DataDirect

Connect® Series *for* ODBC

User's Guide and Reference

Release 5.1
April 2006

Table of Contents

List of Tables 7

Preface 9

Using this Book 9

Conventions Used in this Book 11

About the Product Documentation 12

1 Quick Start Connect 13

Configuring and Connecting on UNIX and Linux 13

2 Using The Product 17

What Is ODBC? 17

Environment-Specific Information 19

Binding Parameter Markers 24

Version String Information 25

Retrieving Data Type Information 26

3 The NSK SQL/MX Wire Protocol Driver 29

Driver Requirements 29

Configuring Data Sources 29

Connecting to a Data Source Using a Connection String 30

Data Types 33

Persisting a Result Set as an XML Data File 35

Isolation and Lock Levels Supported 37

ODBC Conformance Level 37
Number of Connections and Statements Supported 38

A ODBC API and Scalar Functions 39

API Functions 39
Scalar Functions 42

B Locking and Isolation Levels 51

Locking 51
Isolation Levels 52
Locking Modes and Levels 55

C Threading 57

Driver Threading Information 58

D Using Indexes 59

Introduction 59
Improving Record Selection Performance 61
Indexing Multiple Fields 61
Deciding Which Indexes to Create 63
Improving Join Performance 65

E Performance Design of ODBC Applications 67

Using Catalog Functions 68
Retrieving Data 72
Selecting ODBC Functions 77
Managing Connections and Updates 81

F Values for IANAAppCodePage Connection String Attribute 87

G The UNIX/Linux Environments 93

Environment Variables 93

The ivtestlib/ddtestlib Tool 96

Data Source Configuration 97

demoodbc 99

example 100

DSN-less Connections 101

File Data Sources 102

UTF-16 Applications on UNIX and Linux 104

H Diagnostic Tools, Error Messages, and Troubleshooting 105

Diagnostic Tools 105

Error Messages 108

Troubleshooting 110

Glossary 115

Index 119

List of Tables

Table 3-1. NSK SQL/MX Wire Protocol Connection String Attributes 32
Table 3-2. NSK SQL/MX Data Types 33
Table A-1. Function Conformance for 2.x ODBC Applications 40
Table A-2. Function Conformance for 3.x ODBC Applications 41
Table A-3. Scalar String Functions 43
Table A-4. Scalar Numeric Functions 45
Table A-5. Scalar Time and Date Functions 47
Table A-6. Scalar System Functions 49
Table B-1. Isolation Levels and Data Consistency 54
Table E-1. Common Performance Problems Using ODBC Applications 67
Table F-1. IANAAppCodePage Values 88

Preface

This book is your user's guide and reference to the DataDirect Connect® Series *for* ODBC drivers (DataDirect Connect *for* ODBC and DataDirect Connect64® *for* ODBC) from DataDirect Technologies™. This product includes database *drivers* that are compliant with the Open Database Connectivity (ODBC) specification.

Using this Book

The content of this book is based on the assumption that you are familiar with your operating system and its commands. It contains the following chapters:

- A quick start chapter (Chapter 1 "Quick Start Connect" on page 13) that explains the basics for quickly configuring and testing the drivers.
- An introductory chapter (Chapter 2 "Using The Product" on page 17) that explains the drivers and ODBC, and discusses environment-specific subjects.
- A chapter for the NSK SQL/MX ODBC driver. First, it lists which versions of the databases the driver supports, the operating environments on which the driver runs, and the driver requirements for your operating environment. Next, it explains how to configure a data source and how to connect to that data source. Finally, the chapter provides information about data types, ODBC conformance levels, isolation and lock levels supported, and other driver-specific information.

This book also includes several appendixes that provide information on technical topics:

- Appendix A "ODBC API and Scalar Functions" on page 39 lists the supported ODBC API functions. Any exceptions are listed in the driver chapter, under the section "ODBC Conformance Level." This appendix also lists the ODBC scalar functions.
- Appendix B "Locking and Isolation Levels" on page 51 provides a general discussion of isolation levels and locking.
- Appendix D "Using Indexes" on page 59 provides general guidelines on how to improve performance when querying a database system.
- Appendix C "Threading" on page 57 discusses how ODBC ensures thread safety.
- Appendix E "Performance Design of ODBC Applications" on page 67 provides guidelines for designing performance-oriented ODBC applications.
- Appendix G "The UNIX/Linux Environments" on page 93 discusses UNIX and Linux, environment variables and configuration of the drivers. It also explains the structure of the *system information file* (used in the UNIX and Linux environments) and provides a sample system information file, as well as discussing other driver tools for UNIX and Linux.
- Appendix F "Values for IANAAppCodePage Connection String Attribute" on page 87 provides the valid values for the IANAAppCodePage connection string attribute.
- Appendix H "Diagnostic Tools, Error Messages, and Troubleshooting" on page 105 discusses the diagnostic tools that are available, explains error messages, and provides a troubleshooting section.
- A "Glossary" on page 115.

If you are writing programs to access ODBC drivers, you need to obtain a copy of the *ODBC Programmer's Reference* for the Microsoft Open Database Connectivity Software Development Kit, available from Microsoft Corporation.

NOTE: This book refers the reader to Web URLs for more information about specific topics, and may include Web URLs not maintained by DataDirect Technologies. Because it is the nature of Web content to change frequently, DataDirect Technologies can guarantee only that the URLs referenced in this book were correct at the time of publishing.

Conventions Used in this Book

The following sections describe the typography, terminology, and other conventions used in this book.

Typographical Conventions

This book uses the following typographical conventions:

Convention	Explanation
italics	Introduces new terms with which you may not be familiar, and is used occasionally for emphasis.
bold	Emphasizes important information. Also indicates button, menu, and icon names on which you can act. For example, click **Next**.
UPPERCASE	Indicates the name of a file. For operating environments that use case-sensitive file names, the correct capitalization is used in information specific to those environments. Also indicates keys or key combinations that you can use. For example, press the ENTER key.

Convention	Explanation
monospace	Indicates syntax examples, values that you specify, or results that you receive.
monospaced italics	Indicates names that are placeholders for values that you specify. For example, *filename*.
forward slash /	Separates menus and their associated commands. For example, Select File / Copy means that you should select Copy from the File menu. The slash also separates directory levels when specifying locations under UNIX.
vertical rule |	Indicates an "OR" separator used to delineate items.
brackets [ ]	Indicates optional items. For example, in the following statement: SELECT [DISTINCT], DISTINCT is an optional keyword. Also indicates sections of the Windows Registry.
braces { }	Indicates that you must select one item. For example, {yes | no} means that you must specify either yes or no.
ellipsis . . .	Indicates that the immediately preceding item can be repeated any number of times in succession. An ellipsis following a closing bracket indicates that all information in that unit can be repeated.

About the Product Documentation

DataDirect product documentation is provided in PDF format, which allows you to view the books online or print them. You can view DataDirect online documentation using Adobe Acrobat Reader 4.*x* or higher.

1 Quick Start Connect

This chapter provides basic information for configuring and test connecting with your DataDirect Connect Series *for* ODBC drivers immediately after installation. To take full advantage of the features of the drivers, we recommend that you read Chapter 2 "Using The Product" and the NSK SQL/MX driver chapter.

Information that the drivers needs to connect to a database is stored in a *data source*. The ODBC specification describes three types of data sources: user data sources, system data sources (not a valid distinction on UNIX and Linux), and file data sources. On Windows, user and system data sources are stored in the registry of the local computer. The difference is that only a specific user can access user data sources, whereas any user of the machine can access system data sources. On Windows, UNIX, and Linux, file data sources, which are simply text files, can be stored locally or on a network computer, and are accessible to other machines.

When you define and configure a data source, you store default connection values for the drivers that are used each time you connect to a particular database. You can change these defaults by modifying the data source.

Configuring and Connecting on UNIX and Linux

The following basic information enables you to configure a data source and test connect with a driver immediately after installation. See Appendix G "The UNIX/Linux Environments" on page 93 for detailed information about how to define and configure a data source in the UNIX and Linux environments.

Environment Setup

1. Check your permissions: You should log in as a user with full r/w/x permissions recursively on the entire product installation directory.
2. Determine which shell you are running by executing the env command.
3. Run the DataDirect Technologies setup script to set variables. Two scripts, odbc.sh and odbc.csh, are installed in the installation directory. For Korn, Borne, and equivalent shells, execute odbc.sh. For a C shell, execute odbc.csh. After running the setup script, execute the env command to verify that the *installation_directory*/lib directory has been added to your shared library path.
4. Set the ODBCINI environment variable. The variable should point to the path from the root directory to the system information file where your data source will reside. The system information file can have any name, but the product is installed with a default template file called odbc.ini in the installation directory. For example, if you use the default installation directory and the default system information file, from the Korn or Borne shell you would enter:

   ```
   ODBCINI=/opt/odbc/odbc.ini; export ODBCINI
   ```

Test Loading the Driver

The 32-bit driver tool, ivtestlib, is described in the following section. For 64-bit drivers, substitute the name ddtestlib for the tool and ddnsk21.so for the driver. The tool is located in the *installation_directory*/bin directory, and is a utility to verify that the driver can be loaded into memory. For example, to load a 32-bit driver you would enter:

```
ivtestlib /opt/odbc/lib/driver_shared_object_name
```

where *driver_shared_object_name* is the name of the specific driver file. For example, the NSK SQL/MX Wire Protocol on Linux is ivnsk21.so.

An error message is returned if the load is not successful.

Configuring a Data Source in the System Information File

In the UNIX and Linux environments, there is no ODBC Administrator. To configure a data source in the UNIX and Linux environments, you must edit the system information file (by default, odbc.ini) to which the ODBCINI variable points. The default odbc.ini installed by Setup in the installation directory is a template into which you enter your site-specific database connection information. Using a text editor, you modify the default attributes in this file as necessary, based on your system values (for example, your server name and port number).

To configure a file data source, you must create a text file that contains the required data source information in a format very similar to the odbc.ini file.

Consult the "Connection String Attributes" table of the driver chapter for specific connection attribute values to use in creating data sources. See Appendix G "The UNIX/Linux Environments" on page 93 for details about how to create and edit these files.

IMPORTANT: The "Connection String Attributes" table of the driver chapter lists both the long and short name of the attribute. When entering attribute names into data source files, you must use the long name of the attribute. The short name is not valid in the odbc.ini file.

Testing the Connection

The product installation includes an ODBC application called example that can be used to connect to a data source and execute SQL. The application is located in the *installation_directory*/example directory.

To run the program after setting up a data source in the odbc.ini, enter `example` and follow the prompts to enter your data source name, user name, and password. If successful, a `SQL>` prompt appears and you can type in SQL statements such as `SELECT * FROM table`. If example is unable to connect, the appropriate error message appears.

2 Using The Product

This chapter contains the following sections:

- "What Is ODBC?"
- "Environment-Specific Information"
- "Binding Parameter Markers"
- "Version String Information"
- "Retrieving Data Type Information"

What Is ODBC?

The Open Database Connectivity (ODBC) interface by Microsoft allows applications to access data in database management systems (DBMS) using SQL as a standard for accessing the data. ODBC permits maximum interoperability, which means a single application can access different DBMS. Application end users can then add ODBC database drivers to link the application to their choice of DBMS.

The ODBC interface defines:

- A library of ODBC function calls of two types:
  - Core functions that are based on the X/Open and SQL Access Group Call Level Interface specification
  - Extended functions that support additional functionality, including scrollable cursors
- SQL syntax based on the X/Open and SQL Access Group SQL CAE specification (1992)
- A standard set of error codes

- A standard way to connect and logon to a DBMS
- A standard representation for data types

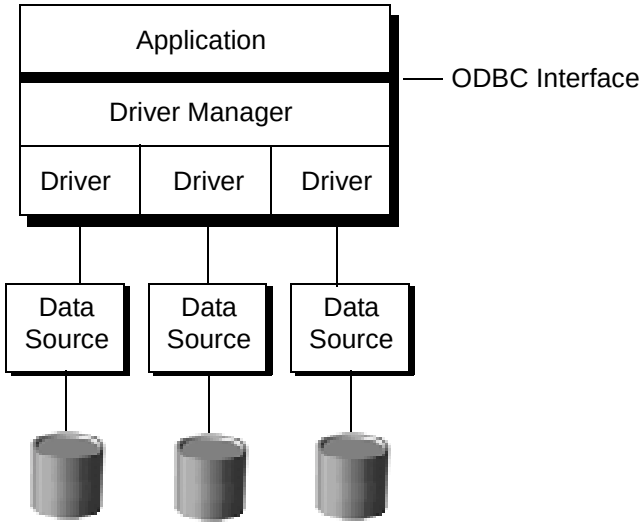
The ODBC solution for accessing data led to ODBC database drivers, which are dynamic-link libraries on Windows and shared objects on UNIX and Linux. These drivers allow an application to gain access to one or more data sources. ODBC provides a standard interface to allow application developers and vendors of database drivers to exchange data between applications and data sources.

How Does It Work?

The ODBC architecture has four components:

- Application, which processes and calls ODBC functions to submit SQL statements and retrieve results
- Driver Manager, which loads drivers for the application
- Driver, which processes ODBC function calls, submits SQL requests to a specific data source, and returns results to the application
- Data source, which consists of the data to access and its associated operating system, DBMS, and network platform (if any) used to access the DBMS

The following figure shows the relationship among the four components:



Why Do Application Developers Need ODBC?

Using ODBC, an application developer can develop, compile, and ship an application without targeting a specific DBMS. In this scenario, the application developer does not need to use embedded SQL; therefore, he does not need to recompile the application for each new environment.

Environment-Specific Information

The following section refer to threading models. See Appendix C "Threading" on page 57 for an explanation of threading.

For UNIX and Linux Users

The following are requirements for UNIX and Linux operating systems. The DataDirect Connect *for* ODBC drivers are 32-bit drivers and the DataDirect Connect64 *for* ODBC drivers are 64-bit drivers.

32-Bit Drivers

- If your application was built with 32-bit system libraries, you must use a 32-bit driver. The database you are connecting to can be either 32-bit or 64-bit enabled.

AIX

- Power PC.
- AIX 5.2 and 5.3 operating systems with the 5.0.2.1 C++ runtime libraries.
- An application compiled with VisualAge C++ Professional 6.0 on AIX 5.2.
- An application built using the AIX native threading.

NOTE: To determine the installed version of your C++ runtime libraries, execute the following command:

```
lslpp -al | grep xlC.rte
```

HP-UX 11 aCC (PA-RISC)

- HP-UX 11i (11.11) operating system.
- An application compiled with HP aCC 3.30.
- An application built using the HP-UX 11 native (kernel) threading model (posix draft 10 threads).

