_FALSE.
		R/W: Read/write.

Rowset Properties (Continued)

Property	Variant Type	Description
DBPROP_LITERALBOOKMARKS	VT_BOOL	**General:** The Literal Bookmarks property indicates whether bookmarks can be compared literally (compared as a sequence of bytes).
		UniOLEDB: The property is set to VARIANT_FALSE, which means that bookmarks can be compared only by using IRowsetLocate::Compare.
		R/W: Read only.
DBPROP_LITERALIDENTITY	VT_BOOL	**General:** The Literal Row Identity property indicates whether the consumer can compare two row handles by using a binary comparison to determine whether they point to the same row.
		UniOLEDB: The property is set to VARIANT_TRUE.
		R/W: Read only.
DBPROP_MAXOPENROWS	VT_I4	**General:** The Maximum Open Rows property indicates the maximum number of rows that can be active concurrently.
		UniOLEDB: The property is set to zero (0), which means that no limit exists for the number of rows that can be active concurrently.
		R/W: Read only.

Rowset Properties (Continued)

Property	Variant Type	Description
DBPROP_MAXPENDINGROWS	VT_I4	**General:** The Maximum Pending Rows property indicates the maximum number of rows that can have pending changes concurrently.
		UniOLEDB: The property is set to zero (0) because UniOLEDB does not support pending changes.
		R/W: Read only.
DBPROP_MAXROWS	VT_I4	**General:** The Maximum Rows property indicates the maximum number of rows that can be returned in a rowset.
		UniOLEDB: The property is set to zero (0), which means that no limit exists on the number of rows that can be returned in a rowset.
		R/W: Read only.
DBPROP_MEMORYUSAGE	VT_I4	**General:** The Memory Usage property estimates the amount of memory the rowset can use.
		UniOLEDB: The property is set to zero (0), which means that there are no specific limits on memory for a rowset to use.
		R/W: Read only.

Rowset Properties (Continued)

Property	Variant Type	Description
DBPROP_ ORDEREDBOOKMARKS	VT_BOOL	**General:** The Bookmarks Ordered property indicates whether the provider can compare bookmarks to determine the relative position of their associated rows in the rowset.
		UniOLEDB supports this property only if it is used with the Microsoft Cursor Engine service component.
		UniOLEDB: The property is set to VARIANT_FALSE, which means that UniOLEDB can compare bookmarks for equality only. The value of the DBPROP_LITERALBOOKMARKS property determines whether UniOLEDB can compare bookmarks byte-by-byte or must compare them by using IRowsetLocate::Compare.
		R/W: Read only.
DBPROP_OTHERINSERT	VT_BOOL	**General:** The Others' Inserts Visible property indicates whether the rowset can see rows inserted by a consumer other than the consumer of the rowset.
		UniOLEDB supports this property only if it is used with the Microsoft Cursor Engine service component.
		UniOLEDB: The property is set to VARIANT_FALSE.
		R/W: Read only.

Rowset Properties (Continued)

Property	Variant Type	Description
DBPROP_ OTHERUPDATEDELETE	VT_BOOL	**General:** The Others' Changes Visible property indicates whether the rowset can see updates or deletions made by a consumer other than the consumer of the rowset.
		UniOLEDB supports this property only if it is used with the Microsoft Cursor Engine service component.
		UniOLEDB: The property is set to VARIANT_FALSE.
		R/W: Read only.
DBPROP_OWNINSERT	VT_BOOL	**General:** The Own Inserts Visible property indicates whether a rowset can see its own inserts. This means that if a consumer of a rowset inserts a row, any consumer of the rowset can see the inserted row the next time it fetches a set of rows that contain it.
		UniOLEDB supports this property only if it is used with the Microsoft Cursor Engine service component.
		UniOLEDB: The property is set to VARIANT_FALSE.
		R/W: Read only.

Rowset Properties (Continued)

Property	Variant Type	Description
DBPROP_OWNUPDATEDELETE	VT_BOOL	**General:** The Own Changes Visible property indicates whether the rowset can see its own updates or deletions. This means that if a consumer of the rowset updates or deletes a row, any consumer of the rowset can see the update or deletion the next time it fetches that row. UniOLEDB supports this property only if it is used with the Microsoft Cursor Engine service component. **UniOLEDB:** The property is set to VARIANT_FALSE. **R/W:** Read only.
DBPROP_QUICKRESTART	VT_BOOL	**General:** The Quick Restart property indicates whether the provider is able to quickly restart the next fetch position (which means IRowset::RestartPosition executes relatively quickly) and does not require the rowset to be created again. **UniOLEDB:** The property is set to VARIANT_FALSE. **R/W:** Read only.
DBPROP_ RETURNPENDINGINSERTS	VT_BOOL	**General:** The Return Pending Inserts property indicates whether methods that fetch rows can return pending insert rows. **UniOLEDB:** The property is set to VARIANT_FALSE. **R/W:** Read only.

Rowset Properties (Continued)

Property	Variant Type	Description
DBPROP_ROWRESTRICT	VT_BOOL	**General:** The Row Privileges property indicates whether the provider restricts access to specific rows of a rowset.
		UniOLEDB: The property is set to VARIANT_FALSE, which means UniOLEDB allows a consumer to set data values for any row.
		R/W: Read only.
DBPROP_ROWTHREADMODEL	VT_I4	**General:** The Row Threading Model property indicates the threading model of the rowsets the command creates.
		UniOLEDB: The property is set to DBPROPVAL_RT_FREETHREAD, which means the rowsets support the free-threading model.
		R/W: Read only.

