LabVIEW™

Database Connectivity Toolkit User Manual

June 2008
371525A-01

NATIONAL INSTRUMENTS™

Worldwide Technical Support and Product Information

ni.com

National Instruments Corporate Headquarters

11500 North Mopac Expressway Austin, Texas 78759-3504 USA Tel: 512 683 0100

Worldwide Offices

Australia 1800 300 800, Austria 43 662 457990-0, Belgium 32 (0) 2 757 0020, Brazil 55 11 3262 3599, Canada 800 433 3488, China 86 21 5050 9800, Czech Republic 420 224 235 774, Denmark 45 45 76 26 00, Finland 358 (0) 9 725 72511, France 01 57 66 24 24, Germany 49 89 7413130, India 91 80 41190000, Israel 972 3 6393737, Italy 39 02 41309277, Japan 0120-527196, Korea 82 02 3451 3400, Lebanon 961 (0) 1 33 28 28, Malaysia 1800 887710, Mexico 01 800 010 0793, Netherlands 31 (0) 348 433 466, New Zealand 0800 553 322, Norway 47 (0) 66 90 76 60, Poland 48 22 3390150, Portugal 351 210 311 210, Russia 7 495 783 6851, Singapore 1800 226 5886, Slovenia 386 3 425 42 00, South Africa 27 0 11 805 8197, Spain 34 91 640 0085, Sweden 46 (0) 8 587 895 00, Switzerland 41 56 2005151, Taiwan 886 02 2377 2222, Thailand 662 278 6777, Turkey 90 212 279 3031, United Kingdom 44 (0) 1635 523545

For further support information, refer to the *Technical Support and Professional Services* appendix. To comment on National Instruments documentation, refer to the National Instruments Web site at ni.com/info and enter the info code feedback.

Important Information

Warranty

The media on which you receive National Instruments software are warranted not to fail to execute programming instructions, due to defects in materials and workmanship, for a period of 90 days from date of shipment, as evidenced by receipts or other documentation. National Instruments will, at its option, repair or replace software media that do not execute programming instructions if National Instruments receives notice of such defects during the warranty period. National Instruments does not warrant that the operation of the software shall be uninterrupted or error free.

A Return Material Authorization (RMA) number must be obtained from the factory and clearly marked on the outside of the package before any equipment will be accepted for warranty work. National Instruments will pay the shipping costs of returning to the owner parts which are covered by warranty.

National Instruments believes that the information in this document is accurate. The document has been carefully reviewed for technical accuracy. In the event that technical or typographical errors exist, National Instruments reserves the right to make changes to subsequent editions of this document without prior notice to holders of this edition. The reader should consult National Instruments if errors are suspected. In no event shall National Instruments be liable for any damages arising out of or related to this document or the information contained in it.

EXCEPT AS SPECIFIED HEREIN, NATIONAL INSTRUMENTS MAKES NO WARRANTIES, EXPRESS OR IMPLIED, AND SPECIFICALLY DISCLAIMS ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. CUSTOMER'S RIGHT TO RECOVER DAMAGES CAUSED BY FAULT OR NEGLIGENCE ON THE PART OF NATIONAL INSTRUMENTS SHALL BE LIMITED TO THE AMOUNT THERETOFORE PAID BY THE CUSTOMER. NATIONAL INSTRUMENTS WILL NOT BE LIABLE FOR DAMAGES RESULTING FROM LOSS OF DATA, PROFITS, USE OF PRODUCTS, OR INCIDENTAL OR CONSEQUENTIAL DAMAGES, EVEN IF ADVISED OF THE POSSIBILITY THEREOF. This limitation of the liability of National Instruments will apply regardless of the form of action, whether in contract or tort, including negligence. Any action against National Instruments must be brought within one year after the cause of action accrues. National Instruments shall not be liable for any delay in performance due to causes beyond its reasonable control. The warranty provided herein does not cover damages, defects, malfunctions, or service failures caused by owner's failure to follow the National Instruments installation, operation, or maintenance instructions; owner's modification of the product; owner's abuse, misuse, or negligent acts; and power failure or surges, fire, flood, accident, actions of third parties, or other events outside reasonable control.

Copyright

Trademarks

Patents

For patents covering National Instruments products, refer to the appropriate location: **Help»Patents** in your software, the `patents.txt` file on your media, or `ni.com/patents`.

WARNING REGARDING USE OF NATIONAL INSTRUMENTS PRODUCTS

(1) NATIONAL INSTRUMENTS PRODUCTS ARE NOT DESIGNED WITH COMPONENTS AND TESTING FOR A LEVEL OF RELIABILITY SUITABLE FOR USE IN OR IN CONNECTION WITH SURGICAL IMPLANTS OR AS CRITICAL COMPONENTS IN ANY LIFE SUPPORT SYSTEMS WHOSE FAILURE TO PERFORM CAN REASONABLY BE EXPECTED TO CAUSE SIGNIFICANT INJURY TO A HUMAN.

(2) IN ANY APPLICATION, INCLUDING THE ABOVE, RELIABILITY OF OPERATION OF THE SOFTWARE PRODUCTS CAN BE IMPAIRED BY ADVERSE FACTORS, INCLUDING BUT NOT LIMITED TO FLUCTUATIONS IN ELECTRICAL POWER SUPPLY, COMPUTER HARDWARE MALFUNCTIONS, COMPUTER OPERATING SYSTEM SOFTWARE FITNESS, FITNESS OF COMPILERS AND DEVELOPMENT SOFTWARE USED TO DEVELOP AN APPLICATION, INSTALLATION ERRORS, SOFTWARE AND HARDWARE COMPATIBILITY PROBLEMS, MALFUNCTIONS OR FAILURES OF ELECTRONIC MONITORING OR CONTROL DEVICES, TRANSIENT FAILURES OF ELECTRONIC SYSTEMS (HARDWARE AND/OR SOFTWARE), UNANTICIPATED USES OR MISUSES, OR ERRORS ON THE PART OF THE USER OR APPLICATIONS DESIGNER (ADVERSE FACTORS SUCH AS THESE ARE HEREAFTER COLLECTIVELY TERMED "SYSTEM FAILURES"). ANY APPLICATION WHERE A SYSTEM FAILURE WOULD CREATE A RISK OF HARM TO PROPERTY OR PERSONS (INCLUDING THE RISK OF BODILY INJURY AND DEATH) SHOULD NOT BE RELIANT SOLELY UPON ONE FORM OF ELECTRONIC SYSTEM DUE TO THE RISK OF SYSTEM FAILURE. TO AVOID DAMAGE, INJURY, OR DEATH, THE USER OR APPLICATION DESIGNER MUST TAKE REASONABLY PRUDENT STEPS TO PROTECT AGAINST SYSTEM FAILURES, INCLUDING BUT NOT LIMITED TO BACK-UP OR SHUT DOWN MECHANISMS. BECAUSE EACH END-USER SYSTEM IS CUSTOMIZED AND DIFFERS FROM NATIONAL INSTRUMENTS' TESTING PLATFORMS AND BECAUSE A USER OR APPLICATION DESIGNER MAY USE NATIONAL INSTRUMENTS PRODUCTS IN COMBINATION WITH OTHER PRODUCTS IN A MANNER NOT EVALUATED OR CONTEMPLATED BY NATIONAL INSTRUMENTS, THE USER OR APPLICATION DESIGNER IS ULTIMATELY RESPONSIBLE FOR VERIFYING AND VALIDATING THE SUITABILITY OF NATIONAL INSTRUMENTS PRODUCTS WHENEVER NATIONAL INSTRUMENTS PRODUCTS ARE INCORPORATED IN A SYSTEM OR APPLICATION, INCLUDING, WITHOUT LIMITATION, THE APPROPRIATE DESIGN, PROCESS AND SAFETY LEVEL OF SUCH SYSTEM OR APPLICATION.

Contents

About This Manual

This manual contains information about how to communicate and pass data between LabVIEW and either a local or a remote database management system (DBMS) using the LabVIEW Database Connectivity Toolkit.

This manual requires that you have a basic understanding of the LabVIEW environment, your computer, and your computer operating system.

Conventions

The following conventions appear in this manual:

» The » symbol leads you through nested menu items and dialog box options to a final action. The sequence **File»Page Setup»Options** directs you to pull down the **File** menu, select the **Page Setup** item, and select **Options** from the last dialog box.

 This icon denotes a note, which alerts you to important information.

bold Bold text denotes items that you must select or click in the software, such as menu items and dialog box options. Bold text also denotes parameter names.

italic Italic text denotes variables, emphasis, a cross-reference, or an introduction to a key concept. Italic text also denotes text that is a placeholder for a word or value that you must supply.

`monospace` Text in this font denotes text or characters that you should enter from the keyboard, sections of code, programming examples, and syntax examples. This font is also used for the proper names of disk drives, paths, directories, programs, subprograms, subroutines, device names, functions, operations, variables, filenames, and extensions.

Related Documentation

The following documents contain information that you may find helpful as you use the Database Connectivity Toolkit.

- *LabVIEW Help*, available by selecting **Help»Search the LabVIEW Help**
- Example VIs located in the `labview\examples\database` directory. You also can use the NI Example Finder, available by selecting **Help»Find Examples**, to find example VIs.

Note The following resources offer useful background information on the general concepts discussed in this documentation. These resources are provided for general informational purposes only and are not affiliated, sponsored, or endorsed by National Instruments. The content of these resources is not a representation of, may not correspond to, and does not imply current or future functionality in the Database Connectivity Toolkit or other National Instruments product.

- Forta, Ben. 2004. *Sams Teach Yourself SQL in 10 Minutes*. 3rd ed. Sams.
- Patrick, John J. 2002. *SQL Fundamentals*. 2nd ed. Prentice Hall.
- Sussman, David. 2004. *ADO Programmer's Reference*. Apress.
- Vaughn, William R. 2000. *ADO Examples and Best Practices*. Apress.

1

Introduction to the Database Connectivity Toolkit

The LabVIEW Database Connectivity Toolkit contains a set of VIs with which you can perform both common database tasks and advanced customized tasks.

The following list describes the main features of the Database Connectivity Toolkit:

- Works with any provider that adheres to the Microsoft ActiveX Data Object (ADO) standard.
- Works with any database driver that complies with ODBC or OLE DB.
- Maintains a high level of portability. In many cases, you can port an application to another database by changing the connection information you pass to the DB Tools Open Connection VI.
- Converts database column values from native data types to standard Database Connectivity Toolkit data types, further enhancing portability.
- Permits the use of SQL statements with all supported database systems, even non-SQL systems.
- Includes VIs to retrieve the name and data type of a column returned by a SELECT statement.
- Creates tables and selects, inserts, updates, and deletes records without using SQL statements.

Because of the wide range of databases with which the Database Connectivity Toolkit works, some portability issues remain. Consider the following issues when choosing your database system:

- Some database systems, particularly the flat-file databases such as dBase, do not support floating-point numbers. In cases where floating-point numbers are not supported, the toolkit converts floating-point numbers to the nearest equivalent, usually binary-coded decimal (BCD), before storing them in the database. Very large or very small floating-point numbers can pass the upper or lower limits of the precision available for a BCD value.

- The Microsoft ODBC driver for Oracle and the Microsoft OLE DB Provider for Oracle do not support BLOB (binary) data types. You cannot use Oracle with the Database Connectivity Toolkit for binary data with these drivers. Instead, use the OLE DB Provider and ODBC driver that Oracle provides. Refer to the Oracle Web site at www.oracle.com for more information about the OLE DB Provider and the ODBC driver that Oracle provides.
- Restrictions on column names vary among database systems. For maximum portability, limit column names to ten uppercase characters without spaces. You might be able to access longer column, or field, names or names that contain spaces by enclosing the name in double quotes.
- Some database systems do not support date, time, or date and time data types.

2

OLE DB Providers

The Microsoft Universal Data Access (UDA) platform allows applications to exchange relational or non-relational data across intranets or the Internet, essentially connecting any type of data with any type of application. OLE DB is the Microsoft system-level programming interface to diverse sources of data. The Microsoft ActiveX Data Object (ADO) standard is the application-level programming interface.

The Microsoft Data Access Components (MDAC) are the practical implementation of the Microsoft UDA strategy. MDAC includes ODBC, OLE DB, and ADO components. MDAC also installs several data providers you can use to open a connection to a specific data source, such as an MS Access database.

OLE DB Standard

OLE DB specifies a set of Microsoft Component Object Model (COM) interfaces that support various database management system services. These interfaces enable you to create software components that comprise the UDA platform. OLE DB is a C++ API that allows for lower-level database access from a C++ compiler. Three general types of COM components for OLE DB include:

- **OLE DB Data Providers**—Data-source-specific software layers that access and expose data.
- **OLE DB Consumers**—Data-centric applications, components, or tools that use data through OLE DB interfaces. Using networking terms, OLE DB consumers are the clients, and the OLE DB data provider is the server.
- **OLE DB Service Providers**—Optional components that implement standard services to extend the functionality of data providers. Examples of these services include cursor engines, query processors, and data conversion engines.

All data access in the LabVIEW Database Connectivity Toolkit occurs through an OLE DB provider. If you do not specify a provider, the toolkit uses the OLE DB Provider for ODBC provider, as described in the *OLE DB Provider for ODBC* section of this chapter. Microsoft provides some relational data providers as part of the MDAC installation.

Microsoft also provides a number of OLE DB data providers for non-relational data sources, including the following:

- OLE DB provider for AS/400
- OLE DB provider for Index Server
- OLE DB provider for Internet Publishing
- OLE DB provider for Active Directory
- OLE DB provider for Microsoft Exchange
- OLE DB provider for OLAP (Online Analytical Processing)

Some third-party vendors also supply OLE DB providers.

OLE DB Provider for ODBC

The OLE DB provider for ODBC acts as a conversion layer between OLE DB interfaces and ODBC. The hierarchy of data interface layers between LabVIEW and a database using the OLE DB provider for ODBC appears in Figure 2-1.