HP-UX 11 aCC (IPF)

- HP-UX 11i (11.22 and 11.23) operating system.
- An application compiled with HP aCC 5.36.
- An application built using the HP-UX 11 native (kernel) threading model (posix draft 10 threads).

Solaris

- Sun Solaris 9 and 10 operating systems.
- Sun SPARCstation.
- An application compiled with Sun C++ 5.6 (Sun Studio 9) on Solaris 2.9.
- An application built using the Solaris native (kernel) threading model.

Linux

- The following Linux distributions are supported:
  - Red Hat Enterprise Linux 3.0
  - SuSE Linux Enterprise Server 8.0 and 9.0
- Intel x86 machine
- An application compiled with GNU project g++ 3.2.3 on RedHat Enterprise Linux 3.0.
- An application built using the Linux native pthread threading model (Linuxthreads)

64-Bit Drivers

- All required network software supplied by your database system vendors must be 64-bit compliant.

AIX

- Power PC.
- AIX 5L (5.2 and 5.3) operating system.
- An application compiled with VisualAge C++ Professional 6.0 on AIX 5.2.
- An application built using the AIX native threading model.

HP-UX 11 aCC (IPF)

- Itanium II.
- HP-UX IPF 11i versions 1.6 and 2 (B.11.22 and B.11.23) operating systems.
- An application compiled with HP aCC v. 5.36.
- An application built using the HP-UX 11 native (kernel) threading model (posix draft 10 threads).

Solaris

- Sun SPARCstation.
- Sun Solaris 9 and 10 operating systems.
- An application compiled with Sun C++ 5.6 (Sun Studio 9) on Solaris 2.9.
- An application built using the Solaris native (kernel) threading model.

Linux

- AMD Opteron and Intel Xeon EM64T x64 processors.
- The following operating systems are supported:
  - SuSE Linux Enterprise Server 9.0 for x64
  - Red Hat Enterprise Linux AS, ES, and WS version 4.0 for x64
- An application compiled with g++ GNU project C++ Compiler version 3.3.3.
- An application built using the Linux native pthread threading model (Linuxthreads).

Setup of the Environment and the Drivers

On UNIX and Linux, several environment variables and the system information file must be configured before the drivers can be used. See Chapter 1 "Quick Start Connect" on page 13 for a quick guide to this process. See Appendix G "The UNIX/Linux Environments" on page 93 for complete details about using the drivers on UNIX and Linux systems.

Driver Names

The DataDirect Connect Series *for* ODBC driver is an ODBC API-compliant dynamic link library, referred to in UNIX and Linux as *shared objects*. The prefix for the driver file name is iv for the 32-bit driver and dd for the 64-bit driver, the driver file name is lowercase, and the extension is .so or .sl. This is the standard form for a shared object. For example, the 32-bit NSK SQL/MX Wire Protocol driver file name is ivnsk21.so on all platforms except HP-UX, in which case it is ivnsk21.sl. The 64-bit driver file name is ddnsk21.so on all platforms.

Binding Parameter Markers

An ODBC application can prepare a query that contains dynamic parameters. Each parameter in a SQL statement must be associated, or bound, to a variable in the application before the statement is executed. When the application binds a variable to a parameter, it describes that variable and that parameter to the driver. Therefore, the application must supply the following information:

- The data type of the variable that the application maps to the dynamic parameter
- The SQL data type of the dynamic parameter (the data type that the database system assigned to the parameter marker)

The two data types are identified separately using the SQLBindParameter function. You can also use descriptor APIs as described in the Descriptor section of the ODBC specification (version 3.0 or higher).

The driver relies on the binding of parameters to know how to send information to the database system in its native format. If an application furnishes incorrect parameter binding information to the ODBC driver, the results will be unpredictable. For example, the statement might not be executed correctly.

To ensure interoperability, the DataDirect Connect Series *for* ODBC driver uses only the parameter binding information provided by the application. Some DBMSs cannot publish dynamic parameter information back to an ODBC driver.

Version String Information

The driver has a version string of the format:

X.YY.ZZZZ(B*AAAA*, U*BBBB*)

The Driver Manager on UNIX and Linux has a version string of the format:

X.YY.ZZZZ(U*BBBB*)

where:

X is the major version of the product.

YY is the minor version of the product.

ZZZZ is the build number of the driver component.

AAAA is the build number of the driver's bas component.

BBBB is the build number of the driver's utl component.

For example:

```
5.1.0002 (B0001, U0002
    |__|  |___|  |___|
   Driver  Bas    Utl
```

On UNIX and Linux, you can check the version string by using the ivtestlib tool shipped with the product. This tool is located in *install_directory*/bin.

Use the following command line:

```
ivtestlib shared_object
```

For example, for the NSK SQL/MX driver:

```
ivtestlib ivnsk21.so

5.1.0001 (B0002, U0001)
```

For example, for the Driver Manager:

```
ivtestlib libodbc.so

5.10.0001 (U0001)
```

NOTE: On Linux, the full path to the driver does not have to be specified for ivtestlib.

getFileVersionString Function

Version string information can also be obtained programmatically through the function getFileVersionString. This function can be used when the application is not directly calling ODBC functions.

This function is defined as follows and is located in each driver's shared object:

```
const unsigned char* getFileVersionString();
```

This function is prototyped in the qesqlext.h file shipped with the product.

Retrieving Data Type Information

At times, you might need to get information about the data types supported by the data source, for example, precision and scale. You can use the ODBC function SQLGetTypeInfo to do this.

On UNIX and Linux, an application can call SQLGetTypeInfo. Here is an example of a C function that calls SQLGetTypeInfo and retrieves the information in the form of a SQL result set.

```
void ODBC_GetTypeInfo(SQLHANDLE hstmt, SQLSMALLINT dataType)
{

    RETCODE rc;

// There are 19 columns returned by SQLGetTypeInfo.
// This example displays the first 3.
// Check the ODBC 3.x specification for more information.

// Variables to hold the data from each column
    char             typeName[30];
    short            sqlDataType;
    unsigned long    columnSize;

    SQLINTEGER       strlenTypeName,
                     strlenSqlDataType,
                     strlenColumnSize;

    rc = SQLGetTypeInfo(hstmt, dataType);
    if (rc == SQL_SUCCESS) {

// Bind the columns returned by the SQLGetTypeInfo result set.
        rc = SQLBindCol(hstmt, 1, SQL_C_CHAR, &typeName,
              (SDWORD)sizeof(typeName), &strlenTypeName);
        rc = SQLBindCol(hstmt, 2, SQL_C_SHORT, &sqlDataType,
              (SDWORD)sizeof(sqlDataType), &strlenSqlDataType);
        rc = SQLBindCol(hstmt, 3, SQL_C_LONG, &columnSize,
              (SDWORD)sizeof(columnSize), &strlenColumnSize);

// Print column headings
         printf ("TypeName      DataType          ColumnSize\n");
         printf ("------------------- ---------- ----------\n");

        do {
// Fetch the results from executing SQLGetTypeInfo
            rc = SQLFetch(hstmt);
            if (rc == SQL_ERROR) {
// Procedure to retrieve errors from the SQLGetTypeInfo function
               ODBC_GetDiagRec(SQL_HANDLE_STMT, hstmt);
               break;
              }
```

```
// Print the results
         if ((rc == SQL_SUCCESS) || (rc == SQL_SUCCESS_WITH_INFO)) {
printf ("%-30s %10i %10u\n", typeName, sqlDataType, columnSize);
             }

        } while (rc != SQL_NO_DATA);
    }
}
```

For information about how a database's data types map to the standard ODBC data types, see the driver chapter in this book.

3 The NSK SQL/MX Wire Protocol Driver

The DataDirect Connect Series *for* ODBC NSK SQL/MX Wire Protocol driver (the NSK SQL/MX Wire Protocol driver) supports:

HP NonStop SQL/MX database version 2.0

The NSK SQL/MX Wire Protocol driver is supported in the UNIX/linux environments. See "Environment-Specific Information" on page 19 for detailed information about the UNIX/linux environments supported by this driver.

Driver Requirements

There are no database client requirements for the NSK SQL/MX Wire Protocol driver.

Configuring Data Sources

After you have installed the driver, you will need to configure a data source or use a connection string to connect to the database. If you want to use a data source but need to change some of its values, you can either modify it or override its values through a connection string.

If you choose to use a connection string, you must use specific connection string attributes. See "Connecting to a Data Source Using a Connection String" on page 30 and Table 3-1 on page 32 for a complete description of driver connection string attributes and their values.

Refer to Chapter 1 "Quick Start Connect" on page 13 for a detailed explanation of different types of data sources.

On UNIX and Linux, data sources are configured and modified by editing the system information file (by default, odbc.ini) and storing default connection values there. See "Data Source Configuration" on page 97 for detailed information about editing the system information file. See Appendix G "The UNIX/Linux Environments" on page 93 for detailed information about the specific steps necessary to configure the UNIX and Linux environments and connect with the driver.

Table 3-1 on page 32 lists driver connection string attributes that must be used in the system information file. Note that only the long name of the attribute can be used in the file.

Connecting to a Data Source Using a Connection String

If you want to use a connection string for connecting to a database, or if your application requires it, you must specify either a DSN (data source name), a File DSN, or a DSN-less connection in the string. The difference is whether you use the DSN=, FILEDSN=, or the DRIVER= keyword in the connection string, as described in the ODBC specification. A DSN or FILEDSN connection string tells the driver where to find the default connection information. Optionally, you may specify *attribute=value* pairs in the connection string to override the default values stored in the data source.

The DSN connection string has the form:

DSN=*data_source_name*[;*attribute*=*value*[;*attribute*=*value*]...]

The FILEDSN connection string has the form:

FILEDSN=*filename.dsn*[;*attribute*=*value*[;*attribute*=*value*]...]

The DSN-less connection string specifies a driver instead of a data source. All connection information must be entered in the connection string because there is no data source storing the information.

The DSN-less connection string has the form:

DRIVER=[{]*driver_name*[}][;*attribute*=*value*[;*attribute*=*value*]...]

NOTE: Empty string is the default value for attributes that use a string value unless otherwise noted.

Table 3-1 gives the long and short names for each attribute, as well as a description. You can specify either long or short names in the connection string.

The defaults listed in the table are initial defaults that apply when no value is specified in either the data source definition or in the connection string. If you specified a value for the attribute when configuring the data source, that value is the default. Attributes are optional unless otherwise noted.

An example of a DSN connection string with overriding attribute values for NSK SQL/MX is:

```
DSN=NSK TABLES;NTIL=4
```

A FILEDSN connection string is similar except for the initial keyword:

```
FILEDSN=NSK.dsn;NTIL=4
```

A DSN-less connection string must provide all necessary connection information:

```
DRIVER=DataDirect 5.1 NSK SQL/MX Wire Protocol;
HST=NSK2;PRT=111;CATALOG=MYCATALOG;SCHEMA=MYSCHEMA;
UID=JOHN;PWD=XYZZY
```

Table 3-1. NSK SQL/MX Wire Protocol Connection String Attributes

Attribute	Description
AuthStr (PWD) (Required)	A case-sensitive password.
DataSourceName (DSN)	A string that identifies an NSK SQL/MX data source configuration.
HostName (HST) (Required)	Either the IP address or the host name of the NonStop MXCS Association server.
IANAAppCodePage (IACP)	See Appendix F "Values for IANAAppCodePage Connection String Attribute" on page 87 for a list of valid values for this attribute. You need to set this attribute if your application is not Unicode-enabled and/or if your database character set is not Unicode. The value you specify must match the database character encoding and the system locale. This attribute applies to UNIX and Linux only. The Driver Manager checks for the value of IANAAppCodePage in the following order: ■ In the connection string ■ In the Data Source section of the system information file (odbc.ini) ■ In the ODBC section of the system information file (odbc.ini) If no IANAAppCodePage value is found, the driver uses the default value of 4 (ISO 8859-1 Latin-1).
NskCatalog (CATALOG) (Required)	The catalog used to qualify NonStop SQL/MX object names.
NskDatasource (NDS)	NskDatasource="TDM_Default_DataSource". The name of the NSK server-side data source.
NskSchema (SCHEMA) (Required)	The schema used to qualify NonStop SQL/MX object names.

Table 3-1. NSK SQL/MX Wire Protocol Connection String Attributes (cont.)

Attribute	Description
NskTransactionIsolation Level (NTIL)	NskTransactionIsolationLevel={1 | 2 | 4 | 8}. Specifies the default isolation level for concurrent transactions, where: 1 = READ UNCOMMITED 2 = READ COMMITTED 4 = REPEATABLE READ 8 = SERIALIZABLE The initial default is 2.
NskWindowText (NWT)	Character string of text as displayed in NSM/web. The initial default is ivnsk.
PortNumber (PRT) (Required)	The port number of the NonStop MXCS Association server.
UserID (UID) (Required)	The logon ID used to connect to your NSK database. This ID is case-sensitive.

Data Types

Table 3-2 shows how the NSK SQL/MX data types map to the standard ODBC data types.

Table 3-2. NSK SQL/MX Data Types

NSK SQL/MX	ODBC
Bigint	SQL_BIGINT
Bigint signed	SQL_BIGINT
Char	SQL_CHAR
Date	SQL_TYPE_DATE
Decimal	SQL_DECIMAL
Decimal signed	SQL_DECIMAL

Table 3-2. NSK SQL/MX Data Types (cont.)

NSK SQL/MX	ODBC
Decimal unsigned	SQL_DECIMAL
Double precision	SQL_DOUBLE
Float	SQL_FLOAT
Integer	SQL_INTEGER
Integer signed	SQL_INTEGER
Integer unsigned	SQL_INTEGER
Interval (p) day to day	SQL_INTERVAL_DAY
Interval (p) day to hour	SQL_INTERVAL_DAY_TO_HOUR
Interval (p) day to minute	SQL_INTERVAL_DAY_TO_MINUTE
Interval (p) day to second	SQL_INTERVAL_DAY_TO_SECOND
Interval (p) hour to hour	SQL_INTERVAL_HOUR
Interval (p) hour to minute	SQL_INTERVAL_HOUR_TO_MINUTE
Interval (p) hour to second	SQL_INTERVAL_HOUR_TO_SECOND
Interval (p) minute to minute	SQL_INTERVAL_MINUTE
Interval (p) minute to second	SQL_INTERVAL_MINUTE_TO_SECOND
Interval (p) month to month	SQL_INTERVAL_MONTH
Interval (p) second to second	SQL_INTERVAL_SECOND
Interval (p) year to month	SQL_INTERVAL_YEAR_TO_MONTH
Interval (p) year to year	SQL_INTERVAL_YEAR
Longvarchar	SQL_LONGVARCHAR
Numeric	SQL_NUMERIC
Numeric signed	SQL_NUMERIC
Numeric unsigned	SQL_NUMERIC
Real	SQL_REAL
Smallint	SQL_SMALLINT
Smallint signed	SQL_SMALLINT
Smallint unsigned	SQL_SMALLINT
Time	SQL_TYPE_TIME

Table 3-2. NSK SQL/MX Data Types (cont.)