Rowset Properties (Continued)

Property	Variant Type	Description
DBPROP_SERVERCURSOR	VT_BOOL	**General:** The Server Cursor property indicates where the cursor materializes.
		UniOLEDB: The property is set to VARIANT_TRUE, which means UniOLEDB always uses server cursors.
		R/W: Read only.
DBPROP_STRONGIDENTITY	VT_BOOL	**General:** The Strong Row Identity property indicates whether the handles of newly inserted rows can by compared successfully as DBPROP_ LITERALIDENTITY specifies. A newly inserted row is one for which an insertion has been transmitted to the data source (not a pending insert row).
		UniOLEDB: The property is set to VARIANT_FALSE. The IRowsetIdentity::IsSameRow interface could return DB_E_NEWLYINSERTED.
		R/W: Read only.

Rowset Properties (Continued)

Property	Variant Type	Description
DBPROP_UPDATABILITY	VT_I4	**General:** The Updatability property indicates the methods the provider supports on the IRowsetChange interface.
		UniOLEDB supports this property only if it is used with the Microsoft Cursor Engine service component.
		UniOLEDB: By default, the property is set to zero (0) if DBPROP_ IRowsetChange is set to VARIANT_ FALSE. If DBPROP_IRowsetChange is set to VARIANT_TRUE, this property is set to DBPROPVAL_UP_CHANGE, which means that UniOLEDB supports IRowsetChange::SetData.
		R/W: Read/write.

Rowset Properties (Continued)

Session

The following table describes the properties UniOLEDB supports in the Session property group.

Property	Variant Type	Description
DBPROP_SESS_ AUTOCOMMITISOLEVELS	VT_I4	**General:** The Autocommit Isolation Levels session property indicates the transaction isolation level the provider supports while in auto-commit mode. n **UniOLEDB:** By default, the property is set to include both of the following DBPROPVAL_TI_ values: n READCOMMITTED n CURSORSTABILITY It can be set to any combination of the following DBPROPVAL_TI_ values: n READUNCOMMITTED n BROWSE n CURSORSTABILITY n READCOMMITTED n REPEATABLEREAD n SERIALIZABLE n ISOLATED **R/W:** Read/write.

Session Properties

Data Type Support

UniOLEDB represents UniData and UniVerse SQL data by using specific OLE DB data types in rowsets and parameters. This section indicates the data types and conversions UniOLEDB supports.

Supported Data Types

The following table shows the OLE DB data types UniOLEDB supports, and it shows how they map to native UniData and UniVerse SQL data types. A dash (–) indicates that a native UniData or UniVerse SQL data type does not exist for the OLE DB data type. For UniData, the conversion code used for converting to the SQL data type is shown in parentheses.

OLE DB Data Type UniOLEDB Supports	UniData SQL Data Type (Conversion Code)	UniVerse SQL Data Type
DBTYPE_I4	INTEGER (MD0)	INT[EGER]
DBTYPE_R4	–	REAL
DBTYPE_R8	FLOAT (MDn)	FLOAT, REAL, DOUBLE PRECISION
DBTYPE_STR	VARCHAR (*no conversion code*)	CHAR, VARCHAR
DBTYPE_DBTIME	TIME (MT)	TIME
DBTYPE_DBDATE	DATE (D)	DATE
DBTYPE_NUMERIC	–	NUMERIC
DBTYPE_DECIMAL	–	DEC[IMAL]

Supported Data Types and UniOLEDB Mappings

The supported OLE DB data types, along with their characteristics, are identified in the PROVIDER_TYPES rowset.

Data Type Conversions

UniOLEDB supports the following data type conversions:

- Conversions in the default types used for columns and parameters as returned by IColumnsInfo::GetColumnInfo or ICommandWithParameters::GetParameterInfo.

- Conversions to and from DBTYPE_WSTR if the conversion is defined by OLE DB.

UniOLEDB uses the OLE DB Data Conversion Library (MSDADC.DLL) for conversions.

The consumer can determine the specific conversions UniOLEDB supports by calling IConvertType::CanConvert and indicating the source and target data types.

Error Support

This section provides information about how UniOLEDB supports error handling. UniOLEDB complies with OLE DB error-handling standards. For more detailed information about error issues presented in this section, see the *Microsoft OLE DB 2.0 Programmer's Reference and Software Development Kit*.

Method Return Codes

All UniOLEDB methods return codes that indicate the success or failure of the method. All UniOLEDB return codes are of the type HRESULT, which means that return codes are in a bit-packed structure. The return codes are divided into two classes:

- Success and warning codes
- Error codes

Success and Warning Codes

UniOLEDB uses success and warning codes to indicate that a method completed successfully or that an error occurred from which the method was able to recover. These codes begin with S_ or DB_S_. If a method succeeds completely, the method returns the code S_OK. If one warning condition occurs, the method returns the appropriate warning code for that condition. If multiple warning conditions occur, the method decides which warning code to return.

Error Codes

UniOLEDB uses error codes to indicate that a method did not complete successfully and was unable to accomplish the task. These codes begin with E_ or DB_E_. If one error condition occurs, the method returns the appropriate error code for that condition. If a failure-specific error code is not available, the method returns the code E_FAIL. If a method generates both errors and warnings, the method fails and returns an error code.

Arrays of Errors

Some methods perform tasks on multiple items or arrays of items. If such a method successfully processes at least one item, it returns the code DB_S_ERRORSOCCURRED. If a method does not successfully process any item among the multiple items or in the array, it returns the code DB_E_ERRORSOCCURRED.

A consumer also can request that these methods return arrays of DBROWSTATUS values, which provide information about the specific warnings and errors that occurred.

