DataDirect Connect® Series *for* ODBC

Reference

Release 6.0
March 2009

Table of Contents

Preface

This book is your reference to the DataDirect Connect® Series *for* ODBC from DataDirect Technologies, which includes the following products:

- DataDirect Connect® *for* ODBC
- DataDirect Connect64® *for* ODBC
- DataDirect Connect XE (Extended Edition) *for* ODBC
- DataDirect Connect64 XE *for* ODBC

Using this Book

The content of this book assumes that you are familiar with your operating system and its commands. It contains the following information:

- Chapter 1 "Code Page Values" on page 17 provides valid values for the IANAAppCodePage connection option. This option is valid only for drivers that run on UNIX and Linux.
- Chapter 2 "ODBC API and Scalar Functions" on page 25 lists the ODBC API functions that each driver supports. Any exceptions are listed in the appropriate driver chapter in the *DataDirect Connect Series for ODBC User's Guide*, under the section "ODBC Conformance Level." This chapter also lists ODBC scalar functions.
- Chapter 3 "Threading" on page 37 discusses how ODBC ensures thread safety.
- Chapter 4 "Internationalization, Localization, and Unicode" on page 41 discusses internationalization issues concerning the use of ODBC drivers.

- Chapter 5 “Designing ODBC Applications for Performance Optimization” on page 61 provides guidelines for designing performance-oriented ODBC applications.
- Chapter 6 “Using Indexes” on page 81 provides general guidelines on how to improve performance when querying a database system.
- Chapter 7 “Locking and Isolation Levels” on page 89 provides a general discussion of isolation levels and locking.
- Chapter 8 “SSL Encryption Cipher Suites” on page 95 provides the SSL encryption cipher suites supported by the DataDirect Connect Series *for* ODBC drivers.
- Chapter 9 “DataDirect Bulk Load” on page 99 provides information about the functions and statement attributes associated with DataDirect Bulk Load.
- Chapter 10 “SQL for Flat-File Drivers” on page 121 explains the SQL statements that you can use with Btrieve, dBASE, Paradox, and text files.

In addition, “Glossary” on page 147 helps you with terminology referenced in this book.

NOTE: This book refers the reader to Web pages using URLs for more information about specific topics, including 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 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 keys or key combinations that you can use. For example, press the ENTER key. Also used for SQL reserved words.
`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.

Convention	Explanation
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.

Environment-Specific Information

The drivers are supported in the Windows, UNIX, and Linux environments. When the information provided is not applicable to all supported environments, the following symbols are used to identify that information:

The Windows symbol signifies text that is applicable only to Windows.

The UNIX symbol signifies text that is applicable only to UNIX and Linux.

About the Product Documentation

The product library consists of the following books:

- *DataDirect Connect Series for ODBC Installation Guide* details requirements and procedures for installing the product.
- *DataDirect Connect Series for ODBC User's Guide* provides information about configuring and using the product.

- *DataDirect Connect Series for ODBC Reference* provides detailed reference information about the product.
- *DataDirect Connect Series for ODBC Troubleshooting Guide* provides information about error messages and troubleshooting procedures for the product.

HTML Version

This library, except for the installation guide, is placed on your system as HTML-based online help during a normal installation of the product. It is located in the help subdirectory of the product installation directory. To use the help, you must have an Internet browser installed.

On Windows, you can access the entire Help system by selecting the help icon that appears in the DataDirect program group.

On all platforms, you can access the entire Help system by opening the following file from within your browser:

install_dir/help/help.htm

where *install_dir* is the path to the product installation directory.

Or, from a command-line environment, at a command prompt, enter:

browser_exe install_dir/help/help.htm

where *browser_exe* is the name of your browser executable and *install_dir* is the path to the product installation directory.

After the browser opens, the left pane displays the Table of Contents, Index, and Search tabs for the entire documentation library. When you have opened the main screen of the Help system in your browser, you can bookmark it in the browser for quick access later.

NOTE: Security features set in your browser can prevent the Help system from launching. A security warning message is displayed. Often, the warning message provides instructions for unblocking the Help system for the current session. To allow the Help system to launch without encountering a security warning message, the security settings in your browser can be modified. Check with your system administrator before disabling any security features.

Help is also available from the setup dialog box for each driver. When you click **Help**, your browser opens to the correct topic without opening the help Table of Contents. A grey toolbar appears at the top of the browser window.

This tool bar contains previous and next navigation buttons. If, after viewing the help topic, you want to see the entire library, click:

on the left side of the toolbar, which opens the left pane and displays the Table of Contents, Index, and Search tabs.

PDF Version

DataDirect product documentation is also provided in PDF format, which allows you to view it, perform text searches, or print it. You can view the PDF documentation using Adobe Reader. The PDF documentation is available on the product CD and also on the DataDirect Technologies Web site:

http://www.datadirect.com/techres/odbcproddoc/index.ssp

You can download the entire library as a compressed file. When you uncompress the file, it appears in the correct directory structure.

If you want to copy the documentation library from the product CD, you must maintain the same directory structure that is on the CD.