NSK SQL/MX	ODBC
Timestamp	SQL_TYPE_TIMESTAMP
Varchar	SQL_VARCHAR

See "Retrieving Data Type Information" on page 26 for more information about data types.

Persisting a Result Set as an XML Data File

This driver allows you to persist a result set as an XML data file with embedded schema. To implement XML persistence, a client application must do the following:

1 Turn on STATIC cursors. For example:

```
SQLSetStmtAttr (hstmt, SQL_ATTR_CURSOR_TYPE,
SQL_CURSOR_STATIC, SQL_IS_INTEGER)
```

NOTE: A result set can be persisted as an XML data file only if the result set is generated using STATIC cursors. Otherwise, the following error is returned:

```
Driver only supports XML persistence when using
driver's static cursors.
```

2 Execute a SQL statement. For example:

```
SQLExecDirect (hstmt, "SELECT * FROM GTABLE", SQL_NTS)
```

3 Persist the result set as an XML data file. For example:

```
SQLSetStmtAttr (hstmt, SQL_PERSIST_AS_XML,
"C:\temp\GTABLE.XML", SQL_NTS)
```

NOTE: A new statement attribute is available to support XML persistence, SQL_PERSIST_AS_XML. A client application must call SQLSetStmtAttr with this new attribute as an argument. See the following table for the definition of valid arguments for SQLSetStmtAttr.

Argument	Definition
StatementHandle	The handle of the statement that contains the result set to persist as XML.
Attribute	SQL_PERSIST_AS_XML. This new statement attribute can be found in the file qesqlext.h, which is installed with the driver.
ValuePtr	Pointer to a URL that specifies the full path name of the XML data file to be generated. The directory specified in the path name must exist, and if the specified file name exists, the file will be overwritten.
StringLength	The length of the string pointed to by ValuePtr or SQL_NTS if ValuePtr points to a null terminated string.

A client application can choose to persist the data at any time that the statement is in an executed or cursor-positioned state. At any other time, the driver returns the following message:

```
Function Sequence Error
```

Using the XML Persistence Demo Tool

On UNIX/linux, the product is shipped with an XML persistence demo tool named demoodbc. This tool is installed in the demo subdirectory of the installation directory. For information about how to use this tool, refer to the demoodbc.txt file installed in the demo directory.

Isolation and Lock Levels Supported

NSK SQL/MX supports isolation levels read uncommitted, read committed, and serializable. The default is read committed.

NSK SQL/MX supports record-level locking.

See Appendix B "Locking and Isolation Levels" on page 51 for details.

ODBC Conformance Level

See Appendix A "ODBC API and Scalar Functions" on page 39 for a list of the API functions supported by the NSK SQL/MX Wire Protocol driver.

The NSK SQL/MX Wire Protocol driver also supports the following functions:

- SQLColumnPrivileges
- SQLForeignKeys
- SQLTablePrivileges

The driver supports the minimum SQL grammar.

Number of Connections and Statements Supported

The NSK SQL/MX Wire Protocol driver supports multiple connections and one statement per connection to the NonStop SQL/MX database system.

A ODBC API and Scalar Functions

This appendix lists the ODBC API functions that the DataDirect Connect Series *for* ODBC drivers support and the scalar functions, which you use in SQL statements. This appendix includes the following:

- "API Functions" on page 39
- "Scalar Functions" on page 42

API Functions

The DataDirect Connect Series *for* ODBC drivers support all ODBC Core and Level 1 functions—they are ODBC Level 1–compliant. They also support a limited set of Level 2 functions. The drivers support the functions listed in Table A-1 on page 40 and Table A-2 on page 41. Any additions to these supported functions or differences in the support of specific functions are listed in the "ODBC Conformance Level" section in the individual driver chapters.

Table A-1. Function Conformance for 2.x ODBC Applications

Core Functions	Level 1 Functions
SQLAllocConnect	SQLColumns
SQLAllocEnv	SQLDriverConnect
SQLAllocStmt	SQLGetConnectOption
SQLBindCol	SQLGetData
SQLBindParameter	SQLGetFunctions
SQLCancel	SQLGetInfo
SQLColAttributes	SQLGetStmtOption
SQLConnect	SQLGetTypeInfo
SQLDescribeCol	SQLParamData
SQLDisconnect	SQLPutData
SQLDrivers	SQLSetConnectOption
SQLError	SQLSetStmtOption
SQLExecDirect	SQLSpecialColumns
SQLExecute	SQLStatistics
SQLFetch	SQLTables
SQLFreeConnect	**Level 2 Functions**
SQLFreeEnv	SQLBrowseConnect
SQLFreeStmt	SQLDataSources
SQLGetCursorName	SQLExtendedFetch (forward scrolling only)
SQLNumResultCols	SQLMoreResults
SQLPrepare	SQLNativeSql
SQLRowCount	SQLNumParams
SQLSetCursorName	SQLParamOptions
SQLTransact	SQLSetScrollOptions

Table A-2. Function Conformance for 3.x ODBC Applications

SQLAllocHandle	SQLGetData
SQLBindCol	SQLGetDescField
SQLBindParameter	SQLGetDescRec
SQLBrowseConnect	SQLGetDiagField
SQLBulkOperations	SQLGetDiagRec
SQLCancel	SQLGetEnvAttr
SQLCloseCursor	SQLGetFunctions
SQLColAttribute	SQLGetInfo
SQLColumns	SQLGetStmtAttr
SQLConnect	SQLGetTypeInfo
SQLCopyDesc	SQLMoreResults
SQLDataSources	SQLNativeSql
SQLDescribeCol	SQLNumParens
SQLDisconnect	SQLNumResultCols
SQLDriverConnect	SQLParamData
SQLDrivers	SQLPrepare
SQLEndTran	SQLPutData
SQLError	SQLRowCount
SQLExecDirect	SQLSetConnectAttr
SQLExecute	SQLSetCursorName
SQLExtendedFetch	SQLSetDescField
SQLFetch	SQLSetDescRec
SQLFetchScroll (forward scrolling only)	SQLSetEnvAttr
SQLFreeHandle	SQLSetStmtAttr
SQLFreeStmt	SQLSpecialColumns
SQLGetConnectAttr	SQLStatistics
SQLGetCursorName	SQLTables
	SQLTransact

Scalar Functions

This section lists the scalar functions that ODBC supports. Your database system may not support all of these functions. See the documentation for your database system to find out which functions are supported. Also, depending on the driver that you are using, all of the scalar functions may not be supported. To check which scalar functions are supported by a driver, use the SQLGetInfo ODBC function.

You can use these functions in SQL statements using the following syntax:

```
{fn scalar-function}
```

where *scalar-function* is one of the functions listed in the following tables. For example:

```
SELECT {fn UCASE(NAME)} FROM EMP
```

String Functions

Table A-3 on page 43 lists the string functions that ODBC supports.

The string functions listed can take the following arguments:

- *string_exp* can be the name of a column, a string literal, or the result of another scalar function, where the underlying data type is SQL_CHAR, SQL_VARCHAR, or SQL_LONGVARCHAR.
- *start*, *length*, and *count* can be the result of another scalar function or a literal numeric value, where the underlying data type is SQL_TINYINT, SQL_SMALLINT, or SQL_INTEGER.

The string functions are one-based; that is, the first character in the string is character 1.

Character string literals must be surrounded in single quotation marks.

Table A-3. Scalar String Functions

Function	Returns
ASCII(*string_exp*)	ASCII code value of the leftmost character of *string_exp* as an integer.
BIT_LENGTH(*string_exp*) *[ODBC 3.0 only]*	The length in bits of the string expression.
CHAR(*code*)	The character with the ASCII code value specified by *code*. *code* should be between 0 and 255; otherwise, the return value is data-source dependent.
CHAR_LENGTH(*string_exp*) *[ODBC 3.0 only]*	The length in characters of the string expression, if the string expression is of a character data type; otherwise, the length in bytes of the string expression (the smallest integer not less than the number of bits divided by 8). (This function is the same as the CHARACTER_LENGTH function.)
CHARACTER_ LENGTH(string_exp) *[ODBC 3.0 only]*	The length in characters of the string expression, if the string expression is of a character data type; otherwise, the length in bytes of the string expression (the smallest integer not less than the number of bits divided by 8). (This function is the same as the CHAR_LENGTH function.)
CONCAT(*string_exp1*, *string_exp2*)	The string resulting from concatenating *string_exp2* and *string_exp1*. The string is system dependent.
DIFFERENCE(*string_exp1*, *string_exp2*)	An integer value that indicates the difference between the values returned by the SOUNDEX function for *string_exp1* and *string_exp2*.
INSERT(*string_exp1*, *start*, *length*, *string_exp2*)	A string where *length* characters have been deleted from *string_exp1* beginning at *start* and where *string_exp2* has been inserted into *string_exp*, beginning at *start*.
LCASE(*string_exp*)	Uppercase characters in *string_exp* converted to lowercase.
LEFT(*string_exp*,*count*)	The *count* of characters of *string_exp*.
LENGTH(*string_exp*)	The number of characters in *string_exp*, excluding trailing blanks and the string termination character.

Table A-3. Scalar String Functions (cont.)

Function	Returns
LOCATE(*string_exp1*, *string_exp2*[,*start*])	The starting position of the first occurrence of *string_exp1* within *string_exp2*. If *start* is not specified, the search begins with the first character position in *string_exp2*. If *start* is specified, the search begins with the character position indicated by the value of *start*. The first character position in *string_exp2* is indicated by the value 1. If *string_exp1* is not found, 0 is returned.
LTRIM(*string_exp*)	The characters of *string_exp*, with leading blanks removed.
OCTET_LENGTH(*string_exp*) *[ODBC 3.0 only]*	The length in bytes of the string expression. The result is the smallest integer not less than the number of bits divided by 8.
POSITION(*character_exp* IN *character_exp*) *[ODBC 3.0 only]*	The position of the first character expression in the second character expression. The result is an exact numeric with an implementation-defined precision and a scale of 0.
REPEAT(*string_exp*, *count*)	A string composed of *string_exp* repeated *count* times.
REPLACE(*string_exp1*, *string_exp2*, *string_exp3*)	Replaces all occurrences of *string_exp2* in *string_exp1* with *string_exp3.*
RIGHT(*string_exp*, *count*)	The rightmost *count* of characters in *string_exp*.
RTRIM(*string_exp*)	The characters of *string_exp* with trailing blanks removed.
SOUNDEX(*string_exp*)	A data-source-dependent string representing the sound of the words in *string_exp*.
SPACE(*count*)	A string consisting of *count* spaces.
SUBSTRING(*string_exp*, *start*, *length*)	A string derived from *string_exp* beginning at the character position *start* for *length* characters.
UCASE(*string_exp*)	Lowercase characters in *string_exp* converted to uppercase.

Numeric Functions

Table A-4 lists the numeric functions that ODBC supports.

The numeric functions listed can take the following arguments:

- *numeric_exp* can be a column name, a numeric literal, or the result of another scalar function, where the underlying data type is SQL_NUMERIC, SQL_DECIMAL, SQL_TINYINT, SQL_SMALLINT, SQL_INTEGER, SQL_BIGINT, SQL_FLOAT, SQL_REAL, or SQL_DOUBLE.
- *float_exp* can be a column name, a numeric literal, or the result of another scalar function, where the underlying data type is SQL_FLOAT.
- *integer_exp* can be a column name, a numeric literal, or the result of another scalar function, where the underlying data type is SQL_TINYINT, SQL_SMALLINT, SQL_INTEGER, or SQL_BIGINT.

Table A-4. Scalar Numeric Functions

Function	Returns
ABS(*numeric_exp*)	Absolute value of *numeric_exp.*
ACOS(*float_exp*)	Arccosine of *float_exp* as an angle in radians.
ASIN(*float_exp*)	Arcsine of *float_exp* as an angle in radians.
ATAN(*float_exp*)	Arctangent of *float_exp* as an angle in radians.
ATAN2(*float_exp1*, *float_exp2*)	Arctangent of the x and y coordinates, specified by *float_exp1* and *float_exp2* as an angle in radians.
CEILING(*numeric_exp*)	Smallest integer greater than or equal to *numeric_exp.*
COS(*float_exp*)	Cosine of *float_exp* as an angle in radians.
COT(*float_exp*)	Cotangent of *float_exp* as an angle in radians.
DEGREES(*numeric_exp*)	Number if degrees converted from *numeric_exp* radians.

Table A-4. Scalar Numeric Functions *(cont.)*

Function	Returns
EXP(*float_exp*)	Exponential value of *float_exp.*
FLOOR(*numeric_exp*)	Largest integer less than or equal to *numeric_exp*.
LOG(*float_exp*)	Natural log of *float_exp.*
LOG10(*float_exp*)	Base 10 log of *float_exp.*
MOD(*integer_exp1*, *integer_exp2*)	Remainder of *integer_exp1* divided by *integer_exp2*.
PI()	Constant value of pi as a floating-point number.
POWER(*numeric_exp*, *integer_exp*)	Value of *numeric_exp* to the power of *integer_exp*.
RADIANS(*numeric_exp*)	Number of radians converted from *numeric_exp* degrees.
RAND([*integer_exp*])	Random floating-point value using *integer_exp* as the optional seed value.
ROUND(*numeric_exp*, *integer_exp*)	*numeric_exp* rounded to *integer_exp* places right of the decimal (left of the decimal if *integer_exp* is negative).
SIGN(*numeric_exp*)	Indicator of the sign of *numeric_exp*. If *numeric_exp* < 0, -1 is returned. If *numeric_exp* = 0, 0 is returned. If *numeric_exp* > 0, 1 is returned.
SIN(*float_exp*)	Sine of *float_exp*, where *float_exp* is an angle in radians.
SQRT(*float_exp*)	Square root of *float_exp*.
TAN(*float_exp*)	Tangent of *float_exp*, where *float_exp* is an angle in radians.
TRUNCATE(*numeric_exp*, *integer_exp*)	*numeric_exp* truncated to *integer_exp* places right of the decimal. (If *integer_exp* is negative, truncation is to the left of the decimal.)