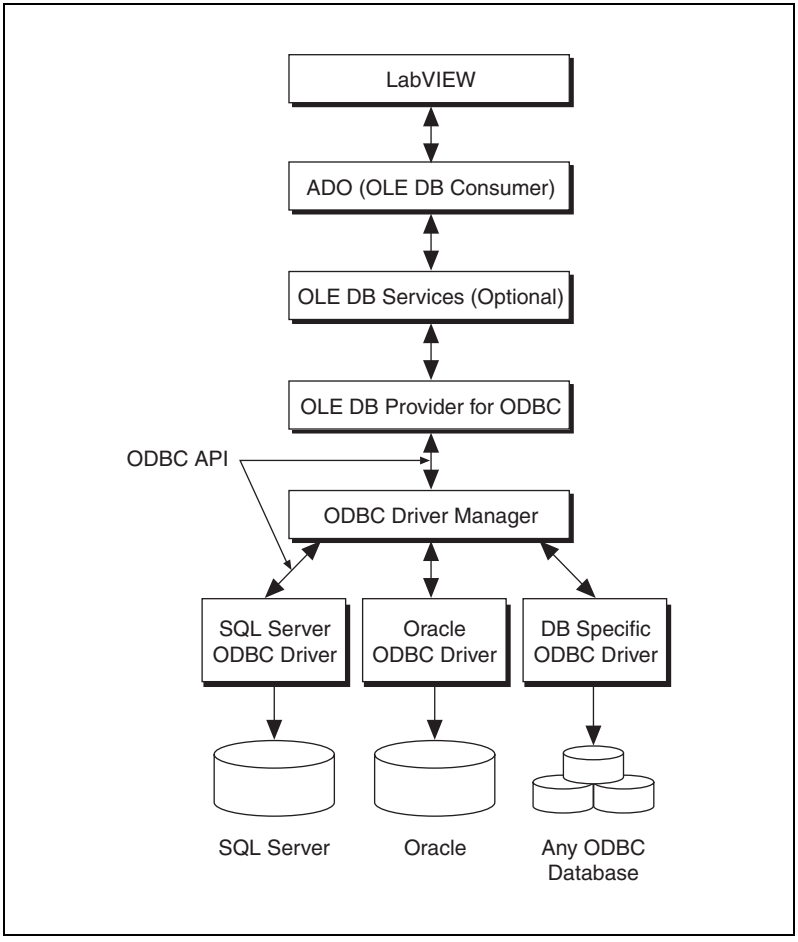

Figure 2-1. Communication Path between LabVIEW and a Database Using the OLE DB Provider for ODBC

MDAC 2.0 and later provide OLE DB providers for SQL Server, Jet, and Oracle database systems. Using native providers is much faster than using the OLE DB Provider for ODBC because native providers eliminate the need for both the OLE DB to ODBC conversion process and for the ODBC

driver and ODBC Driver Manager layers. For this reason, always use the native OLE DB data provider for the data source you are accessing if a native provider is available.

OLE DB Provider for SQL Server

The OLE DB provider for SQL Server, shown in Figure 2-2, exposes data stored in Microsoft SQL Server 6.5 or later databases.

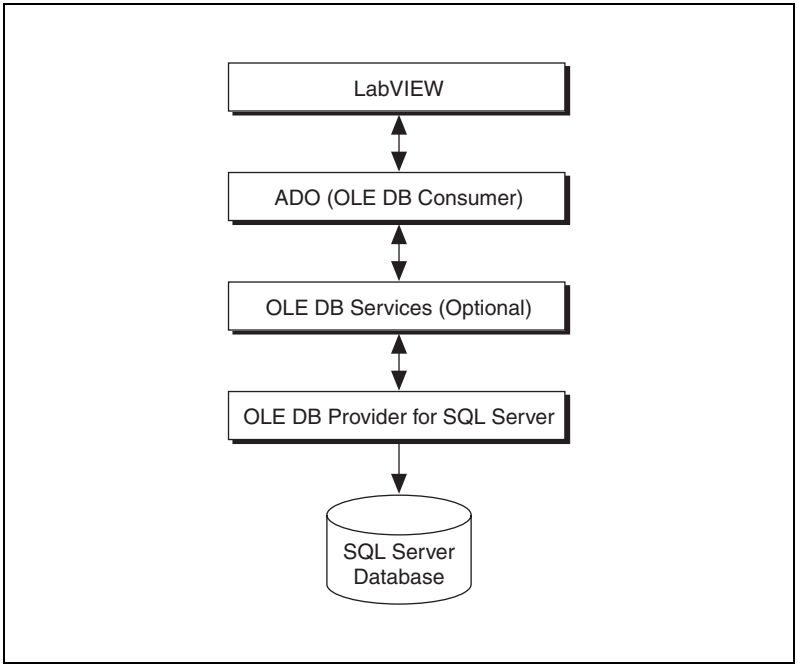

Figure 2-2. Communication Path between LabVIEW and an SQL Server Database Using the Native OLE DB Provider

OLE DB Provider for Jet

The OLE DB Provider for Jet uses the Microsoft Jet database engine to expose data stored in Microsoft Access databases (`.mdb`) and numerous Indexed Sequential Access Method (ISAM) databases, including Paradox, dBase, Btrieve, Excel, and FoxPro. The Jet database engine is included with Microsoft Access and is the underlying Database Management System (DBMS) of Microsoft Access. Visual Basic for Applications is the host language for the Jet DBMS.

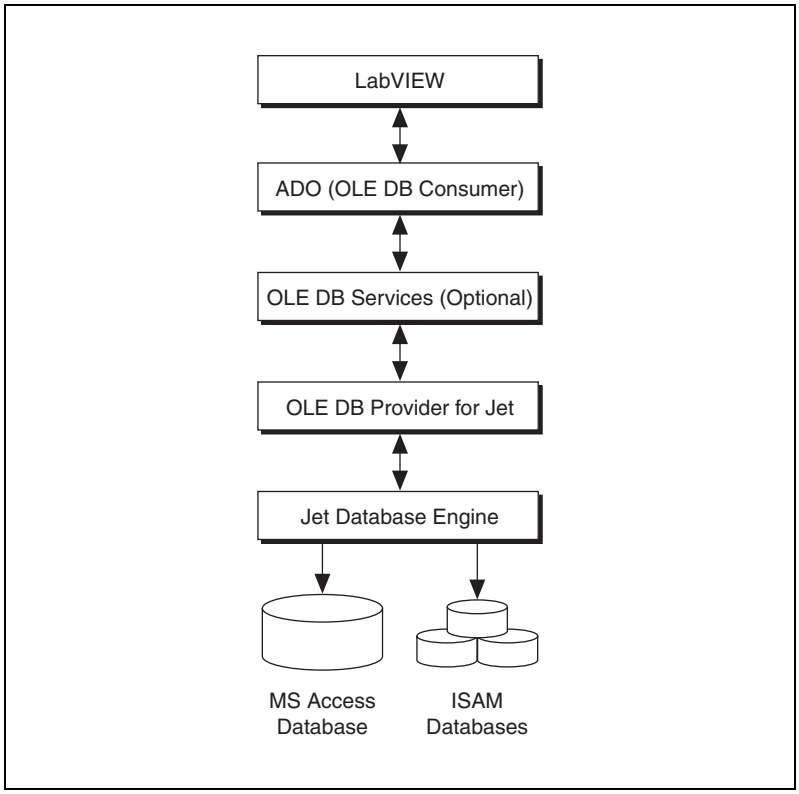

Figure 2-3. Communication Path between LabVIEW and an Access Database Using the Native OLE DB Provider

Data Access Objects (DAO) is the Jet interface for using the Jet database engine programmatically. DAO is a COM component that provides custom applications with the power and flexibility of the Jet database engine in a simple object model. DAO is also language-independent. Any programming language or toolkit that supports OLE Automation can use DAO and the Jet database engine.

Despite the availability of the OLE DB Provider for Jet and comparable benchmarks, some of the functionality of DAO, such as data definition and security, is not available in the OLE DB Provider for Jet.

Note Although DAO and ADO are both APIs for communicating with and manipulating data in databases, they are separate and different. DAO is specifically used with the Jet database engine, but ADO is part of Microsoft's UDA strategy for sharing data between applications and over the Internet.

OLE DB Provider for Oracle

The OLE DB provider for Oracle exposes OLE DB interfaces for retrieving and manipulating data stored in Oracle 7.3.3 or later databases. The OLE DB provider for Oracle is implemented as a layer on top of the Oracle native API, the Oracle Call Interface (OCI). Refer to the Oracle Web site at www.oracle.com for more information about the OLE DB provider for Oracle.

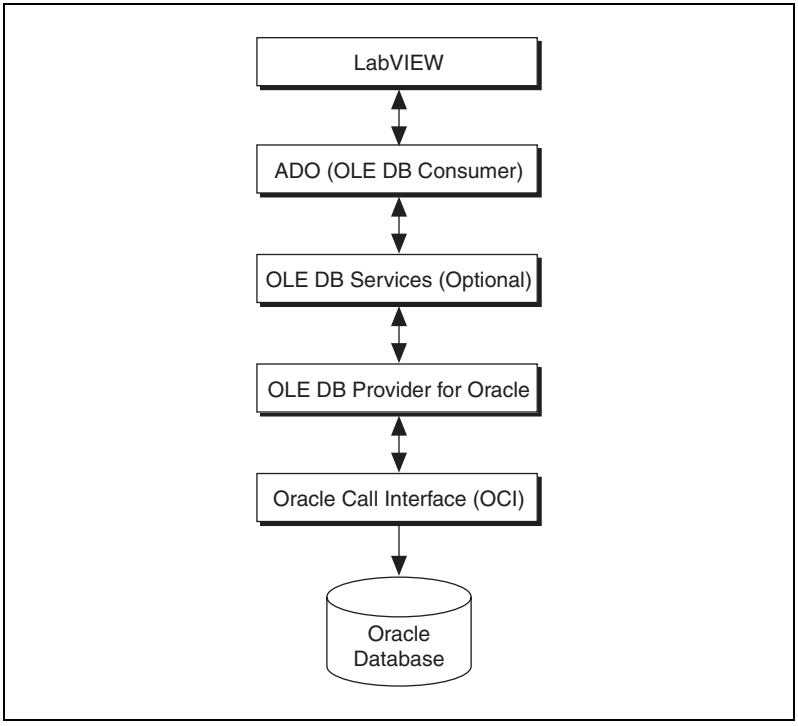

Figure 2-4. Communication Path between LabVIEW and an Oracle Database Using the Native OLE DB Provider

Custom OLE DB Providers

An advantage of UDA is the ability to develop custom OLE DB data providers because UDA enables standardized access to data sources beyond Microsoft products and the popular relational database systems.

If you need access to a data source that does not provide an OLE DB data provider and does not support ODBC, you can create custom OLE DB data providers that can expose any data source. For example, you can develop custom OLE DB data providers for data sources such as the following:

- Personal address book
- Windows registry
- Scheduled tasks
- Shared memory

3

Connecting to a Database

Before you can access data in a table or execute SQL statements, you must establish a connection to a database. The LabVIEW Database Connectivity Toolkit supports multiple simultaneous connections to a single database or to multiple databases. Use the DB Tools Open Connection VI to establish the connection to a database.

Connecting to a database is where most errors occur because each database management system (DBMS) uses different parameters for the connection and different levels of security. The different standards also use different methods of connecting to databases. For example, ODBC uses Data Source Names (DSN) for the connection, whereas the Microsoft ActiveX Data Object (ADO) standard uses Universal Data Links (UDL) for the connection. The DB Tools Open Connection VI supports all methods for connecting to a database.

DSNs and Data Source Types

A DSN is the name of the data source, or database, to which you are connecting. The DSN also contains information about the ODBC driver and other connection attributes including paths, security information, and read-only status of the database. Two main types of DSNs exist: machine DSNs and file DSNs. Machine DSNs are in the system registry and apply to all users of the computer system or to a single user. DSNs that apply to all users of a computer system are system DSNs. DSNs that apply to single users are user DSNs. A file DSN is a text file with a `.dsn` extension and is accessible to anyone with proper permissions. File DSNs are not restricted to a single user or computer system. Use the ODBC Data Source Administrator to create and configure DSNs.

ODBC Data Source Administrator

Use the ODBC Data Source Administrator to register and configure drivers to make them available as data sources for applications. In the Windows Control Panel, select **Administrative Tools»Data Sources (ODBC)** to display the ODBC Data Source Administrator. The system saves the configuration for each data source in the registry or in a file.

ODBC drivers for databases such as SQL Server and Oracle contain settings and additional dialog boxes for configuring items such as server information, user identification, and passwords. Figure 3-1 shows the ODBC Microsoft Access Setup dialog box for the system DSN named LabVIEW that the Database Connectivity Toolkit examples use.

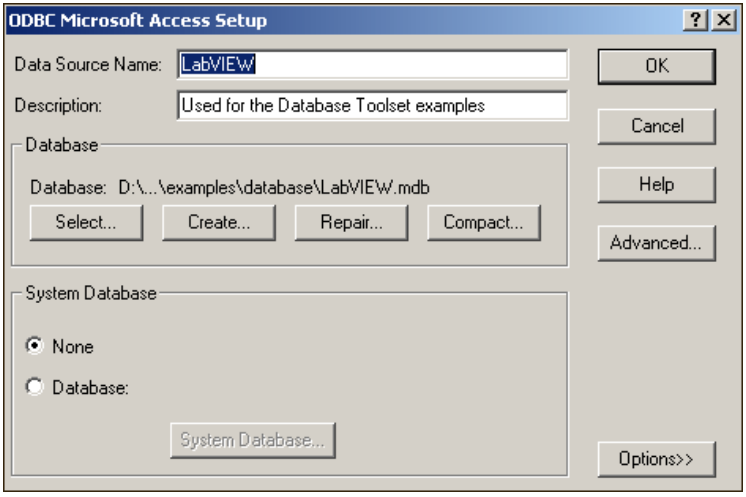

Figure 3-1. ODBC Microsoft Access Setup Dialog Box

The Database Connectivity Toolkit complies with the ODBC standard, so you can use the toolkit with any ODBC-compliant database drivers. The Database Connectivity Toolkit does not provide custom ODBC drivers. However, Microsoft Data Access Components (MDAC) includes several ODBC drivers. Database system vendors and third-party developers also offer large selections of ODBC drivers. Refer to the vendor documentation for information about registering the specific database drivers in the ODBC Data Sources Administrator.

Connecting to Databases Using DSNs

You can use the DB Tools Open Connection VI to connect to various databases that you specify with DSNs.

You can use a string to specify a system DSN or user DSN. The VI in Figure 3-2 specifies a DSN called `MS Access` to open a connection to that specific database.

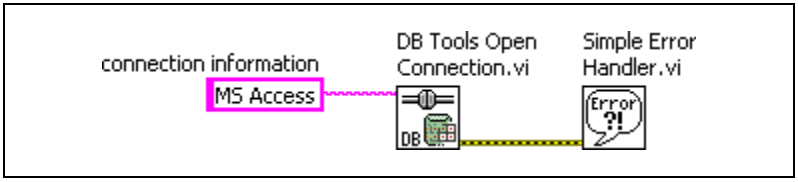

Figure 3-2. Connecting to an Access Database Using a System DSN

You can use a path to specify a file DSN. The VI in Figure 3-3 specifies a path to a file DSN called `access.dsn` to open a connection to the database.

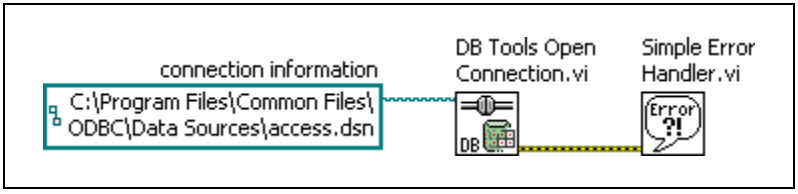

Figure 3-3. Connecting to an Access Database Using a File DSN

Notice that the **connection information** input of the DB Tools Open Connection VI is polymorphic. This VI accepts either a string or path for the DSN.

The VI in Figure 3-4 connects to an Oracle database using a system DSN. Notice that the **userID** and **password** parameters are wired. Some DBMS require that these parameters be set in order to connect to a database. You should be familiar with your DBMS and how to specify the connection parameters.