OLE DB Error Objects

UniOLEDB objects support the ISupportErrorInfo interface if they expose an interface that can return OLE DB error objects. When any method fails, UniOLEDB creates error objects to describe the error. Each error object consists of one or more error records, which contain error information such as the code returned by the method, the class ID of the object that returned the error, and the ID of the interface that generated the error.

When a new method starts, UniOLEDB uses the SetErrorInfo function to clear any error object on the thread that was created by a previous method. UniOLEDB does not expose the IErrorInfo interface, which retrieves information about the error object itself.

UniOLEDB does not support the ISQLErrorInfo interface because it returns information about the error severity and state of custom error objects, which UniOLEDB does not implement.

Error Lookup Service

UniOLEDB implements an error lookup service that exposes the IErrorLookup interface. The service retrieves and stores error information, such as error messages and help file information, from a specific OLE DB error object.

Transaction and SQL Support

This section provides information about how UniOLEDB supports transactions and SQL. For more detailed information about transaction issues presented in this section, see the *Microsoft OLE DB 2.0 Programmer's Reference and Software Development Kit*.

Transaction Support

UniOLEDB supports one local level of transaction scope per session. Advanced transaction features, such as nested and coordinated transactions, are not supported.

UniOLEDB supports all OLE DB transaction isolation levels except ISOLATIONLEVEL_ CHAOS. Concurrent consumers can control transaction isolation levels for a connection by using:

- The DBPROP_SESS_AUTOCOMMITISOLEVELS session property, which indicates the transaction isolation levels UniOLEDB will support for auto-commit transactions. By default, UniOLEDB supports both READCOMMITTED and CURSORSTABILITY levels.
- The *isoLevel* parameter of ITransactionLocal::StartTransaction for local manual-commit transactions.

The ITransactionLocal::StartTransaction interface begins a local manual-commit transaction, and the transaction ends when a consumer calls either the ITransaction::Commit or the ITransaction::Abort interface.

SQL Support

Command objects can use SQL statements to create rowsets and manipulate data. Any SQL statement the UniData or UniVerse engine (whichever one to which you are connected) supports can be executed as a command object. In particular, both UniData and UniVerse support at least the ANSI92_ENTRY level definition of SQL.

Note: UniOLEDB does not support data definition language (DDL) statements when connected to UniData.

UniOLEDB also supports parameters in SQL statement commands. For commands that call procedures, UniOLEDB supports input, output, and input/output parameters. When a consumer application executes a procedure call, output parameters are returned to the application. For UniVerse, a procedure can return rowsets in addition to parameters. In this case, output parameters are returned after the application consumes all returned rowsets.

Examples of Consumer Source Code

This appendix provides examples of consumer source code written by using C++ and ADO/ASP.

Example of C++ Source Code

The following program example shows the source code for a consumer developed by using C++. The comments included in the example describe the tasks the consumer performs.

```cpp
// smaple.cpp

/*
    Sample consumer for UniOLEDB

Application flow in OLE DB is similar to the application flow in
ODBC.
In both cases, the application:

    1. Initializes OLE.
    2. Connects to a data source object.
    3. Creates and executes a command.
    4. Processes the results.
    5. Releases objects and uninitializes OLE.

*/


#include <iostream.h>

#include <windows.h>            // MFC core and standard components

#include <oledb.h>      // OLE DB include files
#include <oledberr.h>


// Global task memory allocator
IMalloc*        g_pIMalloc = NULL;

// connection information
#define C_DATASOURCE    OLESTR("udunix")
#define C_LOCATION              OLESTR("oledbtest_ud")
#define C_USERNAME              OLESTR("unidata")
#define C_PASSWORD              OLESTR("******")

int main()
{
    IDBInitialize *         pIDBInitialize = NULL;
    IRowset *               pIRowset = NULL;
        const ULONG                 nProps = 5;
    IDBProperties *         pIDBProperties;
    DBPROP                          InitProperties[nProps];
    DBPROPSET               rgInitPropSet;
    HRESULT                         hr;
        ULONG                       i;
```

Example of C++ Source Code

```
        /*
        In OLE DB, initialization of the environment is achieved
by a call to
        OleInitialize, which initializes the OLE library. This is
shown in the
        preceding code example. After the OLE library is
initialized, the proper
        data provider is loaded by the system according to its
class ID, and
        calls are made directly to the provider.
        */

        HINSTANCE hInst= ::GetModuleHandle(NULL);

        // Init OLE and set up the DLLs
    CoInitialize(NULL);

    // Get the task memory allocator.
    if (FAILED(CoGetMalloc(MEMCTX_TASK, &g_pIMalloc)))
         goto EXIT;

        /*
        The data source object exposes the IDBInitialize and
IDBProperties
        interfaces that contain the methods to connect to a data
source. The
        authentication information such as user ID, password, and
the name of the
        data source are specified as properties of the data source
object by
        calling IDBProperties::SetProperties. The method
        IDBInitialize::Initialize uses the specified properties to
connect to the
        data source.
        */

    // We have the file name, lets get the CLSID for the Check
Book Data
    // Provider so that an instance can be created.
        CLSID CLSID_UniOLEDB;

    if (CLSIDFromProgID(OLESTR("Ardent.UniOLEDB"),
&CLSID_UniOLEDB) != S_OK)
        {
        cerr << "ERROR. Unable to obtain class ID of UniOLEDB
provider." << endl;
        goto EXIT;
        }

      // Create an instance of the UniOLEDB provider
   CoCreateInstance(CLSID_UniOLEDB, NULL, CLSCTX_INPROC_SERVER,
                          IID_IDBInitialize,
(void**)&pIDBInitialize);
```

```
   if (pIDBInitialize == NULL)
       {
         cerr << "ERROR. Failed to create UniOLEDB provider
instance." << endl;
         goto EXIT;
       }

       /*
       Getting and Setting Properties

       Properties are used in OLE DB to specify options, such as
initialization
       information on the data source object or supported
properties of a rowset,
       as well as to discover properties of certain objects, such
as the
       updatability of a rowset. Properties in OLE DB are similar
to the
       environment, connection, and statement attributes in ODBC,
with the
       following exceptions:

       In OLE DB, the provider can be queried for a list of all
supported
       properties. In OLE DB, properties are grouped into "Property
Groups."
       Property groups are identified by a GUID. This allows third
parties to
       define properties within their own property group, rather
than trying to
       reserve ranges within a single set of attribute values.

       Instead of setting and retrieving properties individually,
multiple
       properties can be set or retrieved from multiple groups in a
single call.
       This is done by building an array of property sets, where
each property set
       contains an array of property structures from a single
property group.

       Set or retrieve properties on a data source using
IDBProperties.
       Set or retrieve properties on a session using
ISessionProperties.
       Set or retrieve properties on a command using
ICommandProperties.
       Retrieve properties and information about a rowset using
IRowsetInfo.

       The following table shows the property groups in OLE DB and
their GUIDs.
```