- **To copy all product books**, copy the entire \books directory to your local or network drive.
- **To copy a specific set of books**, copy that book set's directory structure (beneath the \books directory) to your local or network drive. For example, in the case of:

  `\books\odbc`

  you would copy the entire `\odbc` directory.

NOTE: Maintaining the correct directory structure allows cross-book text searches and cross-references. If you download or copy the books individually outside of their normal directory structure, their cross-book search indexes and hyperlinked cross-references to other books will not work. You can view a book individually, but it will not open other books to which it has cross-references.

To help you navigate through the library, a file named **books.pdf** is provided. This file lists each online book provided for the product. We recommend that you open this file first and, from this file, open the book you want to view.

Contacting Technical Support

DataDirect Technologies offers a variety of options to meet your technical support needs. Please visit our Web site for more details and for contact information:

http://support.datadirect.com

The DataDirect Technologies Web site provides the latest support information through our global service network. The SupportLink program provides access to support contact details, tools, patches, and valuable information, including a list of FAQs for each product. In addition, you can search our Knowledgebase for technical bulletins and other information.

To obtain technical support for an evaluation copy of the product, go to:

http://www.datadirect.com/support/eval_help/index.ssp

or contact your sales representative.

When you contact us for assistance, please provide the following information:

- The serial number that corresponds to the product for which you are seeking support, or a case number if you have been provided one for your issue. If you do not have a SupportLink contract, the SupportLink representative assisting you will connect you with our Sales team.
- Your name, phone number, email address, and organization. For a first-time call, you may be asked for full customer information, including location.
- The DataDirect product and the version that you are using.
- The type and version of the operating system where you have installed your DataDirect product.

- Any database, database version, third-party software, or other environment information required to understand the problem.
- A brief description of the problem, including, but not limited to, any error messages you have received, what steps you followed prior to the initial occurrence of the problem, any trace logs capturing the issue, and so on. Depending on the complexity of the problem, you may be asked to submit an example or reproducible application so that the issue can be recreated.
- A description of what you have attempted to resolve the issue. If you have researched your issue on Web search engines, our Knowledgebase, or have tested additional configurations, applications, or other vendor products, you will want to carefully note everything you have already attempted.
- A simple assessment of how the severity of the issue is impacting your organization.

1 Code Page Values

Table 1-1 lists supported code page values, along with a description, for the IANAAppCodePage connection option. Refer to the individual driver chapters in the *DataDirect Connect Series for ODBC User's Guide* for information about this attribute.

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

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 Table 1-1 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 support for that value.

Table 1-1. IANAAppCodePage Values

Value (MIBenum)	Description
3	US_ASCII
4	ISO_8859_1
5	ISO_8859_2
6	ISO_8859_3

Table 1-1. IANAAppCodePage Values (cont.)

Value (MIBenum)	Description
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
2009	IBM850
2010	IBM852
2011	IBM437
2013	IBM862
2024	WINDOWS-31J

Table 1-1. IANAAppCodePage Values (cont.)

Value (MIBenum)	Description
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
2052	IBM865
2053	IBM868
2054	IBM869
2055	IBM870
2056	IBM871

Table 1-1. IANAAppCodePage Values (cont.)

Value (MIBenum)	Description
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
2255	WINDOWS_1255
2256	WINDOWS_1256
2257	WINDOWS_1257
2258	WINDOWS_1258
2259	TIS_620

Table 1-1. IANAAppCodePage Values (cont.)

Value (MIBenum)	Description
2000000939[a]	IBM-939
2000000943[1]	IBM-943_P14A-2000
2000004396[1]	IBM-4396
2000005026[1]	IBM-5026
2000005035[1]	IBM-5035

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

IBM to IANA Code Page Values

Table 1-2 lists the most commonly used IBM code pages and their IANA code page equivalents. These IANA values are valid for the Character Set for CCSID 65535 connection option in the DB2 Wire Protocol driver. Refer to Chapter 5 "The DB2 Wire Protocol Driver" on page 169 in the *DataDirect Connect Series for ODBC User's Guide.*

Table 1-2. IBM to IANA Code Page Values

IBM Number	Value (MIBenum)	IANA Name
37	2028	IBM037
38	2029	IBM038
290	2039	IBM290
300	2000000939[a]	IBM-939
301[b]	2000000943[1]	IBM-943_P14A-2000
301[c]	2024	WINDOWS-31J

Table 1-2. IBM to IANA Code Page Values (cont.)

IBM Number	Value (MIBenum)	IANA Name
500	2044	IBM500
857	2047	IBM857
860	2048	IBM860
861	2049	IBM861
897	17	Shift_JIS
913	6	ISO_8859-3
914	7	ISO_8859-4
932	17	Shift_JIS
939	2000000939[1]	IBM-939
943[2]	2000000943[1]	IBM-943_P14A-2000
943[3]	2024	WINDOWS-31J
950	2026	Big5
1200	1015	UTF-16
1208	106	UTF-8
1250[2]	5	ISO_8859-2
1250[3]	2250	WINDOWS-1250
1251[2]	8	ISO_8859-5
1251[3]	2251	WINDOWS-1251
1252[2]	4	ISO_8859-1
1252[3]	2252	WINDOWS-1252
1253[2]	10	ISO_8859-7
1253[3]	2253	WINDOWS-1253
1254[2]	12	ISO_8859-9
1254[3]	2254	WINDOWS-1254
1255[2]	11	ISO_8859-8

Table 1-2. IBM to IANA Code Page Values (cont.)