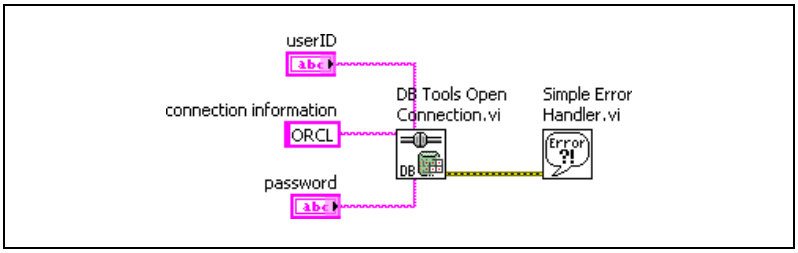

Figure 3-4. Connecting to an Oracle Database Using a System DSN

As shown in the previous examples, connecting to a database using the DB Tools Open Connection VI requires only a string or path value specifying the DSN along with optional user ID and password strings depending upon the DBMS. Therefore, the majority of problems in defining a connection occur when creating the DSN. Some ODBC drivers have an option to test the connection. Test the connection between the DSN and the database before you try to do anything with the Database Connectivity Toolkit.

Whereas you must create a DSN to connect to a database using ODBC, you use UDLs to connect to databases that use ADO and OLE DB.

UDLs

A UDL is similar to a DSN in that it describes more than just the data source. A UDL specifies what OLE DB provider is used, server information, the user ID and password, the default database, and other related information.

You can create a UDL in one of the following three ways:

- Use the **prompt?** input of the DB Tools Open Connection VI, as shown in Figure 3-5.

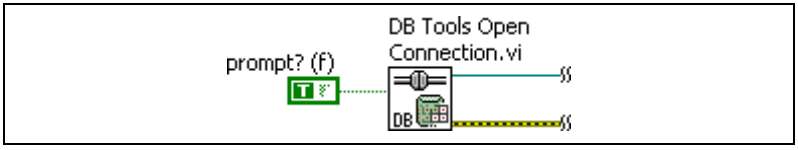

Figure 3-5. Using a Prompt to Create a UDL

The **prompt?** input displays the **Data Link Properties** dialog box when the DB Tools Open Connection VI runs. You can select the appropriate options in this dialog box to make the database connection.

- Select **Tools»Create Data Link** in LabVIEW to display the **Data Link Properties** dialog box.

Note The Database Connectivity Toolkit installer creates a directory called `data links` inside the `labview/Database` directory. Save all UDL files and file DSNs to this directory so you can find them easily.

- In Windows Explorer, right-click an empty location in a folder and select **New»Text Document** from the shortcut menu. Change the file extension of this document from `.txt` to `.udl`. You then can double-click the UDL file to display the **Data Link Properties** dialog box.

Configuring a UDL

Any method of creating a UDL involves the **Data Link Properties** dialog box. Select a data provider from the **Provider** page of this dialog box. Refer to Chapter 2, *OLE DB Providers*, to determine which provider to use with a database.

After you select a data provider from the list on the **Provider** page, you can configure the database connection on the **Connection** page. The options on the **Connection** page are different depending upon which provider you choose. For example, the **Connection** page for an ODBC provider contains a selection for a DSN or connection string along with user name and password information. Click the **Test Connection** button to test the database connection after you configure the various properties. Make sure the connection test passes before you click the **OK** button to exit.

Connecting to Databases Using UDLs

Use a path control or constant to specify the path to a UDL file unless you set the **prompt?** input of the DB Tools Open Connection VI to TRUE. The VI in Figure 3-6 uses a path constant to specify the UDL for a Microsoft Access database.

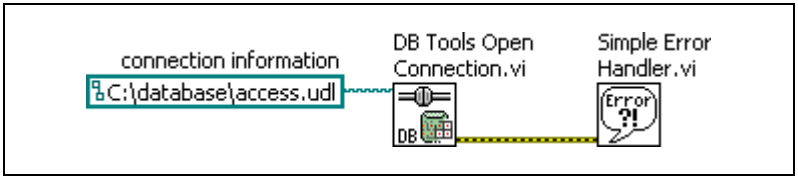

Figure 3-6. Connecting to a Microsoft Access Database Using a UDL

Although you might have created the DSN or UDL correctly, you still might not be able to connect to a specific database because of situations beyond your control. The following situations can prevent you from connecting to a database:

- The requested server is down.
- The network is down.
- All server connections are full, and no other users can connect.
- The maximum number of user licenses have been reached.
- You do not have permission to access the specified database.
- The specified DSN does not exist. Either you are on a different machine, or the specified DSN was deleted.
- The selected data provider is the wrong one for the database.

If the DB Tools Open Connection VI returns errors, you can open the UDL file manually and click the **Test Connection** button on the **Data Link Properties** dialog box to verify that you have the correct settings and that you have access to the database. If the test connection fails, you cannot connect to that database with the Database Connectivity Toolkit. Contact the database administrator for help.

4

Supported Data Types

LabVIEW, the Microsoft ActiveX Data Object (ADO) standard, and each database management system (DBMS) support a different set of data types.

Data Type Mapping

The LabVIEW Database Connectivity Toolkit maps the various LabVIEW data types to data types supported by some of the common DBMS. Table 4-1 shows which SQL data types the Database Connectivity Toolkit supports.

Table 4-1. Database Connectivity Toolkit Data Types

Database Connectivity Toolkit Data Type	SQL Data Type	Description of SQL Data Type
string	CHAR (x), VARCHAR (x)	**CHAR**—Fixed character data such as CHAR (16). Extra is filled with spaces. **VARCHAR**—Varying character data. Does not pad with spaces.
long	INTEGER	Precision depends on the specific SQL implementation; database developer cannot specify the precision.
single	REAL	Single-precision floating-point number determined by the OS implementation of a SGL.
double	DOUBLE PRECISION	Double-precision floating-point number determined by the OS implementation of a DBL.

Table 4-1. Database Connectivity Toolkit Data Types (Continued)

Database Connectivity Toolkit Data Type	SQL Data Type	Description of SQL Data Type
date/time	DATE, TIME (p)	**DATE**—Length of 10 positions in the form: YYYY-MM-DD. **VARCHAR**—Has the form: HH:MM:SS.SSS... specified by p.
binary	BINARY (n), VARBINARY (n)	**BINARY**—Fixed length binary string with maximum length n. **VARBINARY**—Variable length binary string with maximum length n.

All LabVIEW data types are supported but not necessarily in their native form. For example, bytes (U8 and I8) and words (U16 and I16) can be treated as longs (I32). The binary data type encompasses any piece of LabVIEW data, such as waveform, cluster, or array data, that cannot be represented natively in the database. Table 4-2 lists LabVIEW data types and the data types in the Database Connectivity Toolkit to which they correspond.

Table 4-2. LabVIEW and the Database Connectivity Toolkit Data Types

LabVIEW Data Type	Database Connectivity Toolkit Data Type
8-bit integers	long
16-bit integers	long
32-bit integers <= 2147483647	long
32-bit integers > 2147483647	string
8-bit enum	long
16-bit enum	long
32-bit enums <= 2147483647	long
32-bit enums > 2147483647	string
64-bit integers	string
64-bit enums	string
Single numeric	single

Table 4-2. LabVIEW and the Database Connectivity Toolkit Data Types (Continued)

LabVIEW Data Type	Database Connectivity Toolkit Data Type
Double numeric	double
Boolean	string
String	string
Date/Time string	date/time
Time stamp	date/time
Path	string
I/O channel	string
Refnum	binary
Complex numeric	binary
Extended numeric	binary
Picture control	binary
Array	binary
Cluster	binary
Variant	binary
Waveform	binary
Digital waveform	binary
Digital data	binary
WDT	binary
Fixed-point numeric	binary

Although the Database Connectivity Toolkit supports refnums, refnums are ephemeral constructs whose values are meaningless after usage. If you want to save a refnum to a database table, you must first type cast the refnum to an integer and then write the integer to the table.

Note The Database Connectivity Toolkit inserts DAQ, IVI, and VISA Channel refnums into the database as strings. You can use the Database Variant To Data function to convert these refnums back into DAQ, IVI, or VISA Channel refnums.

Working with Date/Time Data Types

Date/time is an important data type for databases. You can use the time stamp data type to represent date and time in LabVIEW. You also can use the DB Tools Format Datetime Str VI to insert date/time strings into a database. The DB Tools Format Datetime Str VI formats a string into the correct format for SQL. This VI places a header at the beginning of the string that is later decoded in other VIs to determine that the string is a date/time string. Refer to the *Formatting Date and Time* section of Chapter 6, *Using the Database Connectivity Toolkit Utility VIs*, for more information about using the DB Tools Format Datetime Str VI to format date and time in a database.

The main problem with the date/time data type is that no uniformity exists and each database supports a different format. In other words, when you select date/time values from a database, they might be returned in a different form depending on the DBMS.

Handling NULL Values

Databases have NULL fields that are empty fields containing no data. LabVIEW treats NULLs as default data, such as an empty string, a zero-value numeric, or a FALSE Boolean. Therefore, for example, you cannot easily differentiate between a `0.00` value in a numeric from one that is NULL. Figures 4-1 and 4-2 show how different formats represent NULL values.

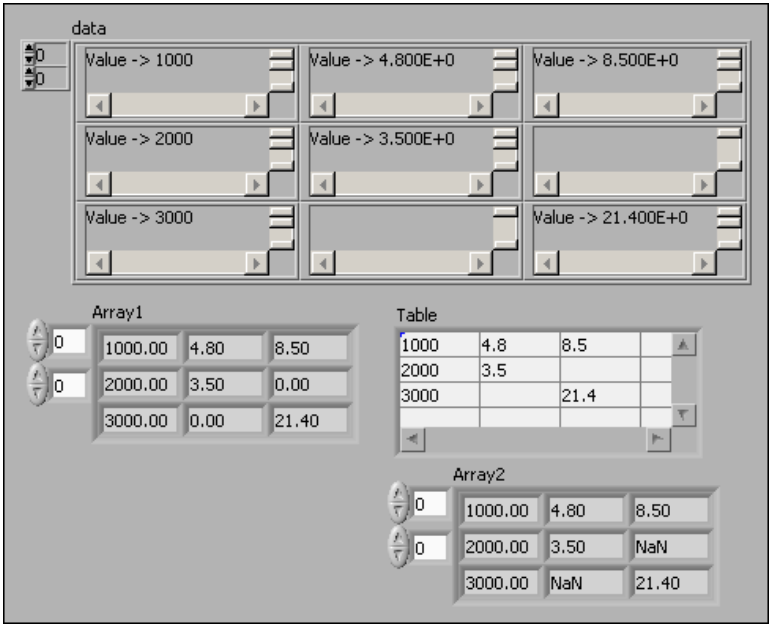

Figure 4-1. Handling of NULLs in Different Formats

When you convert the NULL values in the variant array into numeric values, the NULLs become `0.00` values. However, when you convert the variant array into strings, the NULLs become empty strings. You can convert the strings to numbers and, when the string is empty, insert a NaN value, as shown in Figure 4-2.

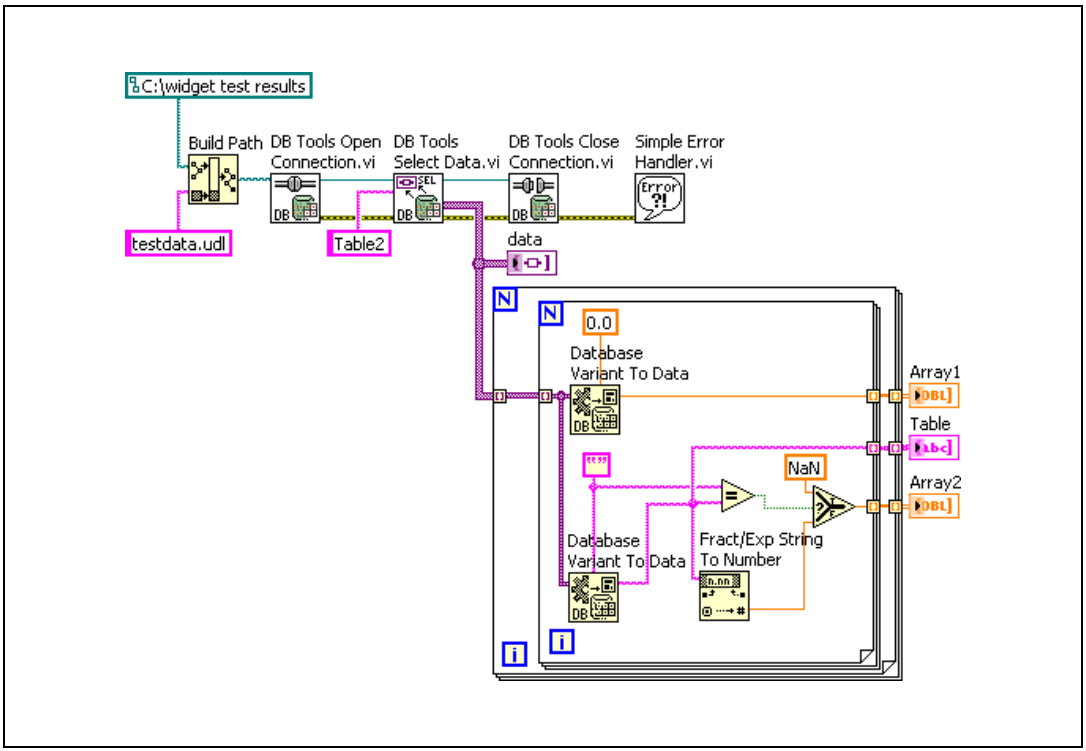

Figure 4-2. Block Diagram Showing How NULLs Are Handled

Currency and Boolean Data Types

Although currency and Boolean (Yes/No in Microsoft Access) are common data types, the Database Connectivity Toolkit does not directly support these data types because these data types are not available in other DBMS such as Oracle. However, you can write data to and read data from these field types with the Database VIs using strings. Figure 4-3 shows how you can convert currency and Boolean data to strings and write the information to the appropriate fields in a Microsoft Access table.

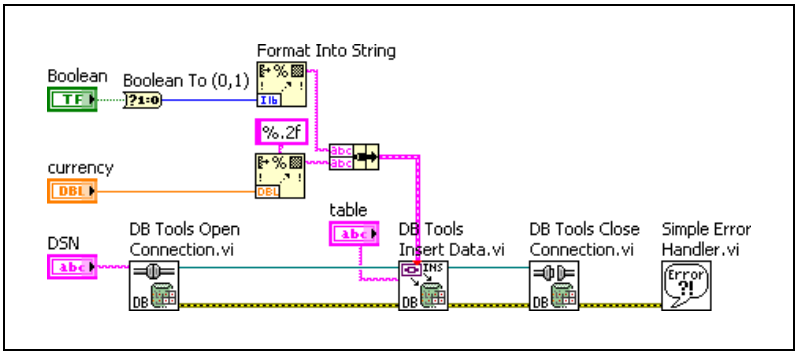

Figure 4-3. Writing Currency and Boolean Data

When you read Boolean data from a table, the data is returned as `TRUE` or `FALSE` strings. Currency data is read from a table as a number without the `$` currency symbol. Avoid using data types that are not supported by the Database Connectivity Toolkit.