Example of C++ Source Code

```
        Property groupProperty group identifier (GUID)
        ---------------------------------------------
        Column      DBPROPFLAGS_COLUMN
        Data Source DBPROPFLAGS_DATASOURCE
        Data Source CreationDBPROPFLAGS_DATASOURCECREATE
        Data Source InformationDBPROPFLAGS_DATASOURCEINFO
        Data Source InitializationDBPROPFLAGS_DBINIT
        Index       DBPROPFLAGS_INDEX
        Rowset      DBPROPFLAGS_ROWSET
        Session     DBPROPFLAGS_SESSION
        Table       DBPROPFLAGS_TABLE

    The following structure contains an array of values of
properties from a
    single property set:

    typedef struct  tagDBPROPSET
  {
            DBPROP __RPC_FAR* rgProperties;     // Pointer to an
array of
                                                // DBPROP
structures.
            ULONG             cProperties;      // Count of
properties
                                                // (DBPROPS) in
the array.
            GUID              guidPropertySet;  // A GUID that
identifies the
                                                // property set
to which the
                                                // properties
belong.
  } DBPROPSET;

    The following structure contains information about a single
property:
    typedef struct  tagDBPROP
  {
            DBPROPID          dwPropertyID;     // ID of
property within a
                                                // property set.
            DBPROPOPTIONS     dwOptions;        // Property is
required?
                                                // Optional?
            DBPROPSTATUS      dwStatus;         // Status
returned by the
                                                // provider
indicating success
                                                // or failure in
setting or
                                                // getting the
property.
                                                // Enumerated
values are:
```

```
                                                      //
           DBPROPSTATUS_OK
                                                      //
           DBPROPSTATUS_NOTSUPPORTED
                                                      //
           DBPROPSTATUS_BADVALUE
                                                      //
           DBPROPSTATUS_BADOPTION
                                                      //
           DBPROPSTATUS_BADCOLUMN
                                                      //
           DBPROPSTATUS_NOTALLSETTABLE
                                                      //
           DBPROPSTATUS_NOTSET
                                                      //
           DBPROPSTATUS_NOTSETTABLE
                                                      //
           DBPROPSTATUS_CONFLICTING
                DBID            colid;        // Optional,
      ordinal column
                                             // property
      applies to. If the
                                             // property
      applies to all
                                             // columns, colid
      should be set
                                             // to DB_NULLID.
                VARIANT         vValue;       // Value of the
      property.
         } DBPROP;
           */

           /*
           Application sets initialization properties on a data source
      object. The
           code sets four properties within a single property group.
      The general
           flow of control is:
                   1. Allocate an array of property structures.
                   2. Allocate an array of a single property set.
                   3. Initialize common property elements for the
      properties.
                   4. Fill in the following properties:
                           · Level of desired prompting (similar to
      DriverCompletion
                            argument in SQLDriverConnect)
                           · Data source name (similar to DSN=
      element of the ODBC
                            connection string)
                           · User name (similar to the UID= element
      of the ODBC
                            connection string)
                           · Password (similar to the PWD= element of
      the ODBC
```