Date and Time Functions

Table A-5 lists the date and time functions that ODBC supports.

The date and time functions listed can take the following arguments:

- *date_exp* can be a column name, a date or timestamp literal, or the result of another scalar function, where the underlying data type can be represented as SQL_CHAR, SQL_VARCHAR, SQL_DATE, or SQL_TIMESTAMP.
- *time_exp* can be a column name, a timestamp or timestamp literal, or the result of another scalar function, where the underlying data type can be represented as SQL_CHAR, SQL_VARCHAR, SQL_TIME, or SQL_TIMESTAMP.
- *timestamp_exp* can be a column name; a time, date, or timestamp literal; or the result of another scalar function, where the underlying data type can be represented as SQL_CHAR, SQL_VARCHAR, SQL_TIME, SQL_DATE, or SQL_TIMESTAMP.

Table A-5. Scalar Time and Date Functions

Function	Returns
CURRENT_DATE() *[ODBC 3.0 only]*	Current date.
CURRENT_TIME[(*time-precision*)] *[ODBC 3.0 only]*	Current local time. The *time-precision* argument determines the seconds precision of the returned value.
CURRENT_TIMESTAMP[(*timestamp-precision*)] *[ODBC 3.0 only]*	Current local date and local time as a timestamp value. The *timestamp-precision* argument determines the seconds precision of the returned timestamp.
CURDATE()	Current date as a date value.
CURTIME()	Current local time as a time value.

Table A-5. Scalar Time and Date Functions *(cont.)*

Function	Returns
DAYNAME(*date_exp*)	Character string containing a data-source-specific name of the day for the day portion of *date_exp*.
DAYOFMONTH(*date_exp*)	Day of the month in *date_exp* as an integer value (1–31).
DAYOFWEEK(*date_exp*)	Day of the week in *date_exp* as an integer value (1–7).
DAYOFYEAR(*date_exp*)	Day of the year in *date_exp* as an integer value (1–366).
HOUR(*time_exp*)	Hour in *time_exp* as an integer value (0–23).
MINUTE(*time_exp*)	Minute in *time_exp* as an integer value (0–59).
MONTH(*date_exp*)	Month in *date_exp* as an integer value (1–12).
MONTHNAME(*date_exp*)	Character string containing the data source-specific name of the month.
NOW()	Current date and time as a timestamp value.
QUARTER(*date_exp*)	Quarter in *date_exp* as an integer value (1–4).
SECOND(*time_exp*)	Second in *date_exp* as an integer value (0–59).
TIMESTAMPADD(*interval*, *integer_exp*, *time_exp*)	Timestamp calculated by adding *integer_exp* intervals of type interval to *time_exp*. *interval* can be: SQL_TSI_FRAC_SECOND SQL_TSI_SECOND SQL_TSI_MINUTE SQL_TSI_HOUR SQL_TSI_DAY SQL_TSI_WEEK SQL_TSI_MONTH SQL_TSI_QUARTER SQL_TSI_YEAR Fractional seconds are expressed in billionths of a second.

Table A-5. Scalar Time and Date Functions (cont.)

Function	Returns
TIMESTAMPDIFF(*interval*, *time_exp1*, *time_exp2*)	Integer number of intervals of type *interval* by which *time_exp2* is greater than *time_exp1*. *interval* has the same values as TIMESTAMPADD. Fractional seconds are expressed in billionths of a second.
WEEK(*date_exp*)	Week of the year in *date_exp* as an integer value (1–53).
YEAR(*date_exp*)	Year in *date_exp*. The range is data-source dependent.

System Functions

Table A-6 lists the system functions that ODBC supports.

Table A-6. Scalar System Functions

Function	Returns
DATABASE()	Name of the database, corresponding to the connection handle (*hdbc*).
IFNULL(*exp,value*)	*value*, if *exp* is null.
USER()	Authorization name of the user.

B Locking and Isolation Levels

This appendix discusses locking and isolation levels and how their settings can affect the data you retrieve. Different database systems support different locking and isolation levels. See the section "Isolation and Lock Levels Supported" in the driver chapter. This appendix includes the following:

- "Locking" on page 51
- "Isolation Levels" on page 52
- "Locking Modes and Levels" on page 55

Locking

Locking is a database operation that restricts a user from accessing a table or record. Locking is used in situations where more than one user might try to use the same table or record at the same time. By locking the table or record, the system ensures that only one user at a time can affect the data.

Locking is critical in multiuser databases, where different users can try to access or modify the same records concurrently. Although such concurrent database activity is desirable, it can create problems. Without locking, for example, if two users try to modify the same record at the same time, they might encounter problems ranging from retrieving bad data to deleting data that the other user needs. If, however, the first user to access a record can lock that record to temporarily prevent other users from modifying it, such problems can be avoided. Locking provides a way to manage concurrent database access while minimizing the various problems it can cause.

Isolation Levels

An isolation level represents a particular locking strategy employed in the database system to improve data consistency. The higher the isolation level, the more complex the locking strategy behind it. The isolation level provided by the database determines whether a transaction will encounter the following behaviors in data consistency:

Dirty reads	User 1 modifies a row. User 2 reads the same row before User 1 commits. User 1 performs a rollback. User 2 has read a row that has never really existed in the database. User 2 may base decisions on false data.
Non-repeatable reads	User 1 reads a row but does not commit. User 2 modifies or deletes the same row and then commits. User 1 rereads the row and finds it has changed (or has been deleted).
Phantom reads	User 1 uses a search condition to read a set of rows but does not commit. User 2 inserts one or more rows that satisfy this search condition, then commits. User 1 rereads the rows using the search condition and discovers rows that were not present before.

Isolation levels represent the database system's ability to prevent these behaviors. The American National Standards Institute (ANSI) defines four isolation levels:

- Read uncommitted (0)
- Read committed (1)
- Repeatable read (2)
- Serializable (3)

In ascending order (0–3), these isolation levels provide an increasing amount of data consistency to the transaction. At the lowest level, all three behaviors can occur. At the highest level, none can occur. The success of each level in preventing these behaviors is due to the locking strategies that they employ, which are as follows:

Read uncommitted (0)	Locks are obtained on modifications to the database and held until end of transaction (EOT). Reading from the database does not involve any locking.
Read committed (1)	Locks are acquired for reading and modifying the database. Locks are released after reading but locks on modified objects are held until EOT.
Repeatable read (2)	Locks are obtained for reading and modifying the database. Locks on all modified objects are held until EOT. Locks obtained for reading data are held until EOT. Locks on non-modified access structures (such as indexes and hashing structures) are released after reading.
Serializable (3)	All data read or modified is locked until EOT. All access structures that are modified are locked until EOT. Access structures used by the query are locked until EOT.

Table B-1 shows what data consistency behaviors can occur at each isolation level.

Table B-1. Isolation Levels and Data Consistency

Level	Dirty Read	Nonrepeatable Read	Phantom Read
0, Read uncommitted	Yes	Yes	Yes
1, Read committed	No	Yes	Yes
2, Repeatable read	No	No	Yes
3, Serializable	No	No	No

Although higher isolation levels provide better data consistency, this consistency can be costly in terms of the concurrency provided to individual users. Concurrency is the ability of multiple users to access and modify data simultaneously. As isolation levels increase, so does the chance that the locking strategy used will create problems in concurrency.

Put another way: The higher the isolation level, the more locking involved, and the more time users may spend waiting for data to be freed by another user. Because of this inverse relationship between isolation levels and concurrency, you must consider how people use the database before choosing an isolation level. You must weigh the trade-offs between data consistency and concurrency, and decide which is more important.

Locking Modes and Levels

Different database systems employ various locking modes, but they have two basic ones in common: shared and exclusive. Shared locks can be held on a single object by multiple users. If one user has a shared lock on a record, then a second user can also get a shared lock on that same record; however, the second user cannot get an exclusive lock on that record. Exclusive locks are exclusive to the user that obtains them. If one user has an exclusive lock on a record, then a second user cannot get either type of lock on the same record.

Performance and concurrency can also be affected by the locking level used in the database system. The locking level determines the size of an object that is locked in a database. For example, many database systems let you lock an entire table, as well as individual records. An intermediate level of locking, page-level locking, is also common. A page contains one or more records and is typically the amount of data read from the disk in a single disk access. The major disadvantage of page-level locking is that if one user locks a record, a second user may not be able to lock other records because they are stored on the same page as the locked record.

C Threading

The ODBC specification mandates that all drivers must be thread-safe; that is, drivers must not fail when database requests are made on separate threads. It is a common misperception that issuing requests on separate threads will always result in improved throughput. Because of network transport and database server limitations, some drivers may serialize threaded requests to the server to ensure thread safety.

The ODBC 3.0 specification does not provide a method to find out how a driver will service threaded requests although this information is quite useful to an application. The DataDirect Connect Series *for* ODBC drivers provide this information to the user via the SQLGetInfo information type 1028.

The result of calling SQLGetInfo with 1028 is a SQL_USMALLINT flag which denotes the session's thread model. A return value of 0 denotes that the session is fully thread enabled and that all requests will fully utilize the threaded model. A return value of 1 denotes that the session is restricted at the connection level. Sessions of this type are fully thread-enabled when simultaneous threaded requests are made with statement handles that do not share the same connection handle. In this model, if multiple requests are made from the same connection, then the first request received by the driver is processed immediately and all subsequent requests are serialized. A return value of 2 denotes that the session is thread-impaired and all requests are serialized by the driver.

Consider the following code fragment:

```
rc = SQLGetInfo (hdbc, 1028, &ThreadModel, NULL, NULL);

If (rc == SQL_SUCCESS) {
   // driver is a DataDirect driver which can report
   // threading information

   if (ThreadModel == 0)
   // driver is unconditionally thread enabled
   // application can take advantage of threading

   else if (ThreadModel == 1)
   // driver is thread enabled when thread requests are
   // from different connections
   // some applications can take advantage of threading

   else if (ThreadModel == 2)
   // driver is thread impaired
   // application should only use threads if it reduces
   // program complexity

}
else
   // driver is only guaranteed to be thread-safe
   // use threading at your own risk
```

Driver Threading Information

The NSK SQL/MX Wire Protocol driver is fully threaded.

D Using Indexes

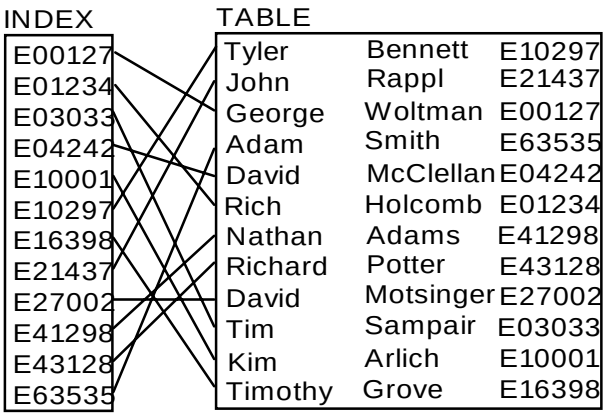
This appendix discusses the ways in which you can improve the performance of database activity using indexes. It provides general guidelines that apply to most databases. Consult your database vendor's documentation for more detailed information.

For information regarding how to create and drop indexes, see your database system documentation.

Introduction

An index is a database structure that you can use to improve the performance of database activity. A database table can have one or more indexes associated with it.

An index is defined by a field expression that you specify when you create the index. Typically, the field expression is a single field name, like EMP_ID. An index created on the EMP_ID field, for example, contains a sorted list of the employee ID values in the table. Each value in the list is accompanied by references to the records that contain that value.



A database driver can use indexes to find records quickly. An index on the EMP_ID field, for example, greatly reduces the time that the driver spends searching for a particular employee ID value. Consider the following Where clause:

```
WHERE emp_id = 'E10001'
```

Without an index, the driver must search the entire database table to find those records having an employee ID of E10001. By using an index on the EMP_ID field, however, the driver can quickly find those records.

Indexes may improve the performance of SQL statements. You may not notice this improvement with small tables but it can be significant for large tables; however, there can be disadvantages to having too many indexes. Indexes can slow down the performance of some inserts, updates, and deletes when the driver has to maintain the indexes as well as the database tables. Also, indexes take additional disk space.

Improving Record Selection Performance

For indexes to improve the performance of selections, the index expression must match the selection condition exactly. For example, if you have created an index whose expression is last_name, the following Select statement uses the index:

```
SELECT * FROM emp WHERE last_name = 'Smith'
```

This Select statement, however, does not use the index:

```
SELECT * FROM emp WHERE UPPER(last_name) = 'SMITH'
```

The second statement does not use the index because the Where clause contains UPPER(LAST_NAME), which does not match the index expression LAST_NAME. If you plan to use the UPPER function in all your Select statements and your database supports indexes on expressions, then you should define an index using the expression UPPER(LAST_NAME).

Indexing Multiple Fields

If you often use Where clauses that involve more than one field, you may want to build an index containing multiple fields. Consider the following Where clause:

```
WHERE last_name = 'Smith' and first_name = 'Thomas'
```

For this condition, the optimal index field expression is LAST_NAME, FIRST_NAME. This creates a concatenated index.

Concatenated indexes can also be used for Where clauses that contain only the first of two concatenated fields. The LAST_NAME, FIRST_NAME index also improves the performance of the following Where clause (even though no first name value is specified):

```
last_name = 'Smith'
```

Consider the following Where clause:

```
WHERE last_name = 'Smith' and middle_name = 'Edward' and
first_name = 'Thomas'
```

If your index fields include all the conditions of the Where clause in that order, the driver can use the entire index. If, however, your index is on two nonconsecutive fields, say, LAST_NAME and FIRST_NAME, the driver can use only the LAST_NAME field of the index.

The driver uses only one index when processing Where clauses. If you have complex Where clauses that involve a number of conditions for different fields and have indexes on more than one field, the driver chooses an index to use. The driver attempts to use indexes on conditions that use the equal sign as the relational operator rather than conditions using other operators (such as greater than). Assume you have an index on the EMP_ID field as well as the LAST_NAME field and the following Where clause:

```
WHERE emp_id >= 'E10001' AND last_name = 'Smith'
```

In this case, the driver selects the index on the LAST_NAME field.

If no conditions have the equal sign, the driver first attempts to use an index on a condition that has a lower *and* upper bound, and then attempts to use an index on a condition that has a lower *or* upper bound. The driver always attempts to use the most restrictive index that satisfies the Where clause.