IBM Number	Value (MIBenum)	IANA Name
1255[3]	2255	WINDOWS-1255
1256[2]	9	ISO_8859-6
1256[3]	2256	WINDOWS-1256
1257	2257	WINDOWS-1257
1258	2258	WINDOWS-1258
4396	2000004396[1]	IBM-4396
5026	2000005026[1]	IBM-5026
5035	2000005035[1]	IBM-5035
5297	1015	UTF-16
5304	106	UTF-8
13488	1013	UTF-16BE

a. These values are assigned by DataDirect Technologies and do not appear in http://www.iana.org/assignments/character-sets.
b. If your application runs on a UNIX or Linux platform, use this value.
c. If your application runs on a Windows platform, use this value.

Teradata Code Page Values

Table 1-3 lists code pages that are valid only for the Driver for the Teradata database. These values do not appear in http://www.iana.org/assignments/character-sets and are assigned by DataDirect Technologies. Refer to Chapter 13 "The Driver for the Teradata Database" on page 663 in the *DataDirect Connect Series for ODBC User's Guide*

Table 1-3. Teradata Code Page Values

Value (MIBenum)	Description
2000005039	ebcdic
2000005040	ebcdic037_0e
2000005041	ebcdic273_0e
2000005042	ebcdic277_0e
2000005043	hangulebcdic933_1ii
2000005044	hangulksc5601_2r4
2000005045	kanjiebcdic5026_0i
2000005046	kanjiebcdic5035_0i
2000005047	kanjieuc_0u
2000005048	kanjisjis_0s
2000005049	katakanaebcdic
2000005050	latin1252_0a
2000005051	latin1_0a
2000005052	latin9_0a
2000005053	schebcdic935_2ij
2000005054	schgb2312_1t0
2000005055	tchbig5_1r0
2000005056	tchebcdic937_3i

2 ODBC API and Scalar Functions

This chapter lists the ODBC API functions that the DataDirect Connect Series *for* ODBC drivers support. In addition, it lists the scalar functions that you use in SQL statements. This chapter includes the following topics:

API Functions

The DataDirect Connect Series *for* ODBC drivers are Level 1 compliant, that is, they support all ODBC Core and Level 1 functions. They also support a limited set of Level 2 functions. The drivers support the functions listed in Table 2-1 on page 26 and Table 2-2 on page 27. 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 in the *DataDirect Connect Series for ODBC User's Guide*.

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

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

Table 2-2. 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 these functions. Refer to the documentation for your database system to find out which functions are supported. Also, depending on the driver that you are using, all 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 scalar functions in SQL statements using the following syntax:

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

where *scalar-function* is one of the functions listed in Table 2-3 through Table 2-6. For example:

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

String Functions

Table 2-3 on page 29 lists the string functions that ODBC supports.

The string functions listed accept 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 2-3. 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*.

Table 2-3. Scalar String Functions *(cont.)*

Function	Returns
LENGTH(*string_exp*)	The number of characters in *string_exp*, excluding trailing blanks and the string termination character.
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 2-4 lists the numeric functions that ODBC supports.

The numeric functions listed accept 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 2-4. 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 2-4. 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 2-5 lists the date and time functions that ODBC supports.

The date and time functions listed accept 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 2-5. 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 2-5. 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 one of the following values: 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 2-5. 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 2-6 lists the system functions that ODBC supports.

Table 2-6. 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.

3 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 always results in improved throughput. Because of network transport and database server limitations, some drivers 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 services threaded requests, although this information is useful to an application. All the DataDirect Connect Series *for* ODBC drivers provide this information to the user through the SQLGetInfo information type 1028.