Example of C++ Source Code

```
                                        connection string)
                    5. Set the property set to the array of properties
        and specify
                       that the properties are from the initialization
        property group.
                    6. Get the IDBProperties interface.
                    7. Call SetProperties on the interface.
            */


            // Initialize common property options.
        for (i = 0; i < nProps; i++ )
            {
                    VariantInit(&InitProperties[i].vValue);
             InitProperties[i].dwOptions = DBPROPOPTIONS_REQUIRED;
             InitProperties[i].colid = DB_NULLID;
             }

        // Level of prompting that will be done to complete the
        // connection process
        InitProperties[0].dwPropertyID = DBPROP_INIT_PROMPT;
        InitProperties[0].vValue.vt = VT_I2;
        InitProperties[0].vValue.iVal = DBPROMPT_NOPROMPT;

        // Data source name
        InitProperties[1].dwPropertyID = DBPROP_INIT_DATASOURCE;
        InitProperties[1].vValue.vt = VT_BSTR;
        InitProperties[1].vValue.bstrVal =
    SysAllocString(C_DATASOURCE);

        // User ID
        InitProperties[2].dwPropertyID = DBPROP_AUTH_USERID;
        InitProperties[2].vValue.vt = VT_BSTR;
        InitProperties[2].vValue.bstrVal = SysAllocString(C_USERNAME);

        // Password
        InitProperties[3].dwPropertyID = DBPROP_AUTH_PASSWORD;
        InitProperties[3].vValue.vt = VT_BSTR;
        InitProperties[3].vValue.bstrVal = SysAllocString(C_PASSWORD);

            // Location (Account Name)
        InitProperties[4].dwPropertyID = DBPROP_INIT_LOCATION;
        InitProperties[4].vValue.vt = VT_BSTR;
        InitProperties[4].vValue.bstrVal = SysAllocString(C_LOCATION);

        rgInitPropSet.guidPropertySet = DBPROPSET_DBINIT;
        rgInitPropSet.cProperties = nProps;
        rgInitPropSet.rgProperties = InitProperties;

        // Set initialization properties.
        pIDBInitialize->QueryInterface(IID_IDBProperties, (void**)
    &pIDBProperties);
        hr = pIDBProperties->SetProperties(1, &rgInitPropSet);
```

```
        SysFreeString(InitProperties[1].vValue.bstrVal);
        SysFreeString(InitProperties[2].vValue.bstrVal);
        SysFreeString(InitProperties[3].vValue.bstrVal);

        pIDBProperties->Release();

        if (FAILED(hr))
        {
                cerr << "ERROR. Set properties failed." << endl;
                goto EXIT;
        }

            // call method IDBInitialize::Initialize to connect to
            // the data source.
        if (FAILED(pIDBInitialize->Initialize()))
            {
                cerr << "ERROR. IDBInitialze->Initialize() failed."
<< endl;
                goto EXIT;
            }

            cout << "Connect OK." << endl;

EXIT:
    // Clean up and disconnect.
    if (pIRowset != NULL)
         pIRowset->Release();

    if (pIDBInitialize != NULL)
         {
         pIDBInitialize->Uninitialize();
         pIDBInitialize->Release();
         }

    if (g_pIMalloc != NULL)
         g_pIMalloc->Release();

    CoUninitialize();
    return (0);
}
```

Example of ADO/ASP Source Code

The following program example shows the source code for a consumer developed by using ADO/ASP. The comments included in the example describe the tasks the consumer performs.

```
/*
HTML code below displays a form that collects connection data
from user.  When user hits the <submit> button, the asp page
"test1_do.asp" is executed to generate a results page from
the query.
*/
// FILE: test1.asp

<%@ Language=VBScript %>
<HTML>
<HEAD>
<META NAME="GENERATOR" Content="Microsoft Visual Studio 6.0">
</HEAD>
<BODY>
<FORM action="test1_do.asp" id=FORM1 method=post name=FORM1>
<P><FONT color=seagreen face="" size=5>SQL Execute</FONT></P>
<P>
Location:      
<INPUT id=location name=location><BR>Data Source:
<INPUT id=datasource name=datasource><BR>User Name:  
<INPUT id=username
name=username><BR>Password:    
<INPUT id=password name=password type=password><BR></P>
<P>
SQL:<BR> <TEXTAREA id=sql name=sql style="HEIGHT: 127px; WIDTH:
470px">
</TEXTAREA></P>
<P><INPUT id=Submit name=Submit type=submit value=Submit></P>
<P> </P>
<P></P>
<P> </P></FORM>

</BODY>
</HTML>



/*
Here is the ASP page, test1_do.asp.  This page executes a query in
response to user hitting <submit> button from the above HTML form.
*/
// FILE: test1_do.asp

<%@ Language=VBScript %>
<% ' VI 6.0 Scripting Object Model Enabled %>
```

```
<!--#include file="_ScriptLibrary/pm.asp"-->
<% if StartPageProcessing() Then Response.End() %>
<FORM name=thisForm METHOD=post>
<HTML>
<HEAD>
<META NAME="GENERATOR" Content="Microsoft Visual Studio 6.0">
</HEAD>
<BODY>
<script LANGUAGE="JavaScript" RUNAT = "server">
function do_test()
{
   // read form fields
   var location1 = Request.Form("location");
   var datasource1 = Request.Form("datasource");
   var username1 = Request.Form("username");
   var password1 = Request.Form("password");
   var sql1 = Request.Form("sql");

   // write back form fields to response page.
   Response.Write("INPUTS:<BR><BR>");
   Response.Write("Location:" + location1 + "<BR>");
   Response.Write("Data source:" + datasource1 + "<BR>");
   Response.Write("Username:" + username1 + "<BR>");
   Response.Write("Attempting connection...<BR>");

   // create a connection using UniOLEDB Provider
   var Conn = Server.CreateObject("ADODB.Connection");
   Conn.Provider = "Ardent.UniOLEDB";
   var ConStr = "DATA SOURCE="+datasource1+";
           LOCATION="+location1+"; USER ID="+username1+";
           PASSWORD="+password1+";";
   Conn.Open(ConStr);
   Response.Write("Connect OK<BR>");
   Response.Write("Executing SQL...<BR>");

   // execute SQL Statement
   var RecordsAffected = 0;
   var Rs = Conn.Execute(sql1);

   // write field names to response page
   for(i = 0; i < Rs.Fields.Count; ++i)
   {
      Response.Write(Rs.Fields(i).Name);
      Response.Write(" ");
   }
   Response.Write("<BR>");

   // write data to response page
   while(!Rs.eof)
   {
      for(i = 0; i < Rs.Fields.Count; ++i)
      {
         Response.Write(Rs.Fields(i).Value);
         Response.Write(" ");
```