In most cases, the driver does not use an index if the Where clause contains an OR comparison operator. For example, the driver does not use an index for the following Where clause:

```
WHERE emp_id >= 'E10001' OR last_name = 'Smith'
```

Deciding Which Indexes to Create

Before you create indexes for a database table, consider how you will use the table. The two most common operations on a table are to:

- Insert, update, and delete records
- Retrieve records

If you most often insert, update, and delete records, then the fewer indexes associated with the table, the better the performance. This is because the driver must maintain the indexes as well as the database tables, thus slowing down the performance of record inserts, updates, and deletes. It may be more efficient to drop all indexes before modifying a large number of records, and re-create the indexes after the modifications.

If you most often retrieve records, you must look further to define the criteria for retrieving records and create indexes to improve the performance of these retrievals. Assume you have an employee database table and you will retrieve records based on employee name, department, or hire date. You would create three indexes—one on the DEPT field, one on the HIRE_DATE field, and one on the LAST_NAME field. Or perhaps, for the retrievals based on the name field, you would want an index that concatenates the LAST_NAME and the FIRST_NAME fields (see "Indexing Multiple Fields" on page 61 for details).

Here are a few rules to help you decide which indexes to create:

- If your record retrievals are based on one field at a time (for example, dept='D101'), create an index on these fields.
- If your record retrievals are based on a combination of fields, look at the combinations.
- If the comparison operator for the conditions is AND (for example, CITY = 'Raleigh' AND STATE = 'NC'), then build a concatenated index on the CITY and STATE fields. This index is also useful for retrieving records based on the CITY field.
- If the comparison operator is OR (for example, DEPT = 'D101' OR HIRE_DATE > {01/30/89}), an index does not help performance. Therefore, you need not create one.
- If the retrieval conditions contain both AND and OR comparison operators, you can use an index if the OR conditions are grouped. For example:

  ```
  dept = 'D101' AND (hire_date > {01/30/89} OR
  exempt = 1)
  ```

  In this case, an index on the DEPT field improves performance.
- If the AND conditions are grouped, an index does not improve performance. For example:

  ```
  (dept = 'D101' AND hire_date) > {01/30/89}) OR
  exempt = 1
  ```

Improving Join Performance

When joining database tables, index tables can greatly improve performance. Unless the proper indexes are available, queries that use joins can take a long time.

Assume you have the following Select statement:

```
SELECT * FROM dept, emp WHERE dept.dept_id = emp.dept
```

In this example, the DEPT and EMP database tables are being joined using the department ID field. When the driver executes a query that contains a join, it processes the tables from left to right and uses an index on the second table's join field (the DEPT field of the EMP table).

To improve join performance, you need an index on the join field of the second table in the From clause. If there is a third table in the From clause, the driver also uses an index on the field in the third table that joins it to any previous table. For example:

```
SELECT * FROM dept, emp, addr
WHERE dept.dept_id = emp.dept AND emp.loc = addr.loc
```

In this case, you should have an index on the EMP.DEPT field and the ADDR.LOC field.

E Performance Design of ODBC Applications

Developing performance-oriented ODBC applications is not easy. Microsoft's *ODBC Programmer's Reference* does not provide information about system performance. In addition, ODBC drivers and the ODBC driver manager do not return warnings when applications run inefficiently. This appendix contains some general guidelines that have been compiled by examining the ODBC implementations of numerous shipping ODBC applications. These guidelines include:

- Use catalog functions appropriately
- Retrieve only required data
- Select functions that optimize performance
- Manage connections and updates

Following these general rules will help you solve some common ODBC performance problems, such as those listed in Table E-1.

Table E-1. Common Performance Problems Using ODBC Applications

Problem	Solution	See guidelines in...
Network communication is slow.	Reduce network traffic.	"Using Catalog Functions" on page 68
The process of evaluating complex SQL queries on the database server is slow and can reduce concurrency.	Simplify queries.	"Using Catalog Functions" on page 68 "Selecting ODBC Functions" on page 77

Table E-1. Common Performance Problems Using ODBC Applications (cont.)

Problem	Solution	See guidelines in...
Excessive calls from the application to the driver slow performance.	Optimize application-to-driver interaction.	"Retrieving Data" on page 72 "Selecting ODBC Functions" on page 77
Disk input/output is slow.	Limit disk input/output.	"Managing Connections and Updates" on page 81

Using Catalog Functions

Because catalog functions, such as those listed here, are slow compared to other ODBC functions, their frequent use can impair system performance:

- SQLColumns
- SQLColumnPrivileges
- SQLForeignKeys
- SQLGetTypeInfo
- SQLProcedures
- SQLProcedureColumns
- SQLSpecialColumns
- SQLStatistics
- SQLTables

SQLGetTypeInfo is included in this list of expensive ODBC functions because many drivers must query the server to obtain accurate information about which types are supported (for example, to find dynamic types such as user defined types).

Minimizing the Use of Catalog Functions

Compared to other ODBC functions, catalog functions are relatively slow. By caching information, applications can avoid multiple executions. Although it is almost impossible to write an ODBC application without catalog functions, their use should be minimized.

To return all result column information mandated by the ODBC specification, a driver may have to perform multiple queries, joins, subqueries, or unions to return the required result set for a single call to a catalog function. These particular elements of the SQL language are performance expensive.

Applications should cache information from catalog functions so that multiple executions are unnecessary. For example, call SQLGetTypeInfo once in the application and cache the elements of the result set that your application depends on. It is unlikely that any application uses all elements of the result set generated by a catalog function, so the cached information should not be difficult to maintain.

Avoiding Search Patterns

Passing null arguments or search patterns to catalog functions generates time-consuming queries. In addition, network traffic potentially increases because of unwanted results. Always supply as many non-null arguments to catalog functions as possible. Because catalog functions are slow, applications should invoke them efficiently. Any information that the application can send the driver when calling catalog functions can result in improved performance and reliability.

For example, consider a call to SQLTables where the application requests information about the table "Customers." Often, this

call is coded as shown, using the fewest non-null arguments necessary for the function to return success:

```
rc = SQLTables (NULL, NULL, NULL, NULL, "Customers",
   SQL_NTS, NULL);
```

A driver processes this SQLTables call into SQL that looks like this:

```
SELECT ...  FROM SysTables WHERE TableName = 'Customers'
   UNION ALL
SELECT ... FROM SysViews WHERE ViewName = 'Customers'
   UNION ALL
SELECT ... FROM SysSynonyms WHERE SynName = 'Customers'
   ORDER BY ...
```

In our example, the application provides scant information about the object for which information was requested. Suppose three "Customers" tables were returned in the result set: the first table owed owned by the user, the second owned by the sales department, and the third is a view created by management.

It may not be obvious to the end user which table to choose. If the application had specified the OwnerName argument in the SQLTables call, only one table would be returned and performance would improve. Less network traffic would be required to return only one result row and unwanted rows would be filtered by the database. In addition, if the TableType argument was supplied, the SQL sent to the server can be optimized from a three-query union into a single Select statement as shown:

```
SELECT ... FROM SysTables WHERE TableName = 'Customers' and
   Owner = 'Beth'
```

Using a Dummy Query to Determine Table Characteristics

Avoid using SQLColumns to determine characteristics about a table. Instead, use a dummy query with SQLDescribeCol.

Consider an application that allows the user to choose the columns that will be selected. Should the application use SQLColumns to return information about the columns to the user or prepare a dummy query and call SQLDescribeCol?

Case 1: SQLColumns Method

```
rc = SQLColumns (... "UnknownTable" ...);
// This call to SQLColumns will generate a query to the
// system catalogs... possibly a join which must be
// prepared, executed, and produce a result set
rc = SQLBindCol (...);
rc = SQLExtendedFetch (...);
// user must retrieve N rows from the server
// N = # result columns of UnknownTable
// result column information has now been obtained
```

Case 2: SQLDescribeCol Method

```
// prepare dummy query
rc = SQLPrepare (... "SELECT * from UnknownTable
    WHERE 1 = 0" ...);
// query is never executed on the server - only prepared
rc = SQLNumResultCols (...);
for (irow = 1; irow <= NumColumns; irow++) {
    rc = SQLDescribeCol (...)
    // + optional calls to SQLColAttributes
    }
// result column information has now been obtained
// Note we also know the column ordering within the table!
// This information cannot be
// assumed from the SQLColumns example.
```

In both cases, a query is sent to the server, but in Case 1, the query must be evaluated and form a result set that must be sent to the client. Clearly, Case 2 is the better performing model.

To complicate this discussion, let us consider a database server that does not natively support preparing a SQL statement. The performance of Case 1 does not change, but the performance of Case 2 improves slightly because the dummy query is evaluated before being prepared. Because the Where clause of the query always evaluates to FALSE, the query generates no result rows and should execute without accessing table data. Again, for this situation, Case 2 outperforms Case 1.

Retrieving Data

To retrieve data efficiently, return only the data that you need, and choose the most efficient method of doing so. The guidelines in this section will help you optimize system performance when retrieving data with ODBC applications.

Retrieving Long Data

Unless it is necessary, applications should not request long data (SQL_LONGVARCHAR and SQL_LONGVARBINARY data) because retrieving long data across the network is slow and resource-intensive. Most users do not want to see long data. If the user does need to see these result items, the application can query the database again, specifying only long columns in the select list. This method allows the average user to retrieve the result set without having to pay a high performance penalty for network traffic.

Although the best method is to exclude long data from the select list, some applications do not formulate the select list before

sending the query to the ODBC driver (that is, some applications simply `SELECT * FROM table_name ...`). If the select list contains long data, the driver must retrieve that data at fetch time even if the application does not bind the long data in the result set. When possible, use a method that does not retrieve all columns of the table.

Reducing the Size of Data Retrieved

To reduce network traffic and improve performance, you can reduce the size of data being retrieved to some manageable limit by calling SQLSetStmtAttr with the SQL_ATTR_MAX_LENGTH option. This reduces network traffic and improves performance.

Although eliminating SQL_LONGVARCHAR and SQL_LONGVARBINARY data from the result set is ideal for performance optimization, sometimes, long data must be retrieved. When this is the case, remember that most users do not want to see 100 KB, or more, of text on the screen. What techniques, if any, are available to limit the amount of data retrieved?

Many application developers mistakenly assume that if they call SQLGetData with a container of size x that the ODBC driver only retrieves x bytes of information from the server. Because SQLGetData can be called multiple times for any one column, most drivers optimize their network use by retrieving long data in large chunks and then returning it to the user when requested. For example:

```
char CaseContainer[1000];
...
rc = SQLExecDirect (hstmt, "SELECT CaseHistory FROM Cases
   WHERE CaseNo = 71164", SQL_NTS);
...
rc = SQLFetch (hstmt);
```

```
rc = SQLGetData (hstmt, 1, CaseContainer,(SWORD)
sizeof(CaseContainer), ...);
```

At this point, it is more likely that an ODBC driver will retrieve 64 KB of information from the server instead of 1000 bytes. In terms of network access, one 64-KB retrieval is less expensive than 64 retrievals of 1000 bytes. Unfortunately, the application may not call SQLGetData again; therefore, the first and only retrieval of CaseHistory would be slowed by the fact that 64 KB of data must be sent across the network.

Many ODBC drivers allow you to limit the amount of data retrieved across the network by supporting the SQL_MAX_LENGTH attribute. This attribute allows the driver to communicate to the database server that only *x* bytes of data are relevant to the client. The server responds by sending only the first *x* bytes of data for all result columns. This optimization substantially reduces network traffic and improves client performance. The previous example returned only one row, but consider the case where 100 rows are returned in the result set—the performance improvement would be substantial.

Using Bound Columns

Retrieving data through bound columns (SQLBindCol) instead of using SQLGetData reduces the ODBC call load and improves performance.

Consider the following code fragment:

```
rc = SQLExecDirect (hstmt, "SELECT <20 columns>
   FROM Employees WHERE HireDate >= ?", SQL_NTS);
do {
rc  = SQLFetch (hstmt);
// call SQLGetData 20 times
} while ((rc == SQL_SUCCESS) || (rc == SQL_SUCCESS_WITH_INFO));
```

Suppose the query returns 90 result rows. In this case, more than 1890 ODBC calls are made (20 calls to SQLGetData x 90 result rows + 91 calls to SQLFetch).

Consider the same scenario that uses SQLBindCol instead of SQLGetData:

```
rc = SQLExecDirect (hstmt, "SELECT <20 columns>
   FROM Employees WHERE HireDate >= ?", SQL_NTS);
// call SQLBindCol 20 times
do {
rc  = SQLFetch (hstmt);
} while ((rc == SQL_SUCCESS) || (rc ==
   SQL_SUCCESS_WITH_INFO));
```

The number of ODBC calls made is reduced from more than 1890 to about 110 (20 calls to SQLBindCol + 91 calls to SQLFetch). In addition to reducing the call load, many drivers optimize how SQLBindCol is used by binding result information directly from the database server into the user's buffer. That is, instead of the driver retrieving information into a container and then copying that information to the user's buffer, the driver simply requests the information from the server be placed directly into the user's buffer.

Using SQLExtendedFetch Instead of SQLFetch

Use SQLExtendedFetch to retrieve data instead of SQLFetch. The ODBC call load decreases (resulting in better performance) and the code is less complex (resulting in more maintainable code).

Most ODBC drivers now support SQLExtendedFetch for forward only cursors; yet, most ODBC applications use SQLFetch to

retrieve data. Again, consider the preceding example using SQLExtendedFetch instead of SQLFetch:

```
rc = SQLSetStmtOption (hstmt, SQL_ROWSET_SIZE, 100);
// use arrays of 100 elements
rc = SQLExecDirect (hstmt, "SELECT <20 columns>
   FROM Employees WHERE HireDate >= ?", SQL_NTS);
// call SQLBindCol 1 time specifying row-wise binding
do {
rc = SQLExtendedFetch (hstmt, SQL_FETCH_NEXT, 0,
   &RowsFetched,RowStatus);
} while ((rc == SQL_SUCCESS) || (rc == SQL_SUCCESS_WITH_INFO));
```

Notice the improvement from the previous examples. The initial call load was more than 1890 ODBC calls. By choosing ODBC calls carefully, the number of ODBC calls made by the application has now been reduced to 4 (1 SQLSetStmtOption + 1 SQLExecDirect + 1 SQLBindCol + 1 SQLExtendedFetch). In addition to reducing the call load, many ODBC drivers retrieve data from the server in arrays, further improving the performance by reducing network traffic.