The result of calling SQLGetInfo with 1028 is a SQL_USMALLINT flag that denotes the session's thread model. A return value of 0 denotes that the session is fully thread-enabled and that all requests use 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, 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 that 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 guaranteed to be thread-safe
   // use threading at your own risk
```

Driver Threading Information

Table 3-1 summarizes the threading information available at this time for the drivers. Always consult the readme file for the most up-to-date information as threading information is subject to change with new database transport and server revisions. Currently, the XML driver is the only thread-impaired driver.

Table 3-1. Threading Information

Driver	Fully Threaded	Thread Per Connect
Btrieve	X	
dBASE	X	
DB2 Wire Protocol		X
Greenplum Wire Protocol		X
Informix Wire Protocol		X
Informix		X
MySQL Wire Protocol		X
Oracle Wire Protocol		X
Oracle		X
Paradox	X	
PostgreSQL Wire Protocol		X
SQL Server Wire Protocol		X
Sybase Wire Protocol		X
Teradata	X	
Text	X	

4 Internationalization, Localization, and Unicode

This chapter provides an overview of how internationalization, localization, and Unicode relate to each other. It also provides a background on Unicode, and how it is accommodated by Unicode and non-Unicode ODBC drivers. This chapter includes the following topics:

Internationalization and Localization

Software that has been designed for *internationalization* is able to manage different linguistic and cultural conventions transparently and without modification. The same binary copy of an application should run on any localized version of an operating system without requiring source code changes.

Software that has been designed for *localization* includes language translation (such as text messages, icons, and buttons), cultural data (such as dates, times, and currency), and other components (such as input methods and spell checkers) for meeting regional market requirements.

Properly designed applications can accommodate a localized interface without extensive modification. The applications can be designed, first, to run internationally, and, second, to accommodate the language- and cultural-specific elements of a designated locale.

Locale

A locale represents the language and cultural data chosen by the user and dynamically loaded into memory at runtime. The locale settings are applied to the operating system and to subsequent application launches.

While language is a fairly straightforward item, cultural data is a little more complex. Dates, numbers, and currency are all examples of data that is formatted according to cultural expectations. Because cultural preferences are bound to a geographic area, country is an important element of locale. Together these two elements (language and country) provide a precise context in which information can be presented. Locale presents information in the language and form that is best understood and appreciated by the local user.

Language

A locale's language is specified by the ISO 639 standard. The following table lists some commonly used language codes.

Language Code	Language
en	English
nl	Dutch
fr	French
es	Spanish
zh	Chinese
ja	Japanese
vi	Vietnamese

Because language is correlated with geography, a language code might not capture all the nuances of usage in a particular area. For example, French and Canadian French may use different phrases and terms to mean different things even though basic grammar and vocabulary are the same. Language is only one element of locale.

Country

The locale's country identifier is also specified by an ISO standard, ISO 3166, which describes valid two-letter codes for all countries. ISO 3166 defines these codes in uppercase letters. The following table lists some commonly used country codes.

Country Code	Country
US	United States
FR	France
IE	Ireland
CA	Canada
MX	Mexico

The country code provides more contextual information for a locale and affects a language's usage, word spelling, and collation rules.

Variant

A variant is an optional extension to a locale. It identifies a custom locale that is not possible to create with just language and country codes. Variants can be used by anyone to add additional context for identifying a locale. The locale en_US represents English (United States), but en_US_CA represents even more information and might identify a locale for English (California, U.S.A). Operating system or software vendors can use these variants to create more descriptive locales for their specific environments.

Using Double-Byte Character Sets on UNIX and Linux

The DataDirect Connect Series *for* ODBC UNIX and Linux drivers can use double-byte character sets. The drivers normally use the character set defined by the default locale "C" unless explicitly pointed to another character set. The default locale "C" corresponds to the 7-bit US-ASCII character set. Use the following procedure to set the locale to a different character set.

1 Add the following line at the beginning of applications that use double-byte character sets:

```
setlocale (LC_ALL, "");
```

This is a standard UNIX function. It selects the character set indicated by the environment variable LANG as the one to be used by X/Open compliant, character-handling functions. If this line is not present, or if LANG is not set or is set to NULL, the default locale "C" is used.

2 Set the LANG environment variable to the appropriate character set. The UNIX command `locale -a` can be used to display all supported character sets on your system.

For more information, refer to the man pages for "locale" and "setlocale."

Unicode Character Encoding

In addition to locale, the other major component of internationalizing software is the use of the Universal Codeset, or Unicode. Most developers know that Unicode is a standard encoding that can be used to support multilingual character sets. Unfortunately, understanding Unicode is not as simple as its

name would indicate. Software developers have used a number of character encodings, from ASCII to Unicode, to solve the many problems that arise when developing software applications that can be used worldwide.

Background

Most legacy computing environments have used ASCII character encoding developed by the ANSI standards body to store and manipulate character strings inside software applications. ASCII encoding was convenient for programmers because each ASCII character could be stored as a byte. The initial version of ASCII used only 7 of the 8 bits available in a byte, which meant that applications could use only 128 different characters. This version of ASCII could not account for European characters and was completely inadequate for Asian characters. Using the eighth bit to extend the total range of characters to 256 added support for most European characters. Today, ASCII refers to either the 7-bit or 8-bit encoding of characters.

As the need increased for applications with additional international support, ANSI again increased the functionality of ASCII by developing an extension to accommodate multilingual software. The extension, known as the Double-Byte Character Set (DBCS), allowed existing applications to function without change, but provided for the use of additional characters, including complex Asian characters. With DBCS, characters map to either one byte (for example, American ASCII characters) or two bytes (for example, Asian characters). The DBCS environment also introduced the concept of an operating system code page that identified how characters would be encoded into byte sequences in a particular computing environment. DBCS encoding provides a cross-platform mechanism for building multilingual applications; however, using variable-width codes is not ideal. (See "Using Double-Byte Character Sets on UNIX and Linux" on page 45 for details.)

Many developers felt that there was a better way to solve the problem. A group of leading software companies joined forces to form the Unicode Consortium. Together, they produced a new solution to building worldwide applications—Unicode. Unicode was originally designed as a fixed-width, uniform two-byte designation that could represent all modern scripts without the use of code pages. The Unicode Consortium has continued to evaluate new characters, and the current number of supported characters is over 95,000.

Although it seemed to be the perfect solution to building multilingual applications, Unicode started off with a significant drawback—it would have to be retrofitted into existing computing environments. To use the new paradigm, all applications would have to change. As a result, several standards-based transliterations were designed to convert two-byte fixed Unicode values into more appropriate character encodings, including, among others, UTF-8, UCS-2, and UTF-16.

UTF-8 is a standard method for transforming Unicode values into byte sequences that maintain transparency for all ASCII codes. UTF-8 is recognized by the Unicode Consortium as a mechanism for transforming Unicode values and is popular for use with HTML, XML, and other protocols. UTF-8 is, however, currently used primarily on AIX, HP-UX, Solaris, and Linux.

UCS-2 encoding is a fixed, two-byte encoding sequence and is a method for transforming Unicode values into byte sequences. It is the standard for Windows 95, Windows 98, Windows Me, and Windows NT.

UTF-16 is a superset of UCS-2, with the addition of some special characters in surrogate pairs. UTF-16 is the standard encoding for Windows 2000, Windows XP, Windows Server 2003, and Windows Vista.

For the DataDirect Connect Series *for* ODBC Unicode drivers, refer to specific driver chapters in the *DataDirect Connect Series for ODBC User's Guide* to determine which encodings are supported.

Unicode Support in Databases

Recently, database vendors have begun to support Unicode data types natively in their systems. With Unicode support, one database can hold multiple languages. For example, a large multinational corporation could store expense data in the local languages for the Japanese, U.S., English, German, and French offices in one database.

Not surprisingly, the implementation of Unicode data types varies from vendor to vendor. For example, the Microsoft SQL Server 2000 implementation of Unicode provides data in UTF-16 format, while Oracle provides Unicode data types in UTF-8 and UTF-16 formats. A consistent implementation of Unicode not only depends on the operating system, but also on the database itself.

Unicode Support in ODBC

Prior to the ODBC 3.5 standard, all ODBC access to function calls and string data types was through ANSI encoding (either ASCII or DBCS). Applications and drivers were both ANSI-based.