Example of ADO/ASP Source Code

```
            }
            Response.Write("<BR>");
            Rs.MoveNext();
        }

        // Close record set
        Rs.Close();

        // Close connection
        Conn.Close();

        Response.Write("DisConnect OK<BR>");
    }

    do_test();
    </script>
    <P> </P>

    </BODY>
    <% ' VI 6.0 Scripting Object Model Enabled %>
    <% EndPageProcessing() %>
    </FORM>
    </HTML>
```

Glossary

1NF	First normal form.
1NF database	A database that conforms to the 1NF type of data storage. Most databases are 1NF databases; as such, they have only singlevalued attributes. Oracle is a 1NF database. See also *first normal form (1NF)*.
1NF table	A first-normal-form table. See also *first normal form (1NF)*.
1NF mapping	A mechanism that enables data stored in UniData and UniVerse files to be viewed (read-only mapping) or updated (updatable mapping) by applications that operate on 1NF data. The applications assume that the data conforms to the 1NF model in which all attribute values are atomic. See also *first normal form (1NF)*.
accessor	A collection of information that describes how to transfer data to and from a consumer's buffer. See also *consumer*.
Active Server Pages (ASP)	A scripting environment in which you can combine HTML, scripts, and reusable ActiveX server components to create web pages.
ActiveX Data Objects (ADO)	An application-level interface that can be used to access diverse types of data regardless of how the data is structured. ADO can function as an interface to OLE DB, and OLE DB provides the system-level interfaces for uniform data access. Together with OLE DB and ODBC, ADO is a main component of Microsoft's Universal Data Access (UDA) specification. See also *interface*, *OLE DB*, and *Universal Data Access (UDA)*.
ADO	ActiveX Data Objects.

American National Standards Institute (ANSI)	A United States organization charged with developing national information technology standards for programming languages, electronic data interchange (EDI), telecommunications, and the physical characteristics of various data storage media.
ANSI	American National Standards Institute.
API	Application programming interface.
application program	A user program that issues function calls to submit statements and retrieve results, and then processes those results.
application programming interface (API)	A set of function calls that provide services to application programs.
ASP	Active Server Pages.
association	A group of related multivalued columns in a table. The first value in any association column corresponds to the first value of every other column in the association, the second value corresponds to the second value, and so forth. An association can be thought of as an extended table. See also *multivalued columns*.
association key	The values in one or more columns of an association that uniquely identify each row in the association. If an assoiation does not have keys, the @ASSOC_ROW keyword can generate unique association row identifiers. See also *association* and *key*.
association row	A sequence of related data values in an association. A row in an extended table. See also *association* and *row*.
binary large object (BLOB)	A collection of complex binary data (such as digitized audio, video, image, and sound information) stored in a database.
BLOB	Binary large object.
client	A computer system or program that uses the resources and services of another system or program (called a server). See also *server*. A process that uses resources provided by a local or remote server process.
column	A vertical row of cells that contain the same kind of information, such as names or phone numbers. A column is another name for field. See also *row* and *table*.
catalog	An object that contains one or more schemas. See also *object* and *schema*.

COLUMNS rowset	An OLE DB rowset that includes columns in all UniOLEDB-visible tables. For UniVerse tables, columns in the COLUMNS rowset are defined by the dictionary's @SELECT phrase, or are defined as the columns created by CREATE TABLE if there is no @SELECT phrase. For files and for associations and unassociated multi-valued columns within them, columns in the COLUMNS rowset are defined by the dictionary's @SELECT phrase, or by the @ phrase if the @SELECT phrase does not exist. If neither an @SELECT phrase nor an @ phrase exists, only the record ID (@ID) appears in the COLUMNS rowset. See also *association*, *column*, *multivalued column*, *rowset*, *table*, *UniOLEDB-visible table*, and *UniVerse table*.
COM	Component Object Model.
COM object	An object that complies with the Component Object Model (COM). See also *Component Object Model (COM)* and *object*.
command object	An OLE DB object that encapsulates a command. See also *encapsulation* and *OLEDB object*.
component	An object or program that performs a specific task and is designed to interact easily with other components or applications. For example, the OLE DB architecture consists of consumers, providers, and service components that interoperate based on common interfaces. See also *consumer*, *provider*, and *service component*.
Component Object Model (COM)	A component architecture that enables programmers to develop objects that are accessible by any COM-compliant application. COM enables objects to communicate with other objects. See also *component* and *object*.
consumer	An application that calls OLE DB interfaces to retrieve and manipulate data. See also *interface*, *provider*, and *service component*.
cursor	A virtual pointer to the set of result rows produced by a query. A cursor points to the "current row" of the result set, one row of data at a time, and advances one row at a time. See *query* and *row*.
data definition language (DDL)	A language used to define data and its relationships to other data. This information is used to create the data structure in a database.
data manipulation language (DML)	A language used to store, retrieve, modify, and erase data from a database.
data provider	A provider that exposes data directly. See also *consumer* and *service component*.

data source	The data a consumer wants to access. Contrast with *data source object*.
data source object	An OLE DB object that connects to a data source. Contrast with *data source*. See also *OLE DB object*.
data type	A classification of a particular type of information (for example, date, decimal, numeric, or string).
database	A collection of information that conforms to a common structure with a format defined by the metadata. It is operated on (stored, modified, retrieved) by a database management system, and it can be concurrently accessed by multiple users under the protection of data sharing and concurrency rules provided by the DBMS. See also *metadata* and *database management system (DBMS)*.
database management system (DBMS)	A software system that enables you to create, store, organize, modify, and extract information in a database. It also controls access to the information. See also *database*, *relational database management system (RDBMS)* and *extended relational database*.
DBMS	Database management system.
DDL	Data definition language.
DLL	Dynamic link library.
DML	Data manipulation language.
dynamic link library (DLL)	A collection of functions linked together into a unit that can be distributed to application developers. When the program runs, the application attaches itself to the DLL when the program calls one of the DLL functions.
dynamic normalization	A UniVerse mechanism that lets DML statements access an association of multivalued columns or an unassociated multivalued column as a 1NF table. See also *1NF table*, *association*, *data manipulation language (DML)*, *first normal form (1NF)*, *multivalued column*, and *normalization*.
encapsulation	The process of combining data and methods, which manipulate the data, to create an object. An object's data can be hidden from external methods. See also *method* and *object*.
enumerator object	An OLE DB object that searches for data sources and other enumerators. See also *OLE DB object*.
error object	An object that contains detailed information about an error. See also *OLE DB error object* and *OLE DB object*.