5

Performing Standard Database Operations

You can use the Database VIs and function to write data to or read data from databases and to create and delete tables.

Writing Data to a Database

Writing data to a database with the LabVIEW Database Connectivity Toolkit is similar to writing data to a file. You open a connection, insert the data, and close the connection when you are finished. Figures 5-1 and 5-2 show the front panel and block diagram of a VI that writes test information to a database table. The connection information is a path to the UDL called `test.udl`, and the table name is `testdata`.

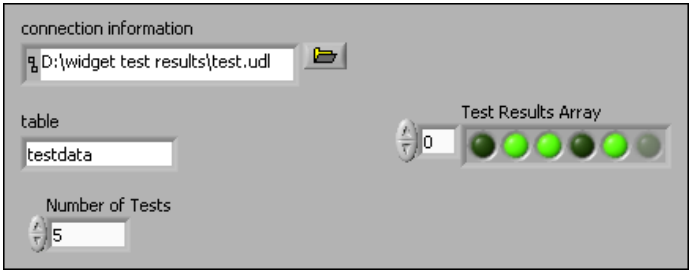

Figure 5-1. Front Panel Showing How to Write Data to a Database Table

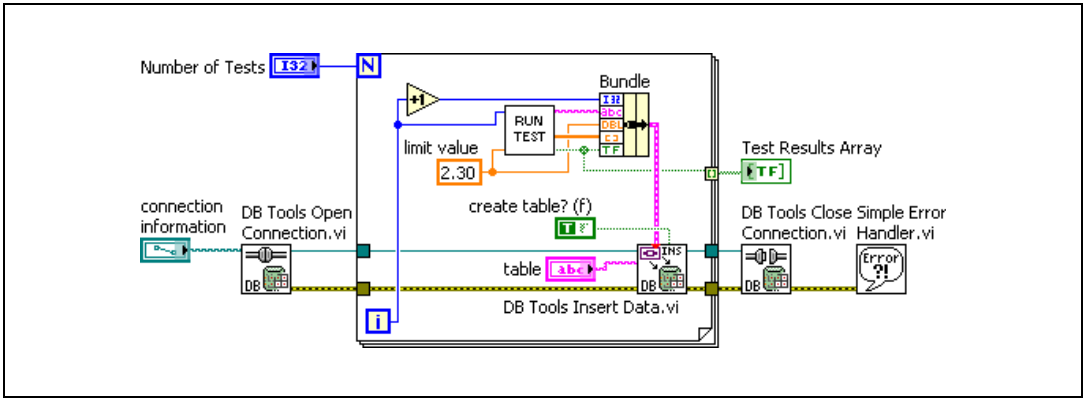

Figure 5-2. Block Diagram Showing How to Write Data to a Database Table

Figure 5-2 uses three Database VIs: the DB Tools Open Connection VI, the DB Tools Insert Data VI, and the DB Tools Close Connection VI. The **create table?** input of the DB Tools Insert Data VI is set to TRUE to create the specified table if it does not already exist. If this table does exist, then the data is appended to the existing table. The DB Tools Insert Data VI accepts any type for the data input. If the input type is a cluster, each cluster element is placed into a different field. The LabVIEW data types are converted to the appropriate database data types. Refer to Chapter 4, *Supported Data Types*, for more information about supported data types.

Figure 5-3 shows the `testdata` table as it appears in Microsoft Access. Note that the front panel and block diagram previously shown do not specify the type of database to use. That configuration occurs when the `test.udl` is created.

	col0	col1	col2	col3	col4
	1	1/25/2001 10:56:42 AM	2.3	Long binary data	false
	2	1/25/2001 10:56:42 AM	2.3	Long binary data	true
	3	1/25/2001 10:56:43 AM	2.3	Long binary data	true
	4	1/25/2001 10:56:43 AM	2.3	Long binary data	false
	5	1/25/2001 10:56:43 AM	2.3	Long binary data	true
▶					

Figure 5-3. Database Table Displayed in Microsoft Access

Notice that the column names are not specified in the VI, so the table uses default column names. You can specify column names using the **columns** input of the DB Tools Insert Data VI.

Reading Data from a Database

Reading data from a database table is similar to writing data to the database. You open a connection to the database, select the data from a table, and then close the connection. Figures 5-4 and 5-5 show how you can read the data back from the `testdata` table used in the previous example.

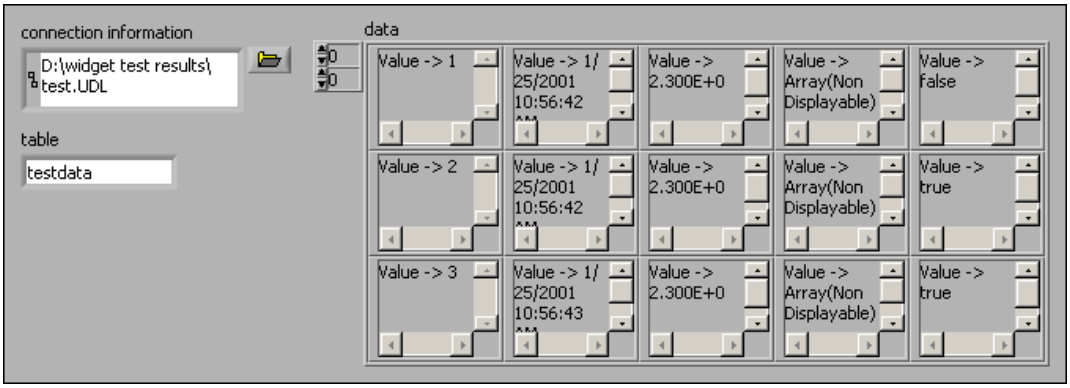

Figure 5-4. Front Panel Showing How to Read Data from a Database Table

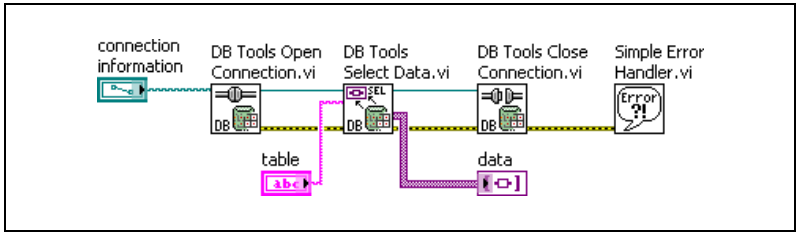

Figure 5-5. Block Diagram Showing How to Read Data from a Database Table

Notice in Figures 5-4 and 5-5 that the database data is returned as a two-dimensional array of variants. As the name implies, the Microsoft ActiveX Data Object (ADO) standard is based on ActiveX, which defines variants as its data types. Variants work well in languages such as Visual Basic that are not strongly typed. Because LabVIEW is strongly typed, you must use the Database Variant To Data function to convert the variant data to a LabVIEW data type before you can display the data in standard indicators such as graphs, charts, and LEDs.

Figures 5-6 and 5-7 show the front panel and block diagram for a VI that reads all data from a database table and then converts the data to appropriate data types in LabVIEW. In Figure 5-6, notice that the fourth column of data that does not display properly in either Microsoft Access or the variant is now displayed in a waveform graph.

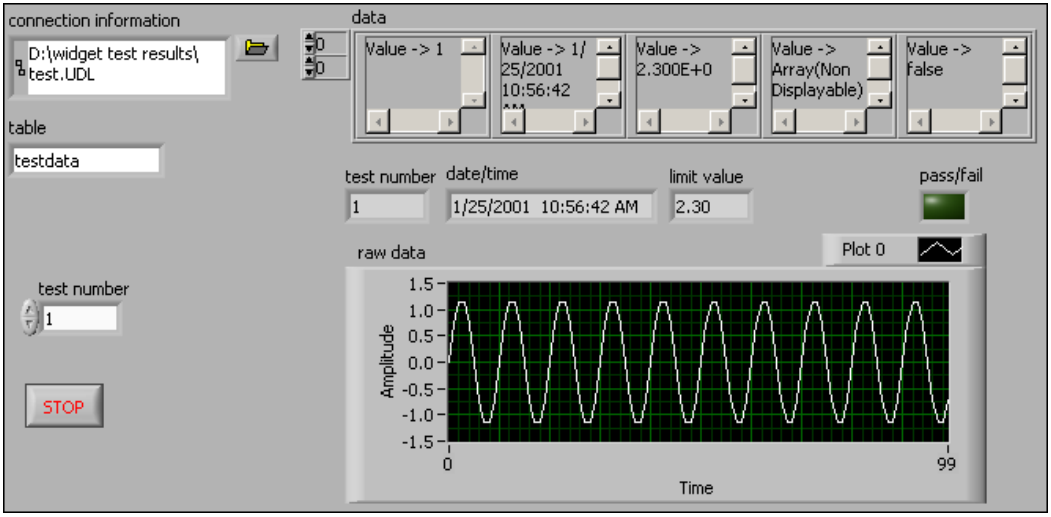

Figure 5-6. Front Panel Showing How to Read and Convert Data from a Database Table

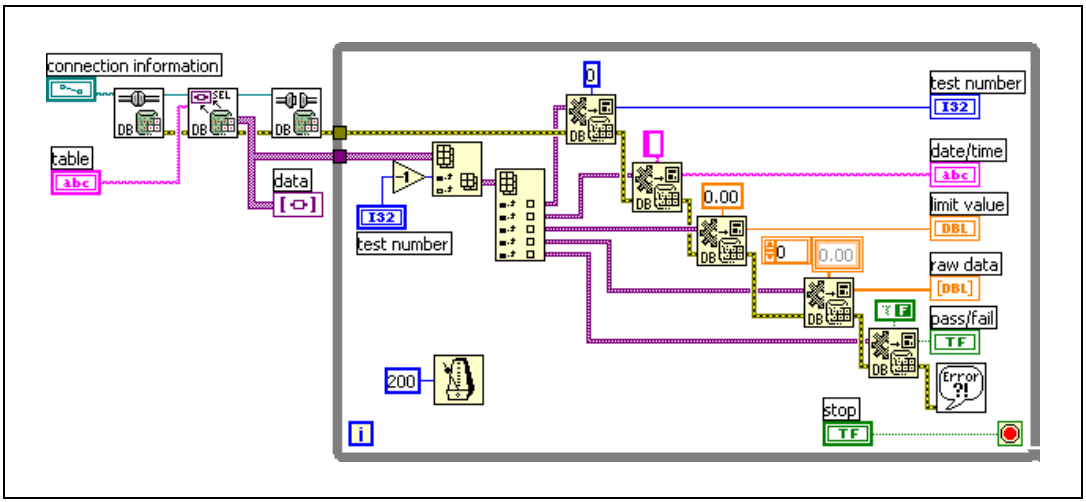

Figure 5-7. Block Diagram Showing How to Read and Convert Data from a Database Table

You can use the **table** input of the DB Tools Select Data VI to read data from more than one table in a database. Figure 5-8 shows how you can use a comma-delimited string to specify multiple table names. The **data** array includes all rows and columns from both tables in the order they appear in the **table** string.

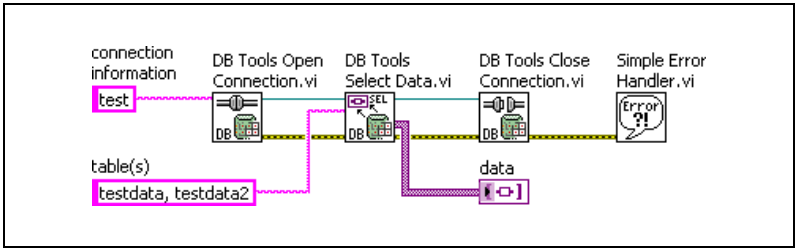

Figure 5-8. Specifying Multiple Database Tables for Reading Data

Limiting Data to Read

If you are reading data from a large table or set of tables, it might take several seconds to return all the data. There is no limit to the size of the database table you can read other than your computer resources, memory, and speed. Read only the necessary fields or perform an SQL query to limit the amount of information to read into LabVIEW at one time. Figure 5-9 shows how you can use the **columns** string array to specify which columns to read and limit the returned data.

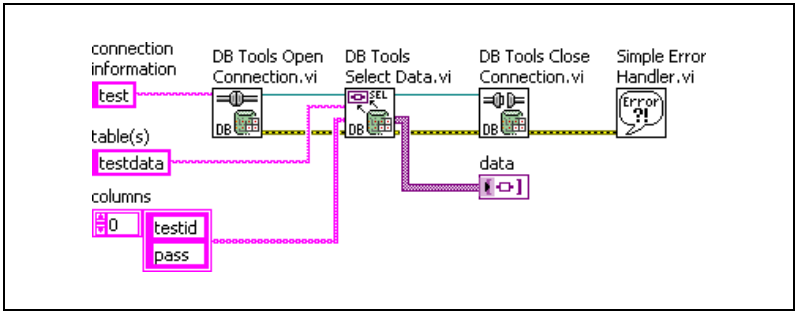

Figure 5-9. Specifying Column Names for Reading Data

In Figure 5-9, only the `testid` and `pass` fields are returned from a table named `testdata`. You can limit the returned data further by specifying conditions using the **optional clause** string. Figure 5-10 shows how you can limit the results from the previous example by returning the `testid` and `testdate` fields for the records where the `pass` field equals `TRUE`.

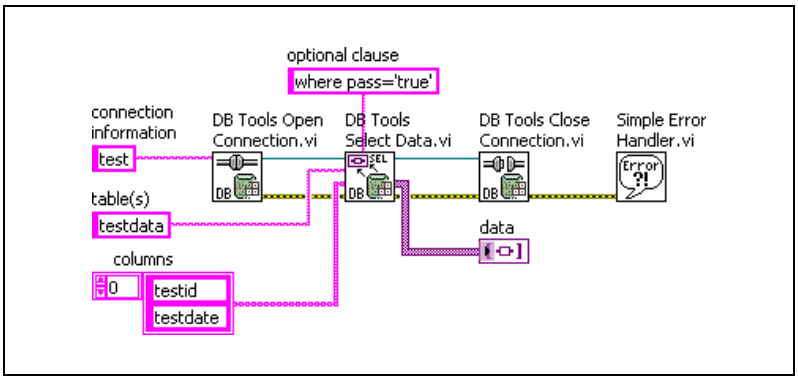

Figure 5-10. Specifying Conditions for Reading Data

The statement `where pass='true'` is part of an SQL query. Refer to one of the SQL references listed in the *Related Documentation* section of this manual for information about creating an SQL query.

Note If you receive an error while using the DB Tools Select Data VI, either a specified field in the **columns** string array does not exist in the table, or that column name contains characters such as a space, `-`, `\`, `/`, or `?`. Do not use these characters when naming tables in a database. However, if an existing database contains such characters, enclosing the column name in double quotes often solves the problem.