For ODBC drivers that do not support SQLExtendedFetch, the application can enable forward-only cursors using the ODBC cursor library (call SQLSetConnectOption using SQL_ODBC_CURSORS or SQL_CUR_USE_IF_NEEDED). Although using the cursor library does not improve performance, it should not be detrimental to application response time when using forward only cursors (no logging is required). Furthermore, using the cursor library when SQLExtendedFetch is not supported natively by the driver simplifies the code because the application can always depend on SQLExtendedFetch being available. The application does not require two algorithms (one using SQLExtendedFetch and one using SQLFetch).

Choosing the Right Data Type

Advances in processor technology brought significant improvements to the way that operations such as floating-point math are handled; however, retrieving and sending certain data types are still expensive when the active portion of your application will not fit into on-chip cache. When you are working with data on a large scale, it is still important to select the data type that can be processed most efficiently. For example, integer data is processed faster than floating-point data. Floating-point data is defined according to internal database-specific formats, usually in a compressed format. The data must be decompressed and converted into a different format so that it can be processed by the wire protocol.

Processing time is shortest for character strings, followed by integers, which usually require some conversion or byte ordering. Processing floating-point data and timestamps is at least twice as slow as processing integers.

Selecting ODBC Functions

The guidelines in this section will help you select which ODBC functions will give you the best performance.

Using SQLPrepare/SQLExecute and SQLExecDirect

Using SQLPrepare/SQLExecute is not always as efficient as SQLExecDirect. Use SQLExecDirect for queries that will be executed once and SQLPrepare/SQLExecute for queries that will be executed multiple times.

ODBC drivers are optimized based on the perceived use of the functions that are being executed. SQLPrepare/SQLExecute is optimized for multiple executions of statements that use parameter markers. SQLExecDirect is optimized for a single execution of a SQL statement. Unfortunately, more than 75% of all ODBC applications use SQLPrepare/SQLExecute exclusively.

Consider the case where an ODBC driver implements SQLPrepare by creating a stored procedure on the server that contains the prepared statement. Creating stored procedures involve substantial overhead, but the statement can be executed multiple times. Although creating stored procedures is performance-expensive, execution is minimal because the query is parsed and optimization paths are stored at create procedure time.

Using SQLPrepare/SQLExecute for a statement that is executed only once results in unnecessary overhead. Furthermore, applications that use SQLPrepare/SQLExecute for large single execution query batches exhibit poor performance. Similarly, applications that always use SQLExecDirect do not perform as well as those that use a logical combination of SQLPrepare/SQLExecute and SQLExecDirect sequences.

Using Arrays of Parameters

Passing arrays of parameter values for bulk insert operations, for example, with SQLPrepare/SQLExecute and SQLExecDirect can reduce the ODBC call load and network traffic. To use arrays of parameters, the application calls SQLSetStmtAttr with the following attribute arguments:

- SQL_ATTR_PARAMSET_SIZE sets the array size of the parameter.
- SQL_ATTR_PARAMS_PROCESSED_PRT assigns a variable filled by SQLExecute, which contains the number of rows that are actually inserted.

- SQL_ATTR_PARAM_STATUS_PTR points to an array in which status information for each row of parameter values is returned.

NOTE: With ODBC 3.*x*, calls to SQLSetStmtAttr with the SQL_ATTR_PARAMSET_SIZE, SQL_ATTR_PARAMS_PROCESSED_ARRAY, and SQL_ATTR_PARAM_STATUS_PRT arguments replace the ODBC 2.*x* call to SQLParamOptions.

Before executing the statement, the application sets the value of each data element in the bound array. When the statement is executed, the driver tries to process the entire array contents using one network roundtrip. For example, let us compare the following examples, Case 1 and Case 2.

Case 1: Executing Prepared Statement Multiple Times

```
rc = SQLPrepare (hstmt, "INSERT INTO DailyLedger (...)
   VALUES (?,?,...)", SQL_NTS);
// bind parameters
...
do {
// read ledger values into bound parameter buffers
...
rc = SQLExecute (hstmt);
// insert row
} while ! (eof);
```

Case 2: Using Arrays of Parameters

```
SQLPrepare (hstmt, " INSERT INTO DailyLedger (...) VALUES
   (?,?,...)", SQL_NTS);
SQLSetStmtAttr (hstmt, SQL_ATTR_PARAMSET_SIZE, (UDWORD)100,
   SQL_IS_UINTEGER);
SQLSetStmtAttr (hstmt, SQL_ATTR_PARAMS_PROCESSED_PRT,
   &rows_processed, SQL_IS_POINTER);
// Specify an array in which to return the status of
// each set of parameters.
SQLSetStmtAttr(hstmt, SQL_ATTR_PARAM_STATUS_PTR,
   ParamStatusArray, SQL_IS_POINTER);
```

```
// pass 100 parameters per execute
// bind parameters
...
do {
// read up to 100 ledger values into
// bound parameter buffers
...
rc = SQLExecute (hstmt);
// insert a group of 100 rows
} while ! (eof);
```

In Case 1, if there are 100 rows to insert, 101 network roundtrips are required to the server, one to prepare the statement with SQLPrepare and 100 additional roundtrips for each time SQLExecute is called.

In Case 2, the call load has been reduced from 100 SQLExecute calls to only 1 SQLExecute call. Furthermore, network traffic is reduced considerably.

Using the Cursor Library

If the driver provides scrollable cursors, do not use the cursor library automatically. The cursor library creates local temporary log files, which are performance-expensive to generate and provide worse performance than native scrollable cursors.

The cursor library adds support for static cursors, which simplifies the coding of applications that use scrollable cursors. However, the cursor library creates temporary log files on the user's local disk drive to accomplish the task. Typically, disk input/output is a slow operation. Although the cursor library is beneficial, applications should not automatically choose to use the cursor library when an ODBC driver supports scrollable cursors natively.

Typically, ODBC drivers that support scrollable cursors achieve high performance by requesting that the database server

produce a scrollable result set instead of emulating the capability by creating log files. Many applications use:

```
rc = SQLSetConnectOption (hdbc, SQL_ODBC_CURSORS,
    SQL_CUR_USE_ODBC);
```

but should use:

```
rc = SQLSetConnectOption (hdbc, SQL_ODBC_CURSORS,
    SQL_CUR_USE_IF_NEEDED);
```

Managing Connections and Updates

The guidelines in this section will help you to manage connections and updates to improve system performance for your ODBC applications.

Managing Connections

Connection management is important to application performance. Optimize your application by connecting once and using multiple statement handles, instead of performing multiple connections. Avoid connecting to a data source after establishing an initial connection.

Although gathering driver information at connect time is a good practice, it is often more efficient to gather it in one step rather than two steps. Some ODBC applications are designed to call informational gathering routines that have no record of already attached connection handles. For example, some applications establish a connection and then call a routine in a separate DLL or shared library that reattaches and gathers information about the driver. Applications that are designed as separate entities should pass the already connected HDBC pointer to the data collection routine instead of establishing a second connection.

Another bad practice is to connect and disconnect several times throughout your application to process SQL statements. Connection handles can have multiple statement handles associated with them. Statement handles can provide memory storage for information about SQL statements. Therefore, applications do not need to allocate new connection handles to process SQL statements. Instead, applications should use statement handles to manage multiple SQL statements.

On Windows, you can significantly improve performance with connection pooling, especially for applications that connect over a network or through the World Wide Web. With connection pooling, closing connections does not close the physical connection to the database. When an application requests a connection, an active connection from the connection pool is reused, avoiding the network input/output needed to create a new connection.

Connection and statement handling should be addressed before implementation. Spending time and thoughtfully handling connection management improves application performance and maintainability.

Managing Commits in Transactions

Committing data is extremely disk input/output intensive and slow. If the driver can support transactions, always turn autocommit off.

What does a commit actually involve? The database server must flush back to disk every data page that contains updated or new data. This is not a sequential write but a searched write to replace existing data in the table. By default, autocommit is on when connecting to a data source. Autocommit mode usually impairs system performance because of the significant amount of disk input/output needed to commit every operation.

Some database servers do not provide an Autocommit mode. For this type of server, the ODBC driver must explicitly issue a COMMIT statement and a BEGIN TRANSACTION for every operation sent to the server. In addition to the large amount of disk input/output required to support Autocommit mode, a performance penalty is paid for up to three network requests for every statement issued by an application.

Although using transactions can help application performance, do not take this tip too far. Leaving transactions active can reduce throughput by holding locks on rows for long times, preventing other users from accessing the rows. Commit transactions in intervals that allow maximum concurrency.

Choosing the Right Transaction Model

Many systems support distributed transactions; that is, transactions that span multiple connections. Distributed transactions are at least four times slower than normal transactions due to the logging and network input/output necessary to communicate between all the components involved in the distributed transaction. Unless distributed transactions are required, avoid using them. Instead, use local transactions when possible.

Using Positional Updates and Deletes

Use positional updates and deletes or SQLSetPos to update data. Although positional updates do not apply to all types of applications, developers should use positional updates and deletes when it makes sense. Positional updates (either through "update where current of cursor" or through SQLSetPos) allow the developer to signal the driver to "change the data here" by positioning the database cursor at the appropriate row to be changed. The designer is not forced to build a complex SQL statement but simply supplies the data to be changed.

In addition to making the application more maintainable, positional updates usually result in improved performance. Because the database server is already positioned on the row for the Select statement in process, performance-expensive operations to locate the row to be changed are not needed. If the row must be located, the server typically has an internal pointer to the row available (for example, ROWID).

Using SQLSpecialColumns

Use SQLSpecialColumns to determine the optimal set of columns to use in the Where clause for updating data. Often, pseudo-columns provide the fastest access to the data, and these columns can only be determined by using SQLSpecialColumns.

Some applications cannot be designed to take advantage of positional updates and deletes. These applications typically update data by forming a Where clause consisting of some subset of the column values returned in the result set. Some applications may formulate the Where clause by using all searchable result columns or by calling SQLStatistics to find columns that are part of a unique index. These methods typically work, but can result in fairly complex queries.

Consider the following example:

```
rc = SQLExecDirect (hstmt, "SELECT first_name, last_name,
   ssn, address, city, state, zip FROM emp", SQL_NTS);
// fetchdata
...
rc = SQLExecDirect (hstmt, "UPDATE EMP SET ADDRESS = ?
   WHERE first_name = ? and last_name = ? and ssn = ? and
   address = ? and city = ? and state = ? and zip = ?",
    SQL_NTS);
// fairly complex query
```

Applications should call SQLSpecialColumns/SQL_BEST_ROWID to retrieve the optimal set of columns (possibly a pseudo-column)

that identifies a given record. Many databases support special columns that are not explicitly defined by the user in the table definition but are "hidden" columns of every table (for example, ROWID and TID). These pseudo-columns provide the fastest access to data because they typically point to the exact location of the record. Because pseudo-columns are not part of the explicit table definition, they are not returned from SQLColumns. To determine if pseudo-columns exist, call SQLSpecialColumns.

Consider the previous example again:

```
...
rc = SQLSpecialColumns (hstmt, ..... 'emp', ...);
...
rc = SQLExecDirect (hstmt, "SELECT first_name, last_name,
   ssn, address, city, state, zip, ROWID FROM emp",
   SQL_NTS);
// fetch data and probably "hide" ROWID from the user
...
rc = SQLExecDirect (hstmt, "UPDATE emp SET address = ?
   WHERE ROWID = ?",SQL_NTS);
// fastest access to the data!
```

If your data source does not contain special pseudo-columns, the result set of SQLSpecialColumns consists of columns of the optimal unique index on the specified table (if a unique index exists). Therefore, your application does not need to call SQLStatistics to find the smallest unique index.

F Values for IANAAppCodePage Connection String Attribute

Table F-1 lists supported values, along with a description, for the IANAAppCodePage connection string attribute at the time of this publication. Support for additional values may have been added since publication time; therefore, for up-to-date values, go to:

http://www.datadirect.com/support/troubleshooting/su-faq-iana/index.ssp

See the appropriate individual driver chapter for information about this attribute.

To determine the correct numeric value (the MIBenum value) for the IANAAppCodePage connection string attribute, do the following:

1. Determine the code page of your database.
2. Determine the MIBenum value that corresponds to your database code page. To do this, go to:

   http://www.iana.org/assignments/character-sets

   On this web page, search for the name of your database code page. This name will be listed as an alias or the name of a character set and will have a MIBenum value associated with it.
3. Check the table in this appendix to make sure that the MIBenum value you looked up on the IANA Web page is supported by the DataDirect Connect Series *for* ODBC. If the value is not listed, contact SupportLink to request that support for that value be added.

Table F-1. IANAAppCodePage Values

Value (MIBenum)	Description
3	US_ASCII
4	ISO_8859_1
5	ISO_8859_2
6	ISO_8859_3
7	ISO_8859_4
8	ISO_8859_5
9	ISO_8859_6
10	ISO_8859_7
11	ISO_8859_8
12	ISO_8859_9
16	JIS_Encoding
17	Shift_JIS
18	EUC_JP
30	ISO_646_IRV
36	KS_C_5601
37	ISO_2022_KR
38	EUC_KR
39	ISO_2022_JP
40	ISO_2022_JP_2
57	GB_2312_80
104	ISO_2022_CN
105	ISO_2022_CN_EXT
109	ISO_8859_13
110	ISO_8859_14
111	ISO_8859_15
113	GBK
2004	HP_ROMAN8

Table F-1. IANAAppCodePage Values (cont.)

Value (MIBenum)	Description
2009	IBM850
2010	IBM852
2011	IBM437
2013	IBM862
2024	WINDOWS-31J
2025	GB2312
2026	Big5
2027	MACINTOSH
2028	IBM037
2029	IBM038
2030	IBM273
2033	IBM277
2034	IBM278
2035	IBM280
2037	IBM284
2038	IBM285
2039	IBM290
2040	IBM297
2041	IBM420
2043	IBM424
2044	IBM500
2045	IBM851
2046	IBM855
2047	IBM857
2048	IBM860
2049	IBM861
2050	IBM863
2051	IBM864

Table F-1. IANAAppCodePage Values (cont.)

Value (MIBenum)	Description
2052	IBM865
2053	IBM868
2054	IBM869
2055	IBM870
2056	IBM871
2062	IBM918
2063	IBM1026
2084	KOI8_R
2085	HZ_GB_2312
2086	IBM866
2087	IBM775
2089	IBM00858
2091	IBM01140
2092	IBM01141
2093	IBM01142
2094	IBM01143
2095	IBM01144
2096	IBM01145
2097	IBM01146
2098	IBM01147
2099	IBM01148
2100	IBM01149
2102	IBM1047
2250	WINDOWS_1250
2251	WINDOWS_1251
2252	WINDOWS_1252
2253	WINDOWS_1253
2254	WINDOWS_1254

Table F-1. IANAAppCodePage Values (cont.)