extended relational database	A database that uses a three-dimensional file structure that supports multivalued or multi-subvalued data within extended tables, and extensible, variable-length data formats. This enables a single file (table) to contain the information that otherwise would be scattered among several interrelated files in a 1NF database. See also *database*, *database management system (DBMS)*, and *relational database management system (RDBMS)*.
field	See *column*.
first normal form (1NF)	The name of a kind of relational database that can have only one value for each row and column position (or cell). Contrast with *nonfirst-normal form (NF2)*.
graphical user interface (GUI)	A user interface that enables users to interact with a computer application by using images and text. See also *interface*.
GUI	Graphical user interface.
handle	A value that uniquely and unambiguously identifies an entity, such as a row or an accessor. See also *accessor* and *row*.
interface	The connection and interaction between hardware, software, and the user. In includes the languages, codes, and messages a program uses to communicate with other programs and hardware.
	A description of a set of possible uses of an object. It describes a set of requests to which an object can respond meaningfully. See also *object*.
Isolation level	A mechanism for separating a transaction from other transactions that run concurrently so that no transaction affects any of the others. See also *transaction*.
key	A data value used to locate a row. See also *row*.
metadata	Data that describes data, such as how it is collected and formatted.
method	Code that is executed against the object's data when the object receives a request. See also *object*.
multivalued column	A column that can contain more than one value for each row in a table. See also *column*, *multi-subvalued column*, and *singlevalued column*.
multi-subvalued column	A multivalued column in which each value in the column can contain additional values, called subvalues, for each row in a table. For illustrative purposes, consider a manufacturing plant that creates an assembled piece (value), which is made up of assemblies (multivalues) that consist of several parts (multi-subvalues). See also *column*, *multivalued column*, and *singlevalued column*.

nested transaction	A transaction that begins while another transaction is active. See also *transaction*.
NF^2	Nonfirst-normal form.
NF^2 database	A database that conforms to the extended relational model. This model enables you to store data in a variety of attributes: singlevalued, multivalued, and multi-subvalued. This model avoids data redundancy. UniData and UniVerse are NF^2 databases. Contrast with *1NF database*.
nonfirst-normal form (NF^2)	The name of a kind of relational database that can have more than one value for a row and column position (or cell). Contrast with *first normal form (1NF)*.
normalization	In relational database management, a process that organizes complex data into simple rows and columns in which each cell has one value only. Normalization usually involves dividing a table into two or more tables and defining a relationship between them. See also *first normal form (1NF)* and *nonfirst-normal form (NF2)*.
object	A self-contained entity that consists of data and the methods that manipulate the data when the object receives a request. See also *method*.
Object Linking and Embedding (OLE)	Microsoft's component architecture for object-oriented programming. See also *object*.
object-oriented programming	A type of programming that combines data structures with functions to create objects that are reusable. See also *object*.
ODBC	Open Database Connectivity.
OLE	Object Linking and Embedding.
OLE DB	A set of object-oriented interfaces that are based on the Component Object Model (COM). It consists of objects that encapsulate various aspects of traditional database functionality. OLE DB COM enables OLE DB components (consumers, providers, and service components) to interoperate and produce, share, and consume data in the form of rowsets. Together with ODBC and ADO, OLE DB is a main component of Microsoft's Universal Data Access (UDA) specification. See also *interface*, *component, Component Object Model (COM)*, and *Universal Data Access (UDA)*.
OLE DB error object	An error object used by OLE DB objects to return an error. See also *error object* and *OLE DB object*.

OLE DB object	A COM object that is defined by OLE DB. Examples of COM objects that are defined by OLE DB include enumerators, data source objects, commands, sessions, rowsets, and OLE DB error objects. See also *enumerator object*, *data source object*, *command object*, *session object*, *rowset object*, and *OLE DB error object*.
Open Database Connectivity (ODBC)	A Microsoft interface that defines a library of function calls that permit a client application program to connect to a data source, execute SQL statements against that source, and retrieve results. ODBC inserts a middle layer, called a database driver, between an application and the DBMS. This layer translates the application's data queries into commands that the DBMS understands. ODBC provides a standard set of error codes, a way to connect to the data source, and a standard set of data types. The ODBC specifications from Microsoft for SQL-based database interoperability cover both the application programming interface (API) and SQL grammar. UCI is modeled on this standard, but it is not ODBC compliant. Together with OLE DB and ADO, ODBC is a main component of Microsoft's Universal Data Access (UDA) specification. See also *interface* and *Universal Data Access (UDA)*.
primary key	The value in one or more columns that uniquely identifies each row in a table. See also *key*.
property	A characteristic of an object that describes a behavior. A property describes the attributes associated with a data structure. An attribute's value can be changed. See also *object*.
provider	A software component that uses OLE DB interfaces to expose data to which consumers request access. See also *component*, *consumer*, and *service component*.
qualifier	An identifier prefixed to the name of a column, table, or alias to distinguish names that would otherwise be identical. See also *column* and *table*.
query	A request for data from a database. See also *database*.
record	See *row*.
row	A set of information that contains columns, each of which contains an item of information. A set of rows make up a table. See also *column* and *table*.
rowset object	An OLE DB object that contains a set of rows, each of which has columns of data. See also *column*, *OLE DB object*, and *row*.
RDA	Remote data access.
RDBMS	Relational database management system.