Creating and Deleting Tables

Use the DB Tools Create Table VI and the DB Tools Drop Table VI to create or delete tables in a database. Use the DB Tools Open Connection VI to connect to a database and then use the DB Tools Create Table VI or the DB Tools Drop Table VI to perform the desired operation. Use the DB Tools Close Connection VI to end communication with the database. Figure 5-11 shows how you can create a new table with these VIs.

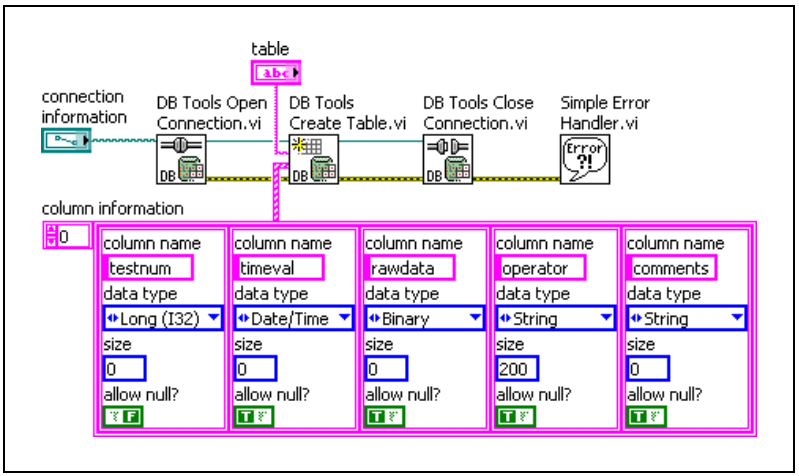

Figure 5-11. Creating a Database Table

Remember that when you set the **create table?** input of the DB Tools Insert Data VI to TRUE, this VI creates the specified table if it does not already exist. The DB Tools Insert Data VI actually uses the DB Tools Create Table VI as a subVI to create a table. You also can use the DB Tools Create Table VI at the highest level if you want more control over the database fields such as specifying column names, data types, and whether to allow NULL values.

The **size** parameter affects only the string data type. If you use the default size of 0 for a string, the maximum size for a string is defined by the specified provider.

Figure 5-12 shows how to use the DB Tools Drop Table VI to delete a table from a database.

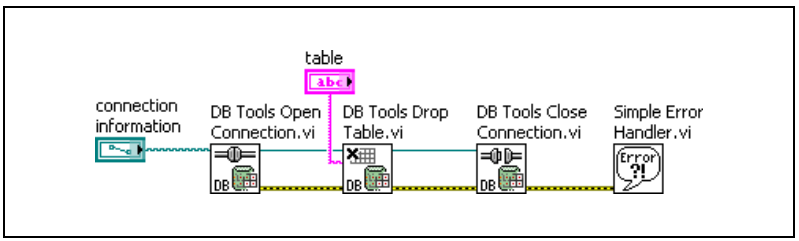

Figure 5-12. Deleting a Database Table

Using the Database Connectivity Toolkit Examples

The Database Connectivity Toolkit provides several examples that demonstrate how to perform common database operations with the Database VIs. These examples use a UDL called `LabVIEW.udl` to link to a Microsoft Access database named `LabVIEW.mdb`.

Using the Examples with Other Databases

You can use the Database Connectivity Toolkit example VIs by modifying the `LabVIEW.udl` file. Double-click this UDL file in the `labview/examples/database` directory to display the **Data Link Properties** dialog box. You then can select a different provider and set the connection properties for your DBMS. The default values for some of the example VIs assume the presence of a particular table in the database from which to read data or to which to add data. You must modify the example to fit the table names, column names, and data types required.

Using the Examples without a Database

You do not need to have MS Access or any other database installed to use the Database Connectivity Toolkit examples. If you run the examples with their default values, the data is read from or written to the `LabVIEW.mdb` file even if you do not have Microsoft Access installed. If you want to create a new database file to write data to and read data from, you can copy and rename the `LabVIEW.mdb` file and use the DB Tools Drop Table VI to remove the existing tables. You then can use the DB Tools Create Table VI to create new tables specific to the application.

6

Using the Database Connectivity Toolkit Utility VIs

Use the Utility VIs for a variety of operations, including getting table and column information, getting and setting database properties, formatting data and time data, performing database transactions, and writing and reading data files. Refer to the *LabVIEW Help* for more information about using these VIs.

Getting Table and Column Information

Sometimes you must work with databases created by other users or groups, and you are not familiar with the structure of the database. You can use the DB Tools List Tables VI to determine what tables exist in a particular database. Use the DB Tools List Columns VI to return an array of column or field names in a table and to return information about the data type and size of each field. Figures 6-1 and 6-2 show how to use these Utility VIs to get information about a database.

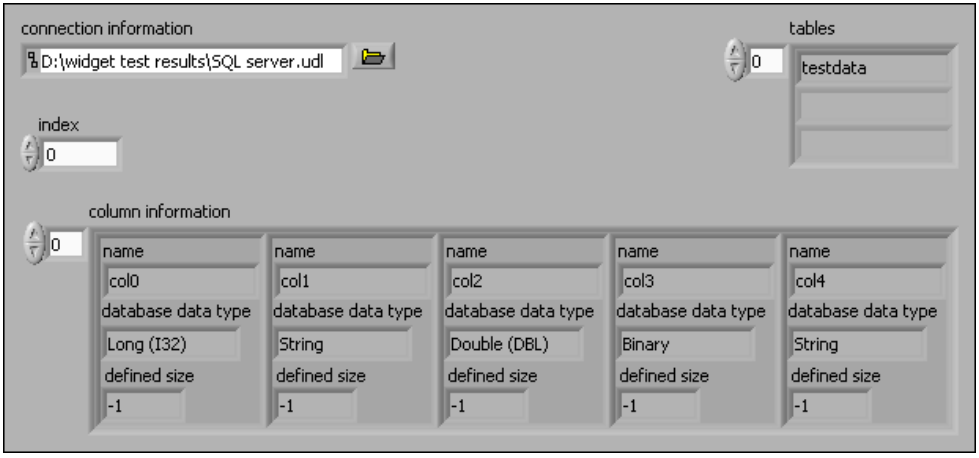

Figure 6-1. Front Panel Showing How to Get Database Information

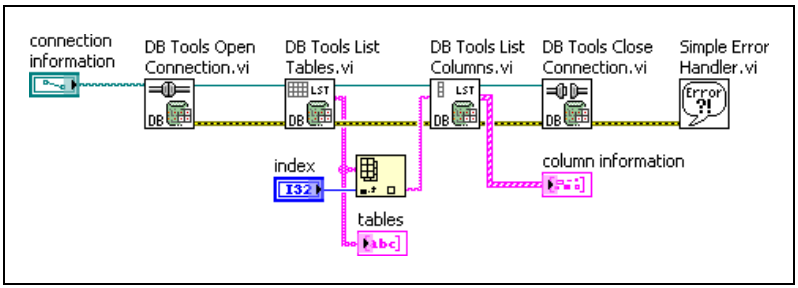

Figure 6-2. Block Diagram Showing How to Get Database Information

If you want to get information about all fields in all tables, you can place the DB Tools List Columns VI into a For Loop instead of using the Index Array function. Refer to Chapter 3, *Connecting to a Database*, and Chapter 4, *Supported Data Types*, for more information about how the LabVIEW Database Connectivity Toolkit maps the data types from DBMS to LabVIEW data types.

Getting and Setting Database Properties

You can read or write various database properties using the DB Tools Get Properties and the DB Tools Set Properties VIs. Both of these VIs are polymorphic and can accept different types of reference inputs. The exact properties you can set or read are based on the type of reference you wire to the reference input. You can use the Connection, Command, and Recordset Microsoft ActiveX Data Object (ADO) reference object classes to read and write properties in a database. The Database Connectivity Toolkit also supports a fourth object class called Command-Recordset to handle the close relationship between the ADO Recordset and Command objects.

ADO Reference Classes

Table 6-1 describes the purpose of each object class as it relates to the Database Connectivity Toolkit.

Table 6-1. Database Connectivity Toolkit Object Classes

Object Class	Description
Connection	Use this class to define the database connection parameters, such as the OLE DB provider, connection string, and default database. After you create a Connection reference, delete it with the DB Tools Free Object VI.
Command	Use this class to execute commands and capture parameters returned from stored procedures. Create a Command reference by first creating a Connection reference and then calling the DB Tools Create Parameterized Query VI. You can get or set properties related to the command or the parameters associated with the command. After you create a Command reference, delete it with the DB Tools Free Object VI.
Recordset	Use this class to manipulate data. Create a Recordset reference by first creating a Connection reference and then calling the DB Tools Execute Query VI. You can get or set properties related to column information, the number of records available, the beginning or end of file markers, and the type of cursor used. After you create a Recordset reference, delete it with the DB Tools Free Object VI.
Command-Recordset	Use this class for situations where commands and recordsets are used together, such as SQL queries. Create a Command-Recordset reference by first creating Connection and Command references and then calling the DB Tools Execute Query VI. You can get or set all the properties available to the Command and Recordset references. After you create a Command-Recordset reference, delete it with the DB Tools Free Object VI.

Be careful what references you wire from one VI to the next. You might obtain unexpected results when you wire a different type of reference than expected to the input of a VI. Refer to the *LabVIEW Help* for more information about the reference types used for VI inputs and outputs.

Database Properties

The list of available properties for a database changes not only with the reference type but also with the data provider. Each OLE DB data provider supports different properties for each of the ADO class types. Also, not all OLE DB developers are required to implement properties in the same way. If you use a particular property with one database, that same property might not work in the same way for another database. Some properties are read-only and you cannot set them. Refer to the *LabVIEW Help* for more information about specific property values.

Formatting Date and Time

You can write time stamp data directly to a database using the Time Stamp control. You also can use the DB Tools Format Datetime Str VI to format a LabVIEW date/time string so that other Database VIs recognize the string as a separate data type. Figure 6-3 shows how to use the DB Tools Format Datetime Str VI to send a time stamp to the second field of the `testdata` table.

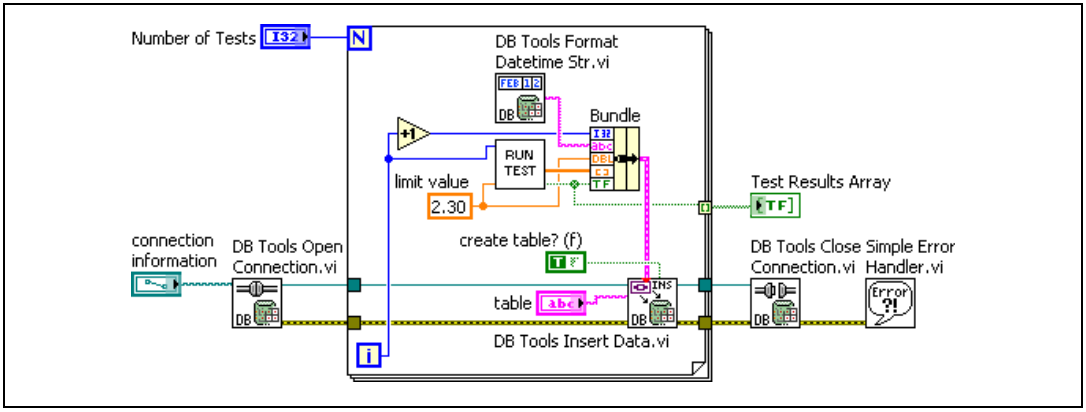

Figure 6-3. Writing Date and Time to a Database

The first version of this VI, shown in Figure 5-2, *Block Diagram Showing How to Write Data to a Database Table*, writes the date and time to the database as a text string because LabVIEW recognizes date and time in this format. Figure 5-3 shows the corresponding Datasheet View in Microsoft Access. However, the Design View for the table in Figure 5-3 shows Field `col1` as a text data type. Figure 6-4 shows the data type for the table created by the block diagram in Figure 6-3 as Date/Time.

testdata2 : Table

	Field Name	Data Type	Description
	col0	Number	
▶	col1	Date/Time	
	col2	Number	
	col3	OLE Object	
	col4	Text	

Figure 6-4. Database Table with Date/Time Data

You read this data back into LabVIEW in the same way as described in the *Reading Data from a Database* section of Chapter 5, *Performing Standard Database Operations*. You treat the date/time data as a string.

Refer to the *Working with Date/Time Data Types* section of Chapter 4, *Supported Data Types*, for more information about the date/time data type.

Performing Database Transactions

Protecting the integrity of a database is often difficult. Multiple users can have access to a single database at the same time, and each user can change the data. You can use the DB Tools Database Transaction VI, as shown in Figure 6-5, to specify when to actually perform, or commit, a database operation and when to return to the previous state of, or roll back, the database operation.

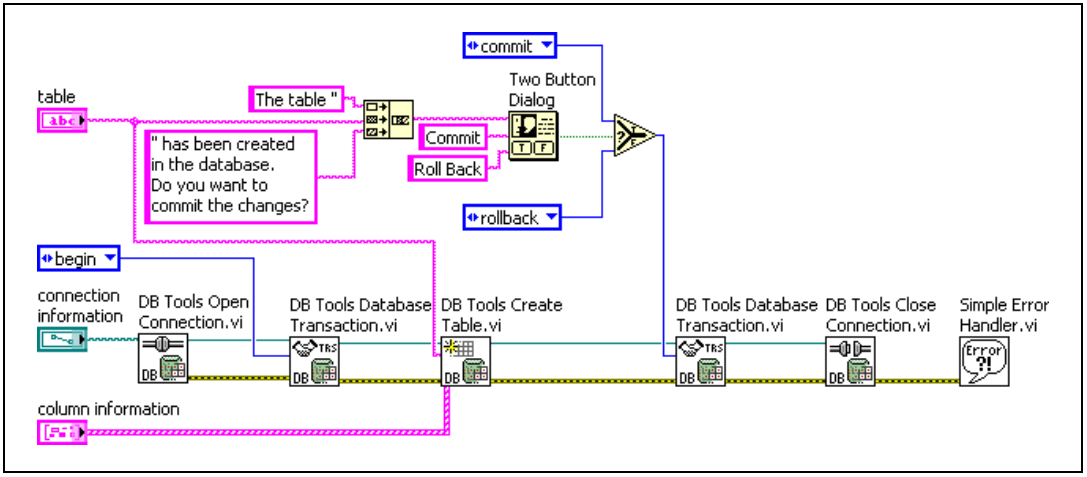

Figure 6-5. Prompting to Commit or Roll Back Database Changes

Figure 6-5 shows how to open a database connection, start a transaction, create a table, prompt the user to either commit or roll back the transaction, and close the database connection. If the user selects to commit the changes, the table is created as specified by the **column information** cluster. If the user selects to roll back the changes, the table is not created. You can use a similar method to protect the data in database tables. You can group operations that belong together into a single transaction and commit the transaction when you are finished or roll back the transaction if an error occurs. You also can use locking to determine who has access to a database during a transaction.

Locking Transactions and Setting Isolation Levels

Locking is an important activity in multi-user database systems where different users have access to the same data at the same time. Without data locking, more than one user can modify the same record at the same time, possibly causing data inconsistencies. Locking allows concurrent database access while minimizing the various problems such access can cause.