Value (MIBenum)	Description
2255	WINDOWS_1255
2256	WINDOWS_1256
2257	WINDOWS_1257
2258	WINDOWS_1258
2259	TIS_620
2000000939[1]	IBM-939
2000000943[1]	IBM-943_P14A-2000
2000004396[1]	IBM-4396
2000005026[1]	IBM-5026
2000005035[1]	IBM-5035

[1]These values are assigned by DataDirect Technologies and do not appear in http://www.iana.org/assignments/character-sets.

G The UNIX/Linux Environments

This appendix contains specific information about using DataDirect Connect Series *for* ODBC in the UNIX and Linux environments. It discusses the following:

- "Environment Variables" on page 93
- "The ivtestlib/ddtestlib Tool" on page 96
- "Data Source Configuration" on page 97
- "demoodbc" on page 99
- "example" on page 100
- "DSN-less Connections" on page 101
- "File Data Sources" on page 102
- "UTF-16 Applications on UNIX and Linux" on page 104

See "Environment-Specific Information" on page 19 for additional platform information.

Environment Variables

The drivers require several environment variables to be set before using the driver. The following procedures require that you have the appropriate permissions to modify your environment and to read, write and execute various files. You should log in as a user with full r/w/x permissions recursively on the entire DataDirect Connect Series *for* ODBC installation directory.

Required Environment Variables

Most of the variables can be set by executing the appropriate shell script located in the ODBC home directory.

For example, C shell (and related shell) users should execute the following command before attempting to use ODBC-enabled applications:

```
% source ./odbc.csh
```

Bourne shell (and related shell) users should initialize their environment as follows:

```
$ . ./odbc.sh
```

Executing these scripts will set the appropriate library search path environment variable: LD_LIBRARY_PATH on Solaris and Linux, SHLIB_PATH on HP/UX, or LIBPATH on AIX for the 32-bit drivers; and LD_LIBRARY_PATH on Solaris, Linux, and HP-UX, or LIBPATH on AIX for the 54-bit drivers.

The library search path environment variables are required to be set so that the ODBC core components and drivers can be located at the time of execution.

Some of the drivers must have environment variables set as required by the database client components used by the drivers. Consult the driver requirements in each of the individual driver sections for additional information pertaining to individual driver requirements.

ODBCINI

UNIX and Linux permit the use of a centralized system information file that a system administrator can control. This file contains data source definitions. Setup installs a default version of this file, called odbc.ini (see "Data Source Configuration" on page 97 for details). The system administrator can choose to use

this file name or to rename the file. In either case, the environment variable ODBCINI, recognized by the DataDirect Connect Series *for* ODBC driver, must be set to point to the fully qualified path name of the system information file.

For example, to point to the default location of the file odbc.ini in the C shell, you would set this variable as follows:

```
setenv ODBCINI /opt/odbc32v51/odbc.ini
```

or

```
setenv ODBCINI /opt/odbc64v50/odbc.ini
```

In the Bourne or Korn shell, you would set it as:

```
ODBCINI=/opt/odbc32v51/odbc.ini;export ODBCINI
```

or

```
ODBCINI=/opt/odbc64v50/odbc.ini;export ODBCINI
```

As an alternative, you can choose to make the system information file a hidden file and not set the ODBCINI variable. In this case, you would need to rename the file to .odbc.ini (to make it a hidden file) and move it to the user's $HOME directory.

The driver searches for the system information file in the following order:

1 Check ODBCINI

2 Check $HOME for .odbc.ini

The system administrator must use one of the preceding two steps to indicate the location of the system information file or the driver will return an error.

The next step is to test load the driver.

The ivtestlib/ddtestlib Tool

The second step in preparing to use a driver is to test load it. The ivtestlib (ddtestlib for the 64-bit drivers) tool is provided to help diagnose configuration problems in the UNIX and Linux environments (such as environment variables not correctly set or missing database client components). This tool is installed in the bin subdirectory in the product installation directory. It attempts to load a specified ODBC driver and prints out all available error information if the load fails.

for example, if the driver is installed in /opt/odbc/lib, the command:

```
ivtestlib /opt/odbc/lib/ivnsk21.so
```

or

```
ddtestlib /opt/odbc/lib/ddnsk21.so
```

attempts to load the NSK SQL/MX Wire Protocol driver. If the driver cannot be loaded, the tool returns an error message explaining why.

NOTE: On Solaris, AIX, and Linux, the full path to the driver does not have to be specified for the tool. The HP-UX version of the tool, however, requires the full path.

Also, you can use ivtestlib to check the version strings of the ODBC driver on UNIX and Linux. See "Version String Information" on page 25 for details.

The next step is to configure a data source through the system information file.

Data Source Configuration

In the UNIX and Linux environments, there is no ODBC Administrator, but a centralized system information file can be used to store data sources. To configure a data source, you must create a data source definition by editing the system information file. Setup installs a default version of this file, called odbc.ini, in the installation directory (see "ODBCINI" on page 94 for details about relocating and renaming this file). This a plain text file that can be modified using any text editor to create data source definitions. Modify the default attributes in this file as necessary based on your system (for example, your server name and port number). Consult the Table 3-1 of the driver chapter for specific attribute values.

IMPORTANT: The "Connection String Attributes" table of each driver chapter lists both the long and short name of the attribute. When entering attribute names into odbc.ini, you must use the long name of the attribute. The short name is not valid in the odbc.ini file.

There must be an [ODBC] section in the system information file that includes the InstallDir keyword. The value of this keyword must be the path to the installation directory under which the /lib and /messages directories are contained. The installation process automatically writes your installation directory to the default odbc.ini.

For example, if you choose the default installation directory, then the following line is written to the [ODBC] section of the default odbc.ini:

```
InstallDir=/opt/odbc32v51
```

or

```
InstallDir=/opt/odbc64v51
```

See "Sample System Information File" on page 98 for an example.

Translators

DataDirect provides a sample translator named "OEM to ANSI" that provides a framework for coding a translation library.

To perform a translation, you must include the keyword TranslationSharedLibrary in the data source sections of the system information file (see "Sample System Information File" on page 98). Adding the TranslationOption keyword is optional.

Keyword	Definition
TranslationSharedLibrary	Full path of translation library
TranslationOption	ASCII representation of the 32-bit integer translation option

For example, with the 32-bit driver:

```
[NSK Wire Protocol]
Driver=ODBCHOME/lib/ivnsk21.so
Description=DataDirect 5.1 NSK SQL/MX Wire Protocol
TranslationSharedLibrary=ODBCHOME/lib/ivtrn21.so
```

See the readme.trn file in the /src/trn directory beneath the product installation directory.

Sample System Information File

The following is a sample 32-bit odbc.ini file on Linux that Setup installs in the installation directory. The only difference for a 64-bit file is the file name: ddnsk21.so. All occurrences of ODBCHOME are replaced with your installation directory path during installation of the file. Values that you must supply are enclosed by angle brackets (< >). If you are using the supplied

odbc.ini file, you must supply these values and remove the angle brackets before that data source section will operate properly.

```
[ODBC Data Sources]
NSK Wire Protocol=DataDirect 5.1 NSK SQL/MX Wire Protocol

[NSK Wire Protocol]
Driver=ODBCHOME/lib/ivnsk21.so
Description=DataDirect 5.1 NSK SQL/MX Wire Protocol
HostName=<NSK_server_address>
NskCatalog=mycatlog
NskDatasource="TDM_Default_DataSource"
NSKSchema=myschema
NskTransactionIsolationLevel=2
NskWindowText=ivnsk
PortNumber=<NSK_server_port>

[ODBC]
IANAAppCodePage=4
InstallDir=ODBCHOME
Trace=0
TraceDll=ODBCHOME/lib/odbctrac.so
TraceFile=odbctrace.out
UseCursorLib=0
```

The final step is to verify that the driver connects and passes SQL.

demoodbc

DataDirect ships an application, called demoodbc, which is installed in the demo subdirectory beneath the product installation directory. Once you have set up your environment and data source, you can use the demoodbc application to test your connection. The syntax to run the application is:

```
demoodbc -uid user_name -pwd password data_source_name
```

For example:

```
demoodbc -uid johndoe -pwd secret DataSource3
```

The demoodbc application is coded to execute a Select statement from a table named EMP. If you have an EMP table in your database, the results are returned. If you do not have an EMP table, you receive the message, `The specified table (emp) is not in the database`. This message should be viewed as a successful connection to the database.

See the readme in the demo directory for an explanation of how to build and use this application. See "demoodbc Application" on page 107 for additional information.

example

DataDirect ships an application, called example, which is installed in the example subdirectory in the product installation directory. Once you have configured your environment and data source, you can use the example application to test passing SQL statements. To run the application, enter `example` and follow the prompts to enter your data source name, user name, and password.

If successful, a SQL> prompt appears and you can type in SQL Statements such as `SELECT * FROM table_name.` If example is unable to connect, an appropriate error message appears.

See the readme in the example directory for an explanation of how to build and use this application. See "example Application" on page 108 for additional information.

DSN-less Connections

Connections to a data source can be made via a connection string without referring to a data source name (DSN-less connections). This is accomplished by specifying the "DRIVER=" instead of the "DSN=" keyword in a connection string, as outlined in the ODBC specification. For this to work on UNIX and Linux, a file called odbcinst.ini must exist when the driver encounters DRIVER= in a connection string.

By default, Setup installs a sample odbcinst.ini file in the same location as the sample odbc.ini file, which is in the product installation directory. See "Data Source Configuration" on page 97 for an explanation of the odbc.ini file. The environment variable ODBCINST, recognized by the DataDirect Connect Series *for* ODBC driver, must be set to point to the fully qualified path name of the odbcinst.ini file.

For example, to point to the default location of the file in the C shell, you would set this variable as follows:

```
setenv ODBCINST /opt/odbc32v51/odbcinst.ini
```

or

```
setenv ODBCINST /opt/odbc64v51/odbcinst.ini
```

In the Bourne or Korn shell, you would set it as:

```
ODBCINST=/opt/odbc32v51/odbcinst.ini;export ODBCINST
```

or

```
ODBCINST=/opt/odbc64v51/odbcinst.ini;export ODBCINST
```

If the ODBCINST variable is not set, the driver looks in the user's home directory for a file named odbcinst.ini. If the driver does not find the file, it returns the message:

```
HY000 - "ODBCINST.INI is not available in the directory
pointed to by the ODBCINST environment variable (or the
```

```
current user's HOME directory) and therefore DSN-Less
connections cannot be made."
```

The following is a sample 32-bit odbcinst.ini file on Linux that Setup installs in the installation directory. The only difference for a 64-bit file is the file name: ddnsk21.so. All occurrences of ODBCHOME are replaced with your installation directory path during installation of the file.

```
[ODBC Drivers]
DataDirect 5.1 NSK SQL/MX Wire Protocol=Installed

[DataDirect 5.1 NSK SQL/MX Wire Protocol]
Driver=ODBCHOME/lib/ivnsk21.so
APILevel=1
ConnectFunctions=YYY
DriverODBCVer=3.52
FileUsage=0
SQLevel=1

[ODBC Translators]
OEM to ANSI=Installed

[ODBC Core]
```

File Data Sources

The Driver Manager on UNIX and Linux supports file data sources. The advantage of a file data source is that it can be stored on a server and accessed by other machines, either Windows, UNIX, or Linux, with the same type of driver. See Chapter 1 "Quick Start Connect" on page 13 for a general description of ODBC data sources.

A file data source is simply a text file that contains connection information. It can be created with a text editor. The file normally has an extension of .dsn.

For example, a file data source for the NSK SQL/MX Wire Protocol driver would be similar to the following:

```
[ODBC]
DRIVER=DataDirect 5.1 NSK SQL/MX Wire Protocol
HostName=NSK2
NskCatalog=<NSK_database_catalog_name>
NSKSchema<=NSK_database_schema_name>
UserID=JOHN
PortNumber=111
```

It must contain all basic connection information plus any optional attributes. Because it uses the "DRIVER=" keyword, an odbcinst.ini file containing the driver location must exist (see "DSN-less Connections" on page 101).

The file data source is accessed by specifying the "FILEDSN=" instead of the "DSN=" keyword in a connection string, as outlined in the ODBC specification. The complete path to the file data source must be specified in the syntax that is normal for the machine on which the file is located. For example:

```
FILEDSN=/home/users/john/filedsn/NSKFile.dsn
```

As with any connection string, attributes can be specified to override the default values in the data source:

```
FILEDSN=/home/users/john/filedsn/NSKFile.dsn;UID=james;PWD=test01
```

If no path is specified for the file data source, the Driver Manager uses the installation directory setting in the odbc.ini file. The Driver Manager does not support the SAVEFILE keyword or the SQLReadFileDSN and SQLWriteFileDSN functions.

UTF-16 Applications on UNIX and Linux

Because the DataDirect Driver Manager allows applications to use either UTF-8 or UTF-16 Unicode encoding, applications written in UTF-16 for Windows platforms can also be used on UNIX and Linux platforms.

The Driver Manager assumes a default of UTF-8 applications; therefore, two things must occur for it to determine that the application is UTF-16:

- The definition of SQLWCHAR in the ODBC header files must be switched from "char *" to "short *." To do this, the application uses #define SQLWCHARSHORT.
- The application must set the ODBC environment attribute SQL_ATTR_APP_UNICODE_TYPE to a value of SQL_DD_CP_UTF16, for example:

```
rc = SQLSetEnvAttr(*henv, SQL_ATTR_APP_UNICODE_TYPE,
(SQLPOINTER)SQL_DD_CP_UTF16, SQL_IS_INTEGER);
```

H Diagnostic Tools, Error Messages, and Troubleshooting

This appendix discusses the diagnostic tools that are available to configure and troubleshoot your ODBC environment, explains the error messages that can be returned from the DataDirect Connect Series *for* ODBC, and provides a troubleshooting section that discusses some common types of issues that you may experience when using ODBC applications. This appendix includes the following:

- "Diagnostic Tools" on page 105
- "Error Messages" on page 108
- "Troubleshooting" on page 110

Diagnostic Tools

This section discusses the diagnostic tools that are available to you when you are configuring and troubleshooting your ODBC environment.

ODBC Trace

ODBC tracing allows you to trace calls to ODBC drivers and create a log of the traces. Creating a trace log is particularly useful when you are troubleshooting an issue.

To create a trace log, turn on tracing, start the ODBC application, reproduce the issue, stop the application, and turn off tracing. Then, open the log file in a text editor and review the output to help you debug the problem. Be sure to turn off tracing when you are finished reproducing the issue because tracing decreases the performance of your ODBC application.