relational database management system (RDBMS)	A type of DBMS that organizes and accesses information in a database in the form of related tables. The columns of a table represent attributes, and the rows represent tuples. See also *database*, *database management system (DBMS)*, *extended relational database*, and *table*.
remote procedure call (RPC)	UCI uses a library of calls developed by Ardent to implement remote procedure calls (UniRPC). An RPC allows a consumer to have a function executed on its behalf by another system (the server). The consumer passes arguments to the server as well as an identifier that specifies the procedure to be executed on the server. The server executes the procedure using the arguments passed to it, and then returns the results to the client. See also *client*, *consumer*, and *server*.
RPC	Remote procedure call.
schema	In SQL, a collection of database objects, such as tables, views, and indexes, that share the same namespace. See also *object*, *table*, and *view*.
Schema API	An application programming interface (API) that consists of a series of UniBasic subroutines designed to accomplish the same tasks (generating schema on UniData files) as Visual Schema Generator (VSG). VSG's interface makes it easier and faster to use than Schema API. See also *interface*, *schema*, and *Visual Schema Generator (VSG)*.
schema rowset	A predefined rowset that provides information about the structure of a database. See also *rowset object*.
server	A computer running software that offers resources to clients. See also *client*. A process that accepts and handles requests from a client process.
service component	A software component that provides extended functionality the provider does not support, such as advanced cursor and query processing features. Service components do not expose data directly, but provide services the data provider does not support. See also *data provider*.
service provider	See *service component*.
session	An OLE DB object that serves as the context for a transaction. See also *OLE DB object* and *transaction*.
singlevalued column	A column that can contain only one value for each row in a table. See also *column*, *multivalued column*, and *multi-subvalued column*.

Structured Query Language (SQL)	An industry standard for data management, data definition and manipulation, access protections, and transaction control. It provides a standardized query language for requesting information from a database. See also *query* and *transaction*.
SQL	Structured Query Language.
table	A matrix of data arranged in rows and columns. See also *column* and *row*.
TABLES rowset	An OLE DB rowset that includes all UniOLEDB-visible tables. See also *rowset* and *UniOLEDB-visible table*.
tuple	See *row*.
transaction	A strategy that treats a group of database operations as one unit. The database remains consistent because either all or none of the operations are completed.
transaction object	An OLE DB object used to support transactions. See also *OLE DB object*.
UCI	A C-language application programming interface (API) that enables application programmers to write client application programs that use SQL function calls to access data in UniData and UniVerse databases. UniOLEDB uses UCI to access data in UniData and UniVerse databases. See also *interface* and *UniOLEDB*.
uci.config file	The client UCI configuration file, which defines data sources to which an application can connect in terms of DBMS, network, service, host, and optional extended parameters. See also *data source*.
UCI data source	A source of data, or database engine, represented by the specifications supplied in the data source entry in the uci.config file. These specifications include the DBMS type, the network, the name of the service, and the host platform. See also *data source* and *uci.config file*.
UDA	Universal Data Access.
udserver process	A UniData server process that handles requests from the client. For each client connection to the server, there is one udserver process. See also *client* and *server*.
UniOLEDB	Ardent's OLE DB provider. See also *provider*.
UniOLEDB table	Any UniData or UniVerse table or file that is accessible to UniOLEDB. UniOLEDB tables are always 1NF tables. See also *1NF tables*, *first normal form (1NF)*, and *tables*.

UniOLEDB-visible table	Any UniOLEDB table that is included in the TABLES rowset. See also *rowset*, *table*, *TABLES rowset*, and *UniOLEDB table*.
UniRPC	A library of calls developed by Ardent to implement remote procedure calls. See also *remote procedure call (RPC)*.
unirpc service	On Windows platforms, the service that waits for a client's request to connect. When it receives a request, unirpcd daemon creates the connection to the server. See also *UniRPC*.
unirpcd daemon	On UNIX servers, the daemon that waits for a client's request to connect. When it receives a request, unirpcd creates the connection to the server. See also *UniRPC*.
Universal Data Access (UDA)	Microsoft's high-level specification for providing access to diverse forms of information across an enterprise. The main components of UDA include OLE DB, ODBC, and ADO. See also *OLE DB*, O*pen Database Connectivity (ODBC)*, and *ActiveX Data Objects (ADO)*.
UniVerse file	A file defined by the UniVerse command CREATE.FILE.
UniVerse table	A table defined by the CREATE TABLE statement. A view defined by the CREATE VIEW statement. A virtual table generated from a UniVerse table or file by dynamic normalization. See also *dynamic normalization*, *normalization*, and *table*.
uvserver process	A UniVerse server process that handles requests from the client. For each client connection to the server, there is one uvserver process. See also *client* and *server*.
view	A derived table created by a SELECT statement that is part of the view's definition. See also *table*.
Visual Schema Generator (VSG)	A windows-based graphical user interface (GUI) tool for generating schema on UniData files. The schema allow Ardent's ODBC driver or UniOLEDB provider to access data in UniData files, which the schema represent in 1NF subtables and views. See also *first normal form (1NF)*, *interface*, *schema*, *UniOLEDB*, and *view*.
VSG	Visual Schema Generator.

Index