Isolation levels represent different locking strategies. The higher the isolation level, the more complex the locking strategy is and the better it is at preventing data inconsistencies. The following data inconsistencies are examples of what different isolation levels try to prevent:

- **Dirty reads**—User 1 modifies data while user 2 uses that same data before user 1 can commit the changes. User 2, therefore, uses incorrect data.
- **Non-repeatable reads**—User 1 reads records while user 2 modifies records. User 1 rereads the records and finds that a record has changed or been deleted.
- **Phantom reads**—User 1 reads records while user 2 adds records. User 1 rereads the records and finds additional records.

At the lowest level of isolation, all the problems mentioned previously can occur. At the highest level of isolation, none of these problems can occur. Different databases support the following different isolation levels:

- **Chaos**—(Lowest level) Transactions are not safe from each other. One transaction might overwrite another.
- **Read Uncommitted**—Locks are obtained on modifications only and held to the end of the transaction. Reading does not involve any locking. Dirty reads, non-repeatable reads, and phantom reads are all possible.

- **Read Committed**—Locks are obtained on reading and modification, but locks are released after reading and held until the end of the transaction for modifications. This transaction cannot see changes made by other transactions until they are committed. Dirty reads are not possible, but non-repeatable reads and phantom reads are possible.
- **Repeatable Read**—Locks are obtained on reading and modifications. Locks are held until the end of the transaction for both reading and modifying records. Locks on non-modified access are released after reading. You do not see any changes in records without re-querying the database. Dirty reads and non-repeatable reads are not possible, but phantom reads are possible.
- **Serializable**—(Highest level) All read or modified data is locked until the end of the transaction. The transaction occurs in complete isolation. Dirty reads, non-repeatable reads, and phantom reads are not possible.

When you choose a higher isolation level, you improve the locking strategy but have less user concurrency.

The DB Tools Database Transaction VI contains an optional input for setting the isolation level used for a transaction. You need to set this value only if other transactions might be pending at the same time. The VI in Figure 6-6 uses the **isolation level** parameter of the DB Tools Database Transaction VI to specify an isolation level for the transaction.

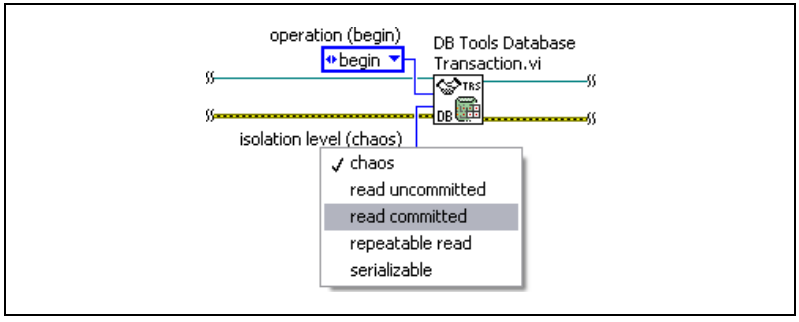

Figure 6-6. Selecting the Isolation Level for a Transaction

Writing and Reading Data Files

You can use the File I/O VIs and functions to write database data to a file just as you do with any other data in LabVIEW. You also can use the DB Tools Save Recordset To File VI to write database data to a file. This VI saves the data as well as the structure and properties of the database recordset. Figure 6-7 shows how you can write recordset data to a file.

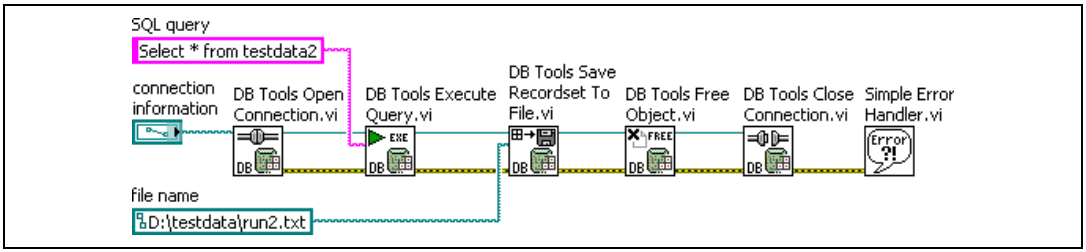

Figure 6-7. Writing Recordset Data to File

The DB Tools Save Recordset To File VI requires a recordset reference input, which you can generate with the DB Tools Execute Query VI. The DB Tools Execute Query VI executes an SQL query. If you specify a table name as the **SQL query** input of the DB Tools Execute Query VI, the query includes all the information in that table. The recordset reference references the results, passes them to the DB Tools Save Recordset To File VI, and writes them to file. The DB Tools Free Object VI releases the recordset reference, and the connection to the database is closed.

The DB Tools Save Recordset To File VI saves recordset data in one of the following formats:

- Extensible Markup Language (XML)
- Advanced Data TableGram (ADTG)

ADTG format is a proprietary Microsoft binary format. ADTG has the advantage of being a compact binary format that results in much smaller files written faster than if you use XML format.

Persisting data to files in the XML and ADTG formats is a feature of ADO and OLE DB and not a special feature of the Database Connectivity Toolkit. You can read data back into LabVIEW from one of these files using the DB Tools Load Recordset From File VI, as shown in Figure 6-8.

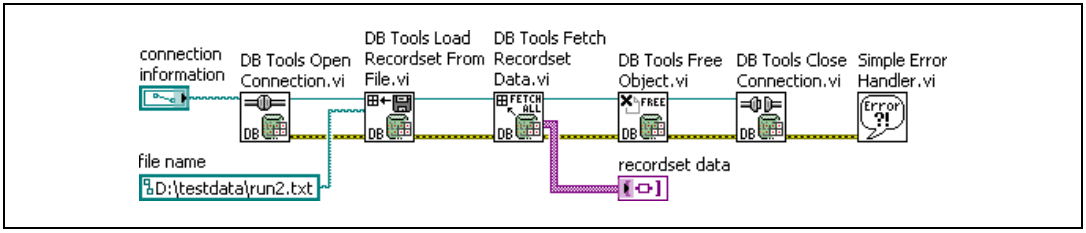

Figure 6-8. Persisting Data from File

The DB Tools Load Recordset From File VI returns a recordset reference when it opens the specified file. You then can use any VI from the Database Connectivity Toolkit that accepts a recordset reference to perform operations on that data. In Figure 6-8, the DB Tools Fetch Recordset Data VI returns the recordset data as a two-dimensional array of variants. The DB Tools Free Object VI releases the recordset reference and closes the database connection.

7

Performing Advanced Database Operations

You can use the Advanced VIs to perform advanced database operations such as executing SQL statements and fetching data. Use the Advanced VIs when you need more control over what is sent to or read from a database.

Executing SQL Statements and Fetching Data

SQL is the language that relational databases use. Common operations you can perform with SQL include creating and deleting tables, inserting data into databases, querying databases for particular recordsets, and manipulating data in tables. Refer to any of the SQL resources listed in the *Related Documentation* section of this manual for more information about SQL. This section describes how you can use SQL statements with the LabVIEW Database Connectivity Toolkit and how you can fetch the data resulting from an SQL query.

Use the DB Tools Execute Query VI to send an SQL string to a database, as shown in Figure 7-1. You then can use the DB Tools Fetch Element Data VI, the DB Tools Fetch Next Recordset VI, or the DB Tools Fetch Recordset Data VI to return the results of a query.

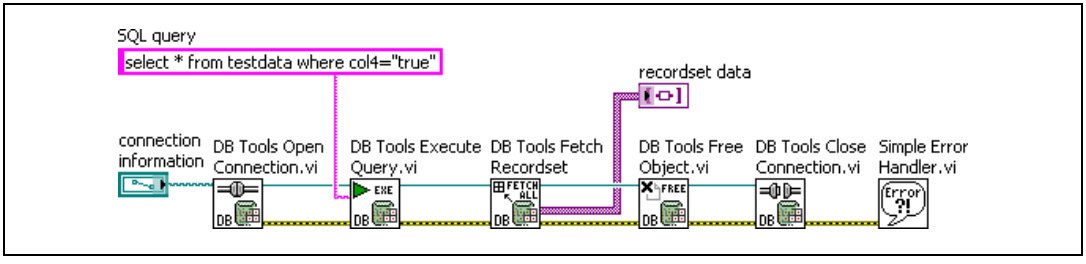

Figure 7-1. Fetching Query Results

The SQL string does not have to specify only a query. You can enter any SQL statement in the SQL string. You also do not need to specify a string if you pass a Command reference to the DB Tools Execute Query VI. When the DB Tools Execute Query VI receives a Command reference input, the

VI executes the previously created SQL query. You can use the DB Tools Create Parameterized Query VI to create a Command reference.

The **SQL query** string shown in Figure 7-1 asks for all records in the `testdata` table where the fifth field contains a TRUE value. The DB Tools Fetch Recordset Data VI returns a two-dimensional array of variants for which all tests passed. Refer to Chapter 5, *Performing Standard Database Operations*, for more information about converting variant data to LabVIEW data types. Because the DB Tools Execute Query VI creates a Recordset reference, you then must use the DB Tools Free Object VI to release the Recordset reference value.

The DB Tools Fetch Recordset Data VI returns all records from a query such as the one shown in Figure 7-1.

Note A record is a single row of data and a recordset is a collection of records, or multiple rows, from a database table.

If you know this query will return a large amount of data or you want to retrieve information from only one record, you can use the DB Tools Fetch Element Data, as shown in Figure 7-2.

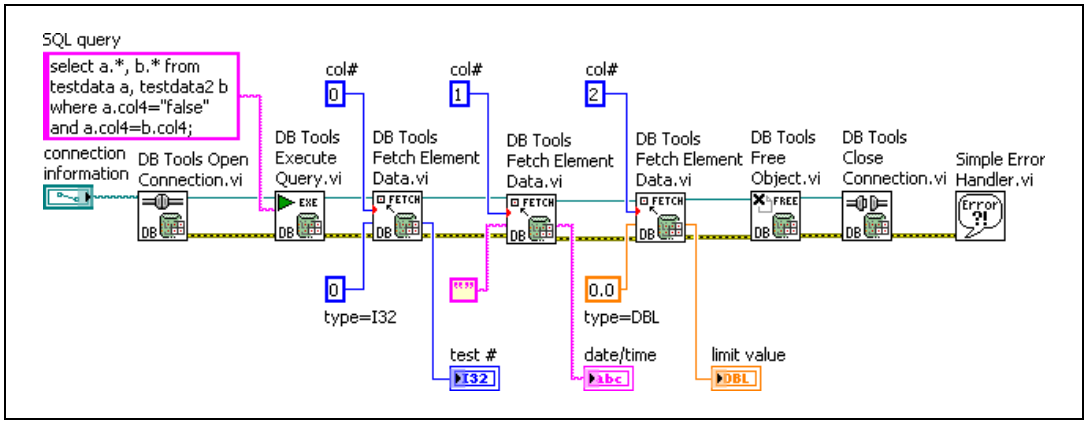

Figure 7-2. Fetching Query Results from One Record

The DB Tools Fetch Element Data VI returns a value from a single field. You can specify the field, or column, either by a numerical value, as shown in Figure 7-2, or by a string specifying the column name. You also must specify the data type because the DB Tools Fetch Element Data VI uses the Database Variant To Data function.

Notice the SQL query used in Figure 7-2. This query is an example of an SQL inner join operation. An inner join operation combines the fields of several tables through a common value or expression. The records in the `testdata` and `testdata2` tables are combined for all the tests where both tables contain failed tests.

Whereas the DB Tools Fetch Recordset Data VI returns all records that satisfy an SQL query, the DB Tools Fetch Element Data VI returns an element from the first record that satisfies the query. Use the VIs described in the *Navigating Database Records* section of this chapter to navigate through the resulting records. Also, some SQL queries, such as stored procedures, return multiple recordsets. Use the DB Tools Fetch Next Recordset VI to read each recordset.

Note Not all databases support queries that return multiple recordsets. The inner join statement in Figure 7-2 returns a single recordset that happens to contain the results from multiple tables. Therefore, the recordset might contain several records, or rows, where the columns from multiple tables have been joined.

Navigating Database Records

Operations in a relational database act on a complete set of rows. The recordset returned by an SQL SELECT statement consists of all rows that satisfy the conditions of the statement. Applications, especially interactive and online applications, sometimes cannot work effectively with the entire recordset as a unit. Use cursors to allow applications that cannot work with the entire recordset as a unit to work with one row at a time.

Note The Database Connectivity Toolkit does not require you to know about cursors in order to use them. However, the following information can help advanced users who want to have more control over their applications.

Using Cursors

A cursor is a placeholder that points to a specific record in a recordset. A cursor keeps track of the position in the recordset and allows you to perform multiple operations row by row against a recordset, with or without returning to the original table. Every cursor uses temporary resources to hold its data. These resources can be memory, a disk paging file, temporary disk files, or even temporary storage in the database. Cursors can reside in one of the following two locations:

- **Client-side cursor**—The temporary storage resources are located on the client computer. In addition, the client receives the entire database

recordset across the network. Client-side cursors lead to very quick database operations because everything happens locally on the client machine. However, when you work with large databases, a client-side cursor can be extremely expensive in time and memory use because the client machine must receive all data from the server. Also, only the static cursor type is supported by client-side cursors. Refer to the *Cursor Types* section of this chapter for more information about cursor types.

- **Server-side cursor**—The temporary storage resources are located on the database server machine. The server-side cursor returns only the requested data over the network. Server-side cursors provide better performance than the client-side cursor when you work with large databases or in situations where excessive network traffic is a problem. You have a choice of four different cursor types when you use a server-side cursor.

The Database Connectivity Toolkit uses only server-side cursors because they offer more flexibility and because they provide better performance for large amounts of data.

Cursor Types

Although you can use only a server-side cursor with the Database Connectivity Toolkit, you do have a choice of server-side cursor types. The type of cursor used by your application to navigate the recordset affects the ability to move forward and backward through the rows in a recordset, sometimes called scrollability. Scrollability adds to the time and resources necessary to use the cursor. Use the simplest cursor that provides the required data access and only change the cursor type if you absolutely need the added functionality. Set the cursor type by creating a **cursor type** constant for the DB Tools Execute Query VI and then selecting from the choices, as shown in Figure 7-3.

Note Not all data providers or databases support all the cursor types shown in Figure 7-3. For example, the Jet 4.0 OLE DB Provider for Microsoft Access does not support dynamic cursors. If you request a dynamic cursor, the provider returns a static cursor that is not correctly implemented.