The ODBC 3.5 standard specified that the ODBC Driver Manager (on both Windows and UNIX) be capable of mapping both Unicode function calls and string data types to ANSI encoding as transparently as possible. This meant that ODBC 3.5-compliant Unicode applications could use Unicode function calls and string data types with ANSI drivers because the Driver Manager could convert them to ANSI. Because of character limitations in ANSI, however, not all conversions are possible.

The ODBC Driver Manager version 3.5 and later, therefore, supports the following configurations:

- ANSI application with an ANSI driver
- ANSI application with a Unicode driver
- Unicode application with a Unicode driver
- Unicode application with an ANSI driver

A Unicode application can work with an ANSI driver because the Driver Manager provides limited Unicode-to-ANSI mapping. The Driver Manager makes it possible for a pre-3.5 ANSI driver to work with a Unicode application. What distinguishes a Unicode driver from a non-Unicode driver is the Unicode driver's capacity to interpret Unicode function calls without the intervention of the Driver Manager, as described in the following section.

Unicode and Non-Unicode ODBC Drivers

The way in which a driver handles function calls from a Unicode application determines whether it is considered a "Unicode driver."

Function Calls

Instead of the standard ANSI SQL function calls, such as SQLConnect, Unicode applications use "W" (wide) function calls, such as SQLConnectW. If the driver is a true Unicode driver, it can understand "W" function calls and the Driver Manager can pass them through to the driver without conversion to ANSI. The DataDirect Connect Series *for* ODBC drivers that support "W" function calls are:

- DB2 Wire Protocol
- Greenplum Wire Protocol
- MySQL Wire Protocol

- Oracle Wire Protocol
- Oracle
- PostgreSQL Wire Protocol
- SQL Server Wire Protocol
- Sybase Wire Protocol
- Teradata
- XML

If the driver is a non-Unicode driver, it cannot understand W function calls, and the Driver Manager must convert them to ANSI calls before sending them to the driver. The Driver Manager determines the ANSI encoding system to which it must convert by referring to a code page. On Windows, this reference is to the Active Code Page. On UNIX and Linux, it is to the IANAAppCodePage connection string attribute, part of the odbc.ini file.

The following examples illustrate these conversion streams for the DataDirect Connect Series *for* ODBC drivers. The Driver Manager on UNIX and Linux prior to the DataDirect Connect Series *for* ODBC Release 5.0 assumes that Unicode applications and Unicode drivers use the same encoding (UTF-8). For the DataDirect Connect Series *for* ODBC Release 5.0 and higher on UNIX and Linux, the Driver Manager determines the type of Unicode encoding of both the application and the driver, and performs conversions when the application and driver use different types of encoding. This determination is made by checking two ODBC environment attributes: SQL_ATTR_APP_UNICODE_TYPE and SQL_ATTR_DRIVER_UNICODE_TYPE. "Default Unicode Mapping" on page 56 describes in detail how this is done.

Unicode Application with a Non-Unicode Driver

An operation involving a Unicode application and a non-Unicode driver incurs more overhead because function conversion is involved.

Windows

1. The Unicode application sends UCS-2/UTF-16 function calls to the Driver Manager.
2. The Driver Manager converts the function calls from UCS-2/UTF-16 to ANSI. The type of ANSI is determined by the Driver Manager through reference to the client machine's Active Code Page.
3. The Driver Manager sends the ANSI function calls to the non-Unicode driver.
4. The driver returns ANSI argument values to the Driver Manager.
5. The Driver Manager converts the function calls from ANSI to UCS-2/UTF-16 and returns these converted calls to the application.

UNIX and Linux: DataDirect Connect® Series for ODBC Releases Prior to 5.0

1. The Unicode application sends UTF-8 function calls to the Driver Manager.
2. The Driver Manager converts the function calls from UTF-8 to ANSI. The type of ANSI is determined by the Driver Manager through reference to the client machine's value for the IANAAppCodePage attribute.
3. The Driver Manager sends the converted ANSI function calls to the non-Unicode driver.
4. The driver returns ANSI argument values to the Driver Manager.
5. The Driver Manager converts the function calls from ANSI to UTF-8 and returns these converted calls to the application.

UNIX and Linux: DataDirect Connect® Series for ODBC 5.0 and Higher

1. The Unicode application sends function calls to the Driver Manager. The Driver Manager expects these function calls to be UTF-8 or UTF-16 based on the value of the SQL_ATTR_APP_UNICODE_TYPE attribute.
2. The Driver Manager converts the function calls from UTF-8 or UTF-16 to ANSI. The type of ANSI is determined by the Driver Manager through reference to the client machine's value for the IANAAppCodePage attribute.
3. The Driver Manager sends the converted ANSI function calls to the non-Unicode driver.
4. The driver returns ANSI argument values to the Driver Manager.
5. The Driver Manager converts the function calls from ANSI to UTF-8 or UTF-16 and returns these converted calls to the application.

Unicode Application with a Unicode Driver

An operation involving a Unicode application and a Unicode driver that use the same Unicode encoding is efficient because no function conversion is involved. If the application and the driver each use different types of encoding, there is some conversion overhead. See "Default Unicode Mapping" on page 56 for details.

Windows

1. The Unicode application sends UCS-2/UTF-16 function calls to the Driver Manager.
2. The Driver Manager does not have to convert the UCS-2/UTF-16 function calls to ANSI. It passes the Unicode function call to the Unicode driver.

3 The driver returns UCS-2/UTF-16 argument values to the Driver Manager.

4 The Driver Manager returns UCS-2/UTF-16 function calls to the application.