For a more thorough explanation of tracing, see the following DataDirect Knowledgebase document:

http://knowledgebase.datadirect.com/kbase.nsf/SupportLink+Online/2549739SL

Enabling Tracing

On UNIX and Linux, the [ODBC] section in the system information file must include the keywords Trace and TraceFile. For example:

```
Trace=1
TraceFile=odbctrace.out
```

In this example, tracing is enabled and trace information is logged in a file named odbctrace.out. To disable tracing, set the Trace value to 0.

ivtestlib/ddtestlib Tool

The ivtestlib (ddtestlib for the 64-bit drivers) tool is provided to help diagnose configuration problems in the UNIX and Linux environments (such as environment variables not correctly set or missing database client components). This tool is installed in the bin subdirectory in the product installation directory. It attempts to load a specified ODBC driver and prints out all available error information if the load fails.

for example, if the driver is installed in /opt/odbc/lib, the command:

```
ivtestlib /opt/odbc/lib/ivnsk21.so
```

or

```
ddtestlib /opt/odbc/lib/ddnsk21.so
```

attempts to load the NSK SQL/MX Wire Protocol driver. If the driver cannot be loaded, the tool returns an error message explaining why.

NOTE: On Solaris, AIX, and Linux, the full path to the driver does not have to be specified for the tool. The HP-UX version of the tool, however, requires the full path.

Also, you can use ivtestlib to check the version strings of the ODBC driver on UNIX and Linux. See "Version String Information" on page 25 for details.

demoodbc Application

The product is shipped with a small C application, demoodbc, on UNIX and Linux that is useful for:

- Executing `SELECT * FROM emp`, where EMP is a database table (one for each supported database) that is provided with the product. The scripts for building the EMP database tables are in the demo subdirectory in the product installation directory.
- Testing database connections.
- Creating reproducibles.
- Persisting data to an XML data file.

Demoodbc is installed in the demo subdirectory in the product installation directory. See the readme in the demo directory for an explanation of how to build and use this application.

example Application

The product is shipped with a small C application, named example, on Windows and UNIX and Linux that is useful for:

- Executing any type of SQL statement
- Testing database connections
- Testing SQL statements
- Verifying your database environment

Example is installed in the example subdirectory in the product installation directory. See the readme in the example directory for an explanation of how to build and use this application.

Translators

DataDirect provides a sample translator named "OEM to ANSI" that provides a framework for coding a translation library.

On UNIX and Linux, see the readme.trn file in the /src/trn subdirectory in the product installation directory.

Error Messages

Error messages can come from:

- An ODBC driver
- The database system
- The ODBC driver manager

An error reported on an ODBC driver has the following format:

`[vendor] [ODBC_component] message`

ODBC_component is the component in which the error occurred. For example, an error message from the NSK SQL/MX Wire Protocol driver would look like this:

```
[DataDirect] [ODBC NSK SQL/MX driver] Invalid precision
specified.
```

If you receive this type of error, check the last ODBC call made by your application for possible problems or contact your ODBC application vendor.

An error that occurs in the data source includes the data store name, in the following format:

[*vendor*] [*ODBC_component*] [*data_store*] *message*

With this type of message, *ODBC_component* is the component that received the error from the data store indicated. For example, you may receive the following message from an NSK data store:

```
[DataDirect] [ODBC NSK SQL/MX driver] [NSK] specified
length too long
```

This type of error is generated by the database system. Check your database system documentation for more information or consult your database administrator. In this example, you would check your NSK documentation.

On UNIX and Linux, the Driver Manager is provided by DataDirect. For example, an error from the DataDirect Driver Manager might look like this:

```
[DataDirect][ODBC lib] String data code page conversion
failed.
```

UNIX and Linux error handling follows the X/Open XPG3 messaging catalog system. Localized error messages are stored in the subdirectory

locale/*localized_territory_directory*/LC_MESSAGES

where *localized_territory_directory* depends on your language.

For instance, German localization files are stored in `locale/de/LC_MESSAGES`, where `de` is the locale for German.

If localized error messages are not available for your locale, then they will contain message numbers instead of text. For example:

```
[DataDirect] [ODBC 20101 driver] 30040
```

Troubleshooting

When you are having an issue while using the DataDirect Connect Series *for* ODBC, the first thing to do is determine the type of issue that you are seeing:

- Setup/connection
- Interoperability (ODBC application, ODBC driver, ODBC Driver Manager, and/or data source)
- Performance

This section describes these three types of issues, provides some typical causes of the issues, lists some diagnostic tools that are useful to troubleshoot the issues, and, in some cases, explains possible actions you can take to resolve the issues.

Setup/Connection Issues

You are experiencing a setup/connection issue if you are encountering an error or hang while you are trying to make a database connection with the ODBC driver or are trying to configure the ODBC driver.

Some common errors that are returned by the ODBC driver if you are experiencing a setup/connection issue are:

- `Specified driver could not be loaded`
- `Data source name not found and no default driver specified`
- `Cannot open share library: libodbc.sl`
- `ORA-12203: Unable to connect to destination`
- `ORA-01017: invalid username/password; logon denied`

Troubleshooting the Issue

Some common reasons that setup/connection issues occur are:

- The library path environment variable is not set correctly for the ODBC driver: LD_LIBRARY_PATH.
- The ODBCINI environment variable is not set correctly.
- The ODBC driver's connection attributes are not set correctly in the system information file (odbc.ini in most cases). For example, the host name or port number are not correctly configured. See each individual driver chapter for a list of connection string attributes that are required for each driver to connect properly to the underlying database.
- The database and/or listener are not started.

See "Environment Variables" on page 93 for more information. See also "ivtestlib Tool" on page 106 for information about a helpful diagnostic tool.

Interoperability Issues

Interoperability issues occur when you have a working ODBC application in place. In these cases, the issue occurs in one or more of the following components of ODBC—the ODBC application, ODBC driver, ODBC Driver Manager, and/or data source. See "What Is ODBC?" on page 17 for an explanation of the components of ODBC.

Some common examples of what you might experience if you have an interoperability issue are:

- SQL statements fail to execute
- Data is returned/updated/deleted/inserted incorrectly
- A hang or core dump

Troubleshooting the Issue

When you experience an interoperability issue, you must isolate in which component the issue is occurring. Is it an ODBC application, an ODBC driver, an ODBC Driver Manager, or a data source issue?

The first step

First, test to see if your ODBC application is the source of the problem. To do this, replace your ODBC application with a smaller, simpler application. If you can reproduce the issue using a simpler ODBC application, then you know your ODBC application is **not** the cause of the issue.

On UNIX and Linux, you can use the example application that is shipped with the DataDirect Connect Series *for* ODBC. See "example Application" on page 108 for details.

The second step

Second, test to see if the data source is the source of the problem. To do this, use the native database tools that are provided by your database vendor.

The third step

If you find that neither the ODBC application nor the data source is the source of your problem, you then can troubleshoot the ODBC driver and the ODBC Driver Manager.

In this case, we recommend that you create an ODBC trace log so that you can provide this to DataDirect technical support. See "Diagnostic Tools" on page 105 for details.

Performance Issues

Developing performance-oriented ODBC applications is not an easy task. You must be willing to change your application and do some testing to see if your changes helped performance. Microsoft's *ODBC Programmer's Reference* does not provide information about system performance. In addition, ODBC drivers and the ODBC Driver Manager do not return warnings when applications run inefficiently.

Some general guidelines for developing performance-oriented ODBC applications include:

- Use catalog functions appropriately
- Retrieve only required data
- Select functions that optimize performance
- Manage connections and updates

See Appendix E "Performance Design of ODBC Applications" on page 67 for complete information.

Glossary

application An application, as it relates to the ODBC standard, performs tasks such as: requesting a connection to a data source; sending SQL requests to a data source; processing errors; and terminating the connection to a data source. It may also perform functions outside the scope of the ODBC interface.

conformance There are two types of conformance levels for ODBC drivers—ODBC API and ODBC SQL grammar (see SQL Grammar). Knowing the conformance levels helps you determine the range of functionality available through the driver, even if a particular database does not support all of the functionality of a particular level.

For ODBC API conformance, most quality ODBC drivers support Core, Level 1, and a defined set of Level 2 functions, depending on the database being accessed.

connection string A string passed in code that specifies connection information directly to the Driver Manager and driver.

data source A data source includes both the source of data itself, such as relational database, a flat-file database, or even a text file, and the connection information necessary for accessing the data. Connection information may include such things as server location, database name, logon ID, and other driver options. Data source information is usually stored in a DSN (see Data Source Name).

driver An ODBC driver communicates with the application through the Driver Manager and performs tasks such as: establishing a connection to a data source; submitting requests to the data source; translating data to and from other formats; returning results to the application; and formatting errors into a standard code and returning them to the application.

Driver Manager

The main purpose of the Driver Manager is to load drivers for the application. The Driver Manager also processes ODBC initialization calls and maps data sources to a specific driver.

DSN (Data Source Name)

A DSN stores the data source information (see Data Source) necessary for the Driver Manager to connect to the database. This can be configured either through the ODBC Administrator or in a DSN file. On Windows, the information is called a system or user DSN and is stored in the Registry. Data source information can also be stored in text configuration files, as is the case on UNIX. Applications deployed in the global assembly cache must have a strong name to handle name and version conflicts.

index

A database structure used to improve the performance of database activity. A database table can have one or more indexes associated with it.

isolation level

An isolation level represents a particular locking strategy employed in the database system to improve data consistency.The higher the isolation level number, the more complex the locking strategy behind it. The isolation level provided by the database determines how a transaction handles data consistency.

The American National Standards Institute (ANSI) defines four isolation levels:

- Read uncommitted (0)
- Read committed (1)
- Repeatable read (2)
- Serializable (3)

locking level

Locking is a database operation that restricts a user from accessing a table or record. Locking is used in situations where more than one user might try to use the same table at the same time. By locking the table or record, the system insures that only one user at a time can affect the data.

SQL Grammar

ODBC defines a core grammar that roughly corresponds to the X/Open and SQL Access Group SQL CAE specification (1992). ODBC also defines a minimum grammar, to meet a basic level of ODBC conformance, and an extended grammar, to provide for

common DBMS extensions to SQL. The following list summarizes the grammar included in each conformance level:

Minimum SQL Grammar:

- Data Definition Language (DDL): CREATE TABLE and DROP TABLE.
- Data Manipulation Language (DML): simple SELECT, INSERT, UPDATE SEARCHED, and DELETE SEARCHED.
- Expressions: simple (such as A > B + C).
- Data types: CHAR, VARCHAR, or LONG VARCHAR.

Core SQL Grammar:

- Minimum SQL grammar and data types.
- DDL: ALTER TABLE, CREATE INDEX, DROP INDEX, CREATE VIEW, DROP VIEW, GRANT, and REVOKE.
- DML: full SELECT.
- Expressions: subquery, set functions such as SUM and MIN.
- Data types: DECIMAL, NUMERIC, SMALLINT, INTEGER, REAL, FLOAT, DOUBLE PRECISION.

Extended SQL Grammar:

- Minimum and Core SQL grammar and data types.
- DML: outer joins, positioned UPDATE, positioned DELETE, SELECT FOR UPDATE, and unions.
- Expressions: scalar functions such as SUBSTRING and ABS, date, time, and timestamp literals.
- Data types: BIT, TINYINT, BIGINT, BINARY, VARBINARY, LONG VARBINARY, DATE, TIME, TIMESTAMP
- Batch SQL statements.
- Procedure calls.

Unicode

Unicode, developed by the Unicode Consortium, is a standard that attempts to provide unique coding for all international language characters. The current number of supported characters is over 95,000.

Index

A

AuthStr 32

B

binding dynamic parameters 24
binding SQL statements 24

C

catalog functions, using 68
client code page
  See code pages
code pages, IANAAppCodePage attribute 87
connection string attributes 32
connections supported 38
conventions, typographical 11

D

data source
  configuration on UNIX and Linux 97
  configuring, NSK SQL/MX Wire Protocol 29
  connecting via connection string, NSK SQL/MX Wire Protocol 30
data types, NSK SQL/MX Wire Protocol 33
DataSourceName 32
date and time functions 47
ddtestlib tool 96
debugging 110
demo tool, XML persistence 35
demoodbc application 99
documentation, about 12
driver requirements 29
driver, version string information 25
dynamic parameters, binding 24

E

environment-specific information 19
error messages
  general 108
  UNIX and Linux 109
example application 108

F

file data sources 102
File DSN 102

G

glossary 115

H

HostName 32

I

IANAAppCodePage 32
improving
  database performance 59
  index performance 59
  join performance 65
  ODBC application performance 67, 113
  record selection performance 61
indexes
  deciding which to create 63
  improving performance 59
indexing multiple fields 61
isolation levels and data consistency
  compared 54
  dirty reads 52
  non-repeatable reads 52
  phantom reads 52
isolation levels, about 52
isolation levels 37
isolation levels, specific
  read committed 53
  read uncommitted 53
  repeatable read 53
  serializable 53
ivtestlib tool 96, 106

L

locking levels, NSK SQL/MX Wire Protocol 37
locking modes and levels 55

N

NSK SQL/MX Wire Protocol driver
  connection string attributes 32
  connections supported 38
  data source
    configuring 29
    connecting via connection string 30
  data types 33
  driver requirements 29
  isolation levels 37
  locking levels 37
  ODBC conformance 37
  persisting result set as XML 35
  SQL grammar supported 37
  statements supported 38
NskCatalog 32
NskDatasource 32
NskSchema 32
NskTransactionIsolationLevel 33
NskWindowText 33
numeric functions 45

O

ODBC
  API functions 39
  designing for performance 67, 113
  scalar functions 42
ODBC conformance 37
ODBC Trace 105
OEM to ANSI translation 108

P

performance, improving
  database 59
  index 59
  join 65
  record selection 61
persisting a result set as XML 35
PortNumber 33

S

scalar functions, ODBC 42
statements supported 38
string functions 42
system functions 49
system information file (.odbc.ini) 97

T

threading, overview 57
time functions 47
trace log 105
tracing 105
translator, in the UNIX environment 108
troubleshooting 110
typographical conventions 11

U

UNIX and Linux
  code pages, IANAAppCodePage attribute 87
  data source configuration 97
  driver, NSK SQL/MX Wire Protocol 29
  environment
    ddtestlib tool 96
    introduction 93
    ivtestlib tool 96
    system information file (.odbc.ini) 97
    translators 98, 108
    variables 93
  error messages 109
  system requirements 20
UserID 33

V

version string information, driver 25