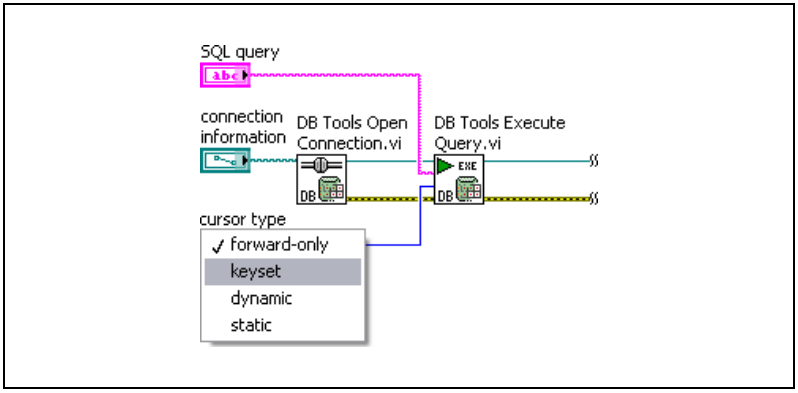

Figure 7-3. Possible Cursor Types

The following cursor types are available:

- **Forward-only**—(Default) This cursor permits only forward movement through the recordset. Any changes made to the database by other users during navigation will not be seen. Forward-only cursors are dynamic because detection of changes occurs as the current row is processed. This is a high-performance cursor that uses the least resources.
- **Keyset**—This cursor allows forward and backward navigation. You can see records added by other users, but records deleted by others will not be removed from view.
- **Dynamic**—This cursor allows forward and backward navigation. You can see all changes made, both locally and by other users, to the database. Use the dynamic cursor if your application must detect all concurrent updates made by other users.
- **Static**—This cursor allows forward and backward navigation with no ability to see any changes made by other users during navigation. The static cursor always displays the result set as it was when the cursor was first opened. Use the static cursor if your application does not need to detect data changes and requires scrolling.

Choose a cursor depending on whether you need to change or simply view the data. If you just need to scroll through a set of results but not change data, use a forward-only or static cursor. If you have a large result set and need to select just a few rows, use a keyset cursor. If you want to synchronize a result set with recent adds, changes, and deletes by all concurrent users, use a dynamic cursor.

Navigating Recordsets

Use the DB Tools Move To Next Record VI, the DB Tools Move To Previous Record VI, and the DB Tools Move To Record N VI to navigate the results of a database query. Figures 7-4 and 7-5 show how you can scroll forward and backward through a recordset using a static cursor.

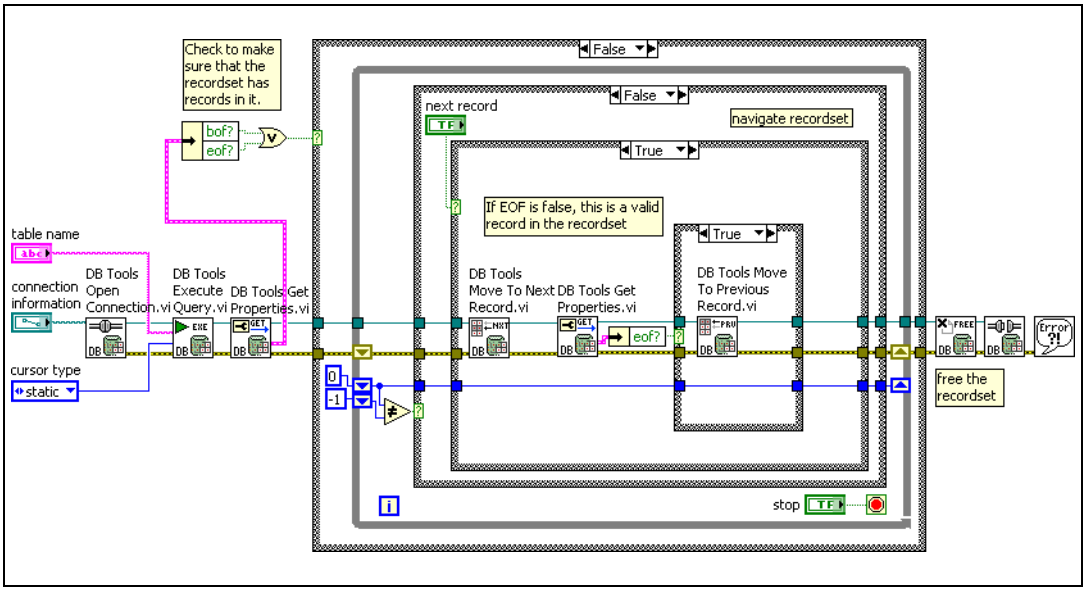

Figure 7-4. Scrolling through a Recordset Using a Static Cursor

In Figure 7-4, the DB Tools Open Connection VI opens the connection to a database. The DB Tools Execute Query then opens a table and specifies a static cursor. Usually, you use the DB Tools Execute Query to send an SQL statement to a database. However, if you send the table name to this VI, the VI returns a recordset reference to all the records in that table. The DB Tools Get Properties VI returns the beginning of file (BOF) and the end of file (EOF) markers to make sure that the table contains records.

The front panel of the VI shown in Figure 7-4 contains two buttons on the panel labeled **Next Record** and **Previous Record**. Clicking the **Next Record** button calls the DB Tools Move To Next Record VI, and the DB Tools Get Properties VI then reads the EOF property. If the end of file is reached, the DB Tools Move To Previous Record VI is called, and the cursor then points to the last record in the table.

Figure 7-5 shows what happens when you click the **Previous Record** button.

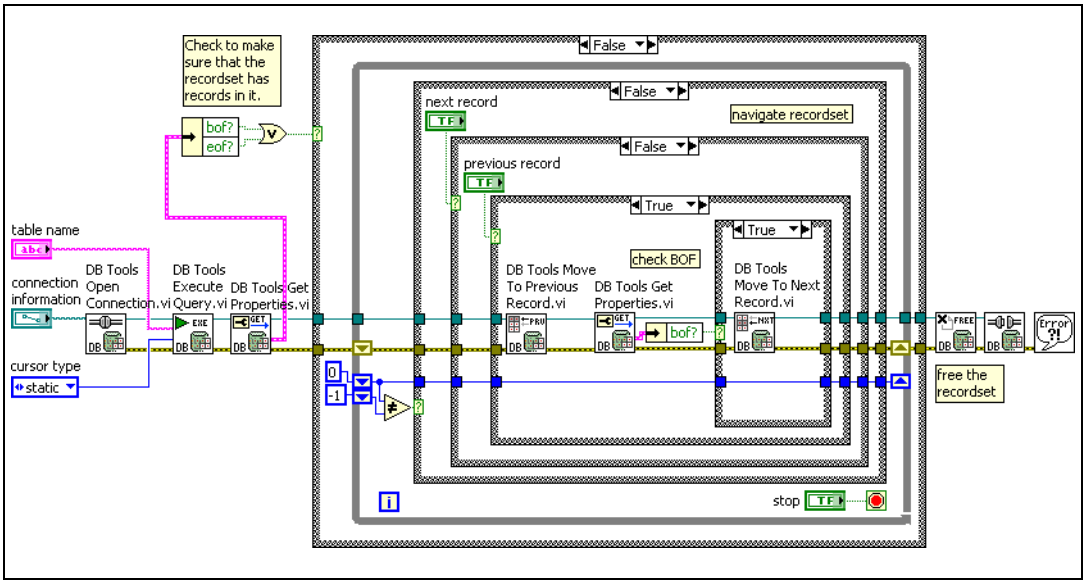

Figure 7-5. Navigating to the Previous Record in a Recordset

The VI in Figure 7-5 calls the DB Tools Move To Previous Record VI, and then the DB Tools Get Properties VI reads the BOF property. If the beginning of file is reached, the DB Tools Move To Next Record VI is called, and the cursor then points to the first record in the table. Notice that you need to use the static cursor type in these examples because you are scrolling forward and backward in the recordset.

Figure 7-6 shows how to use the DB Tools Move To Record N VI to display the information from any record in a table when you know the record number. The first record in the recordset has a value of zero.

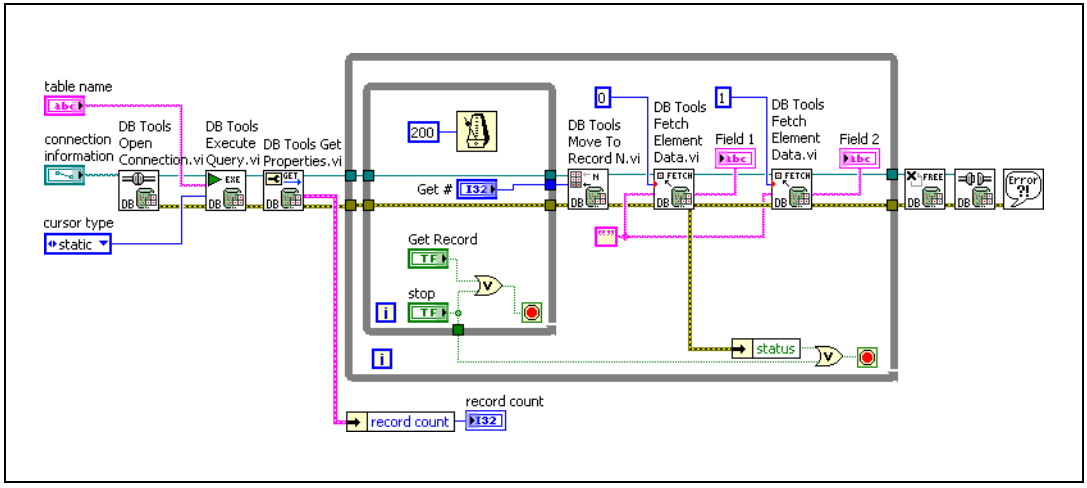

Figure 7-6. Navigating to the *n*th Record in a Recordset

In Figure 7-6, the DB Tools Get Properties VI returns the number of records in the table and displays that value in the panel. You also can use this value to make sure the user does not enter a value for **Get #** that is not a valid record number. The VI ends with an error message if you ask for a record number that does not exist. The static cursor allows scrollability through the entire recordset.

Using Parameterized Statements

Parameterized statements allow you to specify an SQL statement once but vary the parameters, such as the matching criteria of a WHERE clause, over time. Prepare a parameterized statement using the DB Tools Create Parameterized Query VI, as shown in Figure 7-7.

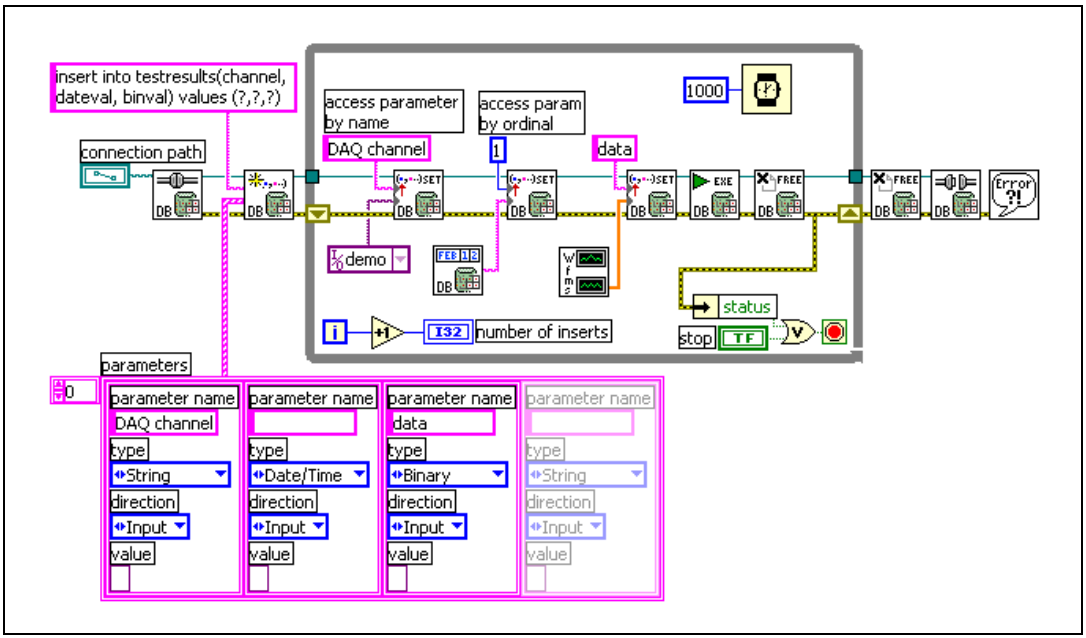

Figure 7-7. Writing Parameterized Data to a Table

You can create the **parameters** input of the DB Tools Create Parameterized Query VI either by creating a control on the panel or by creating a constant on the block diagram. The **parameters** input is an array of clusters where each array value represents a column or field in the database table. Each parameter cluster contains the following four values:

- **parameter name**—Leave this string empty if the parameter is not named.
- **type**—Specify whether the parameter data type is string, long, single, double, date/time, or binary.
- **direction**—Specify whether the parameter is an input, output, input/output, or return value.

- **value**—Specify the initial value of the parameter. You also can leave this value as an empty variant and set the value later with the DB Tools Set Parameter Value VI, as shown in Figure 7-7.

Figure 7-7 shows the SQL statement `insert into testresults (channel, dateval, binval) values (?, ?, ?)`. This statement indicates that a table named `testresults` already exists and contains three fields, or columns, named `channel`, `dateval`, and `binval`. The question marks represent parameters to be set later in the data acquisition loop. The **parameters** input specifies the names of the parameters and their data types. These data types must match the data types as defined by the `testresults` database table. The DB Tools Create Parameterized Query VI returns an error if the number of parameters described in the SQL statement do not match the number of array elements in the **parameters** input, if the column names in the SQL statement do not match the column names in the `testresults` database table, or if the data types defined in the **parameters** array cluster do not match the data types in the `testresults` database table.

The VI in Figure 7-7 calls the DB Tools Open Connection VI and the DB Tools Create Parameterized Query VI before the data acquisition loop begins. Inside the acquisition loop, the DB Tools Set Parameter Value VI writes the data to the database. Notice that you can specify the parameter index input as either a string specifying the parameter name or as a number representing the index. After setting all the parameter values, the DB Tools Execute Query VI executes the statement. The DB Tools Free Object VI and the DB Tools Close Connection VI release the various reference values and close the database connection.

The example in Figure 7-7 can be viewed in a different way by taking the Microsoft ActiveX Data Object (ADO) object reference types into account. The DB Tools Open Connection VI creates a Connection reference. The DB Tools Create Parameterized Query VI uses the Connection reference and creates a Command reference. The Command reference passes through the DB Tools Set Parameter Value VI, and the DB Tools Execute Query VI changes it to a Command-Recordset reference. The first DB Tools Free Object VI takes the Command-Recordset reference and returns a Command reference. The second DB Tools Free Object VI takes the Command reference and returns a Connection reference. Last, the DB Tools Close Connection VI releases the Connection reference. Each time an ADO object reference opens, you must call the DB Tools Free Object to close it. Refer to the *ADO Reference Classes* section of Chapter 6, *Using the Database Connectivity Toolkit Utility VIs*, for more information about ADO object reference classes.

You can use parameters in any kind of SQL statement. However, not all databases or data providers support parameterized statements. Refer to your ADO, data provider, or database documentation for more information about what features are supported.

Using Stored Procedures

A stored procedure is a precompiled collection of SQL statements and optional control-of-flow statements, similar to a macro. Each database and data provider supports stored procedures differently. For example, you can create a stored procedure using the Jet 4.0 provider, but Access does not support stored procedures through its usual user interface. A stored procedure created in one DBMS might not work with another. You can use the Database Connectivity Toolkit to create and run stored procedures, both with and without parameters.