UNIX and Linux: DataDirect Connect® Series for ODBC Releases Prior to 5.0

1 The Unicode application sends UTF-8 function calls to the Driver Manager.

2 The Driver Manager does not have to convert the UTF-8 function calls to ANSI. It passes the Unicode function call with UTF-8 arguments to the Unicode driver.

3 The driver returns UTF-8 argument values to the Driver Manager.

4 The Driver Manager returns UTF-8 function calls to the application.

UNIX and Linux: DataDirect Connect® Series for ODBC 5.0 and Higher

1 The Unicode application sends function calls to the Driver Manager. The Driver Manager expects these function calls to be UTF-8 or UTF-16 based on the value of the SQL_ATTR_APP_UNICODE_TYPE attribute.

2 The Driver Manager passes Unicode function calls to the Unicode driver. The Driver Manager has to perform function call conversions if the SQL_ATTR_APP_UNICODE_TYPE is different from the SQL_ATTR_DRIVER_UNICODE_TYPE.

3 The driver returns argument values to the Driver Manager. Whether these are UTF-8 or UTF-16 argument values is based on the value of the SQL_ATTR_DRIVER_UNICODE_TYPE attribute.

4 The Driver Manager returns appropriate function calls to the application based on the SQL_ATTR_APP_UNICODE_TYPE attribute value. The Driver Manager has to perform function call conversions if the SQL_ATTR_DRIVER_UNICODE_TYPE value is different from the SQL_ATTR_APP_UNICODE_TYPE value.

Data

ODBC C data types are used to indicate the type of C buffers that store data in the application. This is in contrast to SQL data types, which are mapped to native database types to store data in a database (data store). ANSI applications bind to the C data type SQL_C_CHAR and expect to receive information bound in the same way. Similarly, most Unicode applications bind to the C data type SQL_C_WCHAR (wide data type) and expect to receive information bound in the same way. Any ODBC 3.5-compliant Unicode driver must be capable of supporting SQL_C_CHAR and SQL_C_WCHAR so that it can return data to both ANSI and Unicode applications.

When the driver communicates with the database, it must use ODBC SQL data types, such as SQL_CHAR and SQL_WCHAR, that map to native database types. In the case of ANSI data and an ANSI database, the driver receives data bound to SQL_C_CHAR and passes it to the database as SQL_CHAR. The same is true of SQL_C_WCHAR and SQL_WCHAR in the case of Unicode data and a Unicode database.

When data from the application and the data stored in the database differ in format, for example, ANSI application data and Unicode database data, conversions must be performed. The driver cannot receive SQL_C_CHAR data and pass it to a Unicode database that expects to receive a SQL_WCHAR data type. The driver or the Driver Manager must be capable of converting SQL_C_CHAR to SQL_WCHAR, and vice versa.

The simplest cases of data communication are when the application, the driver, and the database are all of the same type and encoding, ANSI-to-ANSI-to-ANSI or Unicode-to-Unicode-to-Unicode. There is no data conversion involved in these instances.

When there is a difference in data types, it must be converted from one type to another at the driver or Driver Manager level, which involves additional overhead. The type of driver determines whether these conversions are performed by the driver or the Driver Manager. "The Driver Manager and Unicode Encoding on UNIX and Linux" on page 58 describes how the Driver Manager determines the type of Unicode encoding of the application and driver.

The following sections discuss two basic types of data conversion in the DataDirect Connect Series *for* ODBC drivers and the Driver Manager. How an individual driver exchanges different types of data with a particular database at the database level is beyond the scope of this discussion.

Unicode Driver

The Unicode driver, not the Driver Manager, must convert SQL_C_CHAR (ANSI) data to SQL_WCHAR (Unicode) data, and vice versa, as well as SQL_C_WCHAR (Unicode) data to SQL_CHAR (ANSI) data, and vice versa.

The driver must use client code page information (Active Code Page on Windows and IANAAppCodePage attribute on UNIX/Linux) to determine which ANSI code page to use for the conversions. The Active Code Page or IANAAppCodePage must match the database default character encoding; if it does not, conversion errors are possible.

ANSI Driver

The Driver Manager, not the ANSI driver, must convert SQL_C_WCHAR (Unicode) data to SQL_CHAR (ANSI) data, and vice versa (see "Unicode Support in ODBC" on page 48 for a detailed discussion). This is necessary because ANSI drivers do not support any Unicode ODBC types.

The Driver Manager must use client code page information (Active Code Page on Windows and the IANAAppCodePage attribute on UNIX/Linux) to determine which ANSI code page to use for the conversions. The Active Code Page or IANAAppCodePage must match the database default character encoding. If not, conversion errors are possible.

Default Unicode Mapping

The default Unicode mapping for an application's SQL_C_WCHAR variable is:

Platform	Default Unicode Mapping
Windows	UCS-2/UTF-16
AIX	UTF-8
HP-UX	UTF-8
Solaris	UTF-8
Linux	UTF-8

Connection Attribute for Unicode

If you do not want to use the default Unicode mappings for SQL_C_WCHAR, a connection attribute is available to override the default mappings. This attribute determines how character data is converted and presented to an application and the database.

Attribute	Description
SQL_ATTR_APP_WCHAR_TYPE (1061)	Sets the SQL_C_WCHAR type for parameter and column binding to the Unicode type, either SQL_DD_CP_UTF16 (default for Windows) or SQL_DD_CP_UTF8 (default for UNIX/Linux).

You can set this attribute before or after you connect. After this attribute is set, all conversions are made based on the character set specified.

For example:

```
rc = SQLSetConnectAttr (hdbc, SQL_ATTR_APP_WCHAR_TYPE,
(void *)SQL_DD_CP_UTF16, SQL_IS_INTEGER);
```

SQLGetConnectAttr and SQLSetConnectAttr for the SQL_ATTR_APP_WCHAR_TYPE attribute return a SQL State of HYC00 for drivers that do not support Unicode.

This connection attribute and its valid values can be found in the file qesqlext.h, which is installed with the product.

NOTE: For the SQL Server Wire Protocol driver, this attribute is supported only on UNIX and Linux, not on Windows.

The Driver Manager and Unicode Encoding on UNIX and Linux

Unicode ODBC drivers on UNIX and Linux can use UTF-8 or UTF-16 encoding. This would normally mean that a UTF-8 application could not work with a UTF-16 driver, and, conversely, that a UTF-16 application could not work with a UTF-8 driver. To accomplish the goal of being able to use a single UTF-8 or UTF-16 application with either a UTF-8 or UTF-16 driver, the Driver Manager must be able to determine with which type of encoding the application and driver use and, if necessary, convert them accordingly.

To make this determination, the Driver Manager supports two ODBC environment attributes: SQL_ATTR_APP_UNICODE_TYPE and SQL_ATTR_DRIVER_UNICODE_TYPE, each with possible values of SQL_DD_CP_UTF8 and SQL_DD_CP_UTF16. The default value is SQL_DD_CP_UTF8.

The Driver Manager undertakes the following steps before actually connecting to the driver.

1. Determine the application Unicode type: Applications that use UTF-16 encoding for their string types need to set SQL_ATTR_APP_UNICODE_TYPE accordingly before connecting to any driver. When the Driver Manager reads this attribute, it expects all string arguments to the ODBC "W" functions to be in the specified Unicode format. This attribute also indicates how the SQL_C_WCHAR buffers must be encoded.

2 Determine the driver Unicode type: The Driver Manager must determine through which Unicode encoding the driver supports its "W" functions. This is done as follows:

   a SQLGetEnvAttr(SQL_ATTR_DRIVER_UNICODE_TYPE) is called in the driver by the Driver Manager. The driver, if capable, returns either SQL_DD_CP_UTF16 or SQL_DD_CP_UTF8 to indicate to the Driver Manager which encoding it expects.

   b If the preceding call to SQLGetEnvAttr fails, the Driver Manager looks either in the Data Source section of the odbc.ini specified by the connection string or in the connection string itself for a connection option named DriverUnicodeType. Valid values for this option are 1 (UTF-16) or 2 (UTF-8). The Driver Manager assumes that the Unicode encoding of the driver corresponds to the value specified.

   c If neither of the preceding attempts are successful, the Driver Manager assumes that the Unicode encoding of the driver is UTF-8.

3 Determine if the driver supports SQL_ATTR_WCHAR_TYPE: SQLSetConnectAttr (SQL_ATTR_WCHAR_TYPE, *x*) is called in the driver by the Driver Manager, where *x* is either SQL_DD_CP_UTF8 or SQL_DD_CP_UTF16, depending on the value of the SQL_ATTR_APP_UNICODE_TYPE environment setting. If the driver returns any error on this call to SQLSetConnectAttr, the Driver Manager assumes that the driver does not support this connection attribute.

   In an error occurs, the Driver Manager returns a warning. The Driver Manager does not convert all bound parameter data from the application Unicode type to the driver Unicode type specified by SQL_ATTR_DRIVER_UNICODE_TYPE. Neither does it convert all data bound as SQL_C_WCHAR to the application Unicode type specified by SQL_ATTR_APP_UNICODE_TYPE.

Based on the information it has gathered prior to connection, the Driver Manager either does not have to convert function calls, or, before calling the driver, it converts to either UTF-8 or UTF-16 all string arguments to calls to the ODBC "W" functions.

References:

Java Internationalization: An Overview, John O'Connor, http://java.sun.com/developer/technicalArticles/Intl/IntlIntro/

Unicode Support in the Solaris Operating Environment, May 2000, Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, CA 94303-4900

5 Designing ODBC Applications for Performance Optimization

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 chapter 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 the following table Table 5-1.

Table 5-1. Common Performance Problems Using ODBC Applications

Problem	Solution	See guidelines in...
Network communication is slow.	Reduce network traffic.	"Using Catalog Functions" on page 62
The process of evaluating complex SQL queries on the database server is slow and can reduce concurrency.	Simplify queries.	"Using Catalog Functions" on page 62 "Selecting ODBC Functions" on page 71

Table 5-1. 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 66 "Selecting ODBC Functions" on page 71
Disk I/O is slow.	Limit disk input/output.	"Managing Connections and Updates" on page 75

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 owned by the user, the second owned by the sales department, and the third 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.

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_PTR 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_PTR 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_PTR,
   &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 I/O 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.

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 round trips 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 I/O 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 I/O 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 I/O 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 round trips 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 Positioned Updates and Deletes

Use positioned updates and deletes or SQLSetPos to update data. Although positioned updates do not apply to all types of applications, developers should use positioned updates and deletes when it makes sense. positioned 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, positioned 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 positioned 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.

6 Using Indexes

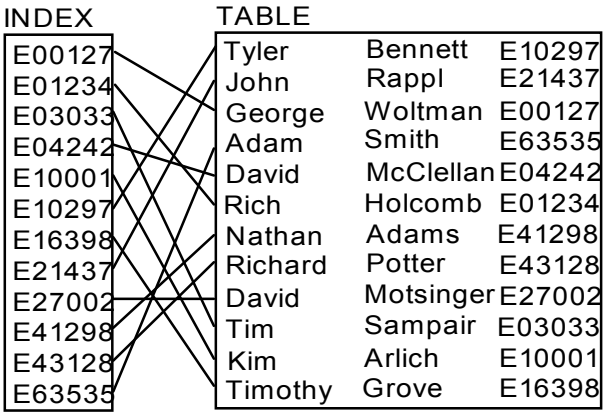
This chapter discusses the ways in which you can improve the performance of database activity using indexes. It provides general guidelines that apply to most databases. Consult your database vendor's documentation for more detailed information.

For information regarding how to create and drop indexes, refer to the appropriate database driver chapter for flat-file drivers or your database system documentation for relational drivers. This chapter includes the following topics:

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 rows that contain that value.



A database driver can use indexes to find rows 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 rows having an employee ID of E10001. By using an index on the emp_id field, however, the driver can quickly find those rows.

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 Row 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, for example, 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 most common operations on a table are:

- Inserting, updating, and deleting rows
- Retrieving rows

If you most often insert, update, and delete rows, 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 row inserts, updates, and deletes. It may be more efficient to drop all indexes before modifying a large number of rows, and re-create the indexes after