Although using stored procedures is an advanced task, stored procedures offer the following benefits to your database applications:

- **Performance**—Stored procedures are usually more efficient and faster than regular SQL queries because SQL statements are parsed for syntactical accuracy and precompiled by the DBMS when the stored procedure is created. Also, combining a large number of SQL statements with conditional logic and parameters into a stored procedure allows the procedures to perform queries, make decisions, and return results without extra trips to the database server.
- **Maintainability**—Stored procedures isolate the lower-level database structure from the LabVIEW application. As long as the table names, column names, parameter names, and types do not change from what is stated in the stored procedure, you do not need to modify the procedure when changes are made to the database schema. Stored procedures are also a way to support modular SQL programming because after you create a procedure, you and other users can reuse that procedure without knowing the details of the tables involved.
- **Security**—When creating tables in a database, the Database Administrator can set EXECUTE permissions on stored procedures without granting SELECT, INSERT, UPDATE, and DELETE permissions to users. Therefore, the data in these tables is protected from users who are not using the stored procedures.

Creating Stored Procedures

You usually create stored procedures in the DBMS environment. Some DBMSs, such as SQL Server, contain a library of system stored procedures that perform common administrative tasks with databases. The names of these stored procedures begin with `sp_`. Refer to the documentation for your DBMS for the exact syntax to use when creating a stored procedure. You also can use the Database Connectivity Toolkit to create stored procedures, as shown in Figure 7-8.

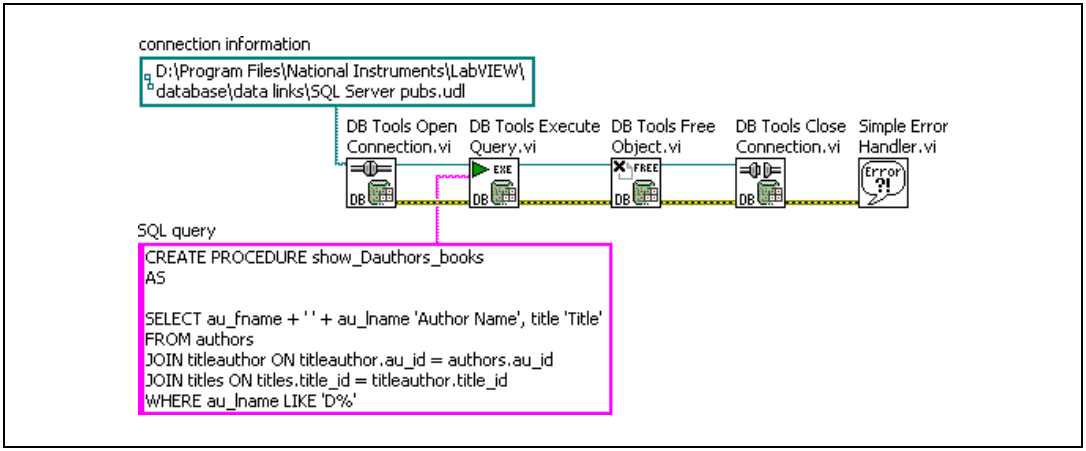

Figure 7-8. Creating a Stored Procedure

Figure 7-8 uses the same VIs as you use to perform a typical SQL query. However, the syntax of the **SQL query** string is different. The **SQL query** string is a stored procedure that calls the `show_Dauthors_books` query. The query string specifies the use of three tables in the `pubs` database. The procedure joins the `authors`, `titles`, and `titleauthor` tables to create a list of all the authors whose last name begins with D and a list of the books they have published. The procedure also arranges the result, where the first column combines the first and last names and the second column contains the book title. When the VI runs, it creates a stored procedure that does not use parameters. The *Running Stored Procedures without Parameters* section of this chapter describes how you then can call the stored procedure using another VI.

Running Stored Procedures without Parameters

You can run a stored procedure by inserting the name of the procedure as an SQL query. Figures 7-9 and 7-10 show how you can call the stored procedure created in Figure 7-8 using the DB Tools Execute Query VI.

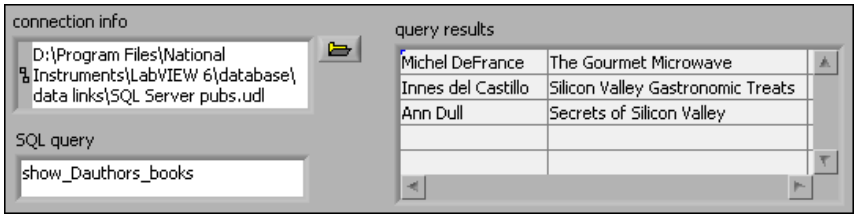

Figure 7-9. Front Panel Showing How to Run a Stored Procedure

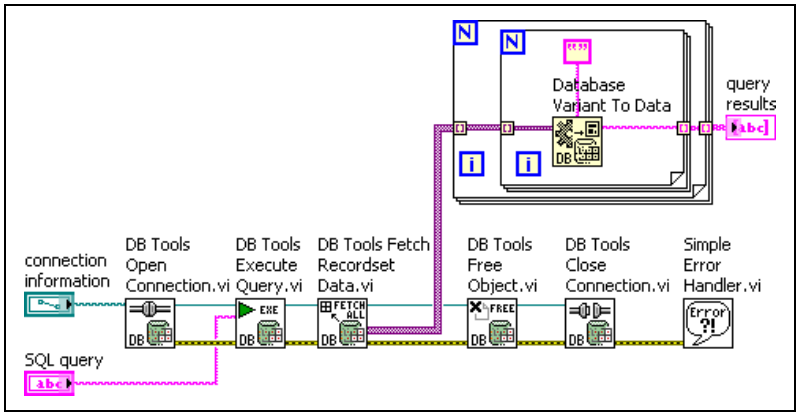

Figure 7-10. Block Diagram Showing How to Run a Stored Procedure

In Figure 7-10, the DB Tools Execute Query VI sends the name of the stored procedure. The DB Tools Fetch Recordset Data returns the results of the stored procedure in a two-dimensional array of variants. The nested For Loops convert the variants to strings so the results can be displayed in a LabVIEW table.

Running Stored Procedures with Parameters

Stored procedures can use variables internally as well as pass parameters into and out of the procedure. You can use parameters with stored procedures in two ways. In the first method, you build SQL query strings that contain the name of the stored procedure with the values embedded at the appropriate places in the query. For example, assume you want to use the following stored procedure:

```
CREATE PROCEDURE AddPart
       @part_name char(40),
       @part_qty int,
       @part_price money,
       @part_descr varchar(255) = NULL
AS
    INSERT parts (name, qty, price, description)
    VALUES (@part_name, @part_qty, @part_price,
    @part_descr)
```

This stored procedure adds a record containing the part name, quantity, unit price, and description to the table named `parts`. Figure 7-11 shows the block diagram that calls this `AddPart` procedure.

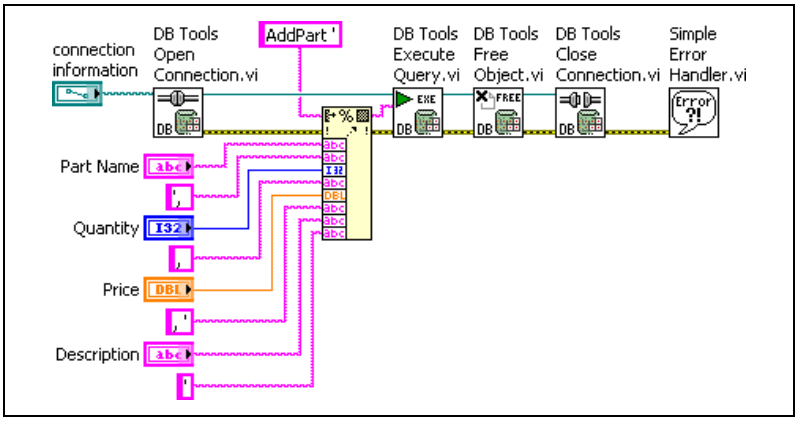

Figure 7-11. Using Parameters with a Stored Procedure

Figure 7-11 shows how you construct the SQL query string using the Format Into String function. For example, if **Part Name** is `widget`, **Quantity** is `24`, **Price** is `0.99`, and **Description** is `misc parts`, the Format Into String function constructs the following SQL query string: `AddPart 'widget', 24, 0.99, 'misc parts'`. The DB Tools Execute Query VI sends this query to the database just as you would send

any other SQL query. You must know the parameters and data types used in a stored procedure in order to call it properly from the Database Connectivity Toolkit.

The second way to use parameters with stored procedures is to use the DB Tools Create Parameterized Query VI, the DB Tools Set Parameter Value VI, and the DB Tools Get Parameter Value VI. Figure 7-12 shows how you can use these three VIs to run the same stored procedure as in Figure 7-11.

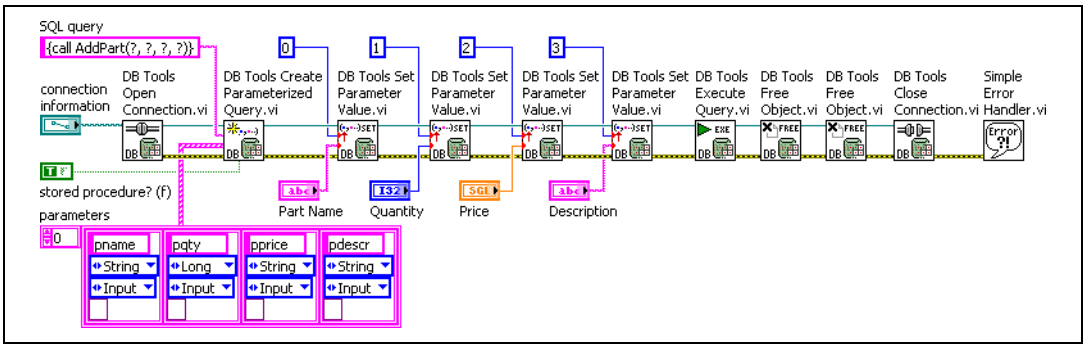

Figure 7-12. Using a Parameterized Stored Procedure

The format for this SQL query is slightly different than in the previous example. The query string shown in Figure 7-12 uses the ODBC method for calling a stored procedure whereas the previous example used the Transact SQL (T-SQL) method used by SQL Server. The parameters in the **SQL query** string are defined by question marks. The **parameters** array must contain as many elements as there are question marks in the query. You specify the parameter name, data type, direction (input, output, input/output, or return value), and parameter value in each **parameter** array element. You also set the **stored procedure?** input of the DB Tools Create Parameterized Query VI to TRUE.

If you do not explicitly state values in the **parameters** array, use the DB Tools Set Parameter Value VI, as shown in Figure 7-12. The DB Tools Set Parameter Value VI is polymorphic and can accept any LabVIEW data type for the value input. The previous example shows string, integer, and single values wired to the DB Tools Set Parameter Value VI. You can specify the **parameter** index as either a numeral, as shown above, or as the parameter name defined in the **parameters** array. After all the parameters are defined, the DB Tools Execute Query VI runs the parameterized query. The recordset and command references are freed with the two DB Tools Free Object VIs, and the database connection is closed.

The stored procedure examples shown in this section are specifically written for SQL Server. Oracle uses PL/SQL to create stored procedures. Although the syntax for PL/SQL is different, you still can create and run stored procedures for Oracle using the Database Connectivity Toolkit.

8

Building Applications

You can build applications or shared libraries that use the LabVIEW Database Connectivity Toolkit. The Database Connectivity Toolkit requires some additional options, such as DSN or UDL files, that are not part of a standard application build routine.

Note Ensure that the target computer to which you want deploy the built application or shared library contains the Microsoft Data Access Components (MDAC).

Right-click a build specification and select **Properties** from the shortcut menu to verify and edit the build specification settings. Be sure to specify any necessary DSN or UDL files on the **Source Files** page of the build specification **Properties** dialog box. Then click the **Build** button to build the application or shared library and to update the project with the build specification settings.

Using UDLs and DSNs

When you build an application that includes Database VIs, you must include the UDL and DSN files for the database connection in the application.

User and system DSNs are applicable only to a particular user or computer. You must create user and system DSNs manually on the target machine using the ODBC Data Source Administrator. Refer to the *ODBC Data Source Administrator* section of Chapter 3, *Connecting to a Database*, for information about the ODBC Data Source Administrator.

If you want to include a UDL in an application, you must add the UDL to the LabVIEW project that contains your build specification. Right-click a target, such as the **My Computer** target, in the **Project Explorer** window and select **Add»File** from the shortcut menu to add a UDL to the project.

After you add the UDL to the project, right-click the application build specification and select **Properties** from the shortcut menu to display the **Application Properties** dialog box. On the **Source Files** page, add the

UDL you want to include in the built application to the **Always Included** list. You also can include the database itself as an included file.

Note If you build an application or shared library that uses a UDL to access a database, the path to the database in the application or shared library might be different from the path to the database in the LabVIEW development environment. Be sure to modify the UDL to map to the database in the target location before building the application or shared library, or add code to modify the UDL as part of the initialization of the application.

Using Connection Strings

Rather than including an existing UDL in an application, you also can use an ODBC connection string with the Microsoft ActiveX Data Object (ADO) standard. You can modify your application to create a connection string that works in either the development environment or the target environment. Refer to the Microsoft Developer Network Web site at `http://msdn.microsoft.com` for more information about using and configuring connection strings in ADO.

Refer to the Using Connection Strings project, located in the `labview\examples\database\EXE` directory, for an example of using a connection string to connect to a database in a built application.

A

Technical Support and Professional Services

Visit the following sections of the award-winning National Instruments Web site at ni.com for technical support and professional services:

- **Support**—Technical support resources at ni.com/support include the following:
  - **Self-Help Technical Resources**—For answers and solutions, visit ni.com/support for software drivers and updates, a searchable KnowledgeBase, product manuals, step-by-step troubleshooting wizards, thousands of example programs, tutorials, application notes, instrument drivers, and so on. Registered users also receive access to the NI Discussion Forums at ni.com/forums. NI Applications Engineers make sure every question submitted online receives an answer.
  - **Standard Service Program Membership**—This program entitles members to direct access to NI Applications Engineers via phone and email for one-to-one technical support as well as exclusive access to on demand training modules via the Services Resource Center. NI offers complementary membership for a full year after purchase, after which you may renew to continue your benefits.

    For information about other technical support options in your area, visit ni.com/services, or contact your local office at ni.com/contact.
- **Training and Certification**—Visit ni.com/training for self-paced training, eLearning virtual classrooms, interactive CDs, and Certification program information. You also can register for instructor-led, hands-on courses at locations around the world.
- **System Integration**—If you have time constraints, limited in-house technical resources, or other project challenges, National Instruments Alliance Partner members can help. To learn more, call your local NI office or visit ni.com/alliance.

If you searched ni.com and could not find the answers you need, contact your local office or NI corporate headquarters. Phone numbers for our worldwide offices are listed at the front of this manual. You also can visit the Worldwide Offices section of ni.com/niglobal to access the branch office Web sites, which provide up-to-date contact information, support phone numbers, email addresses, and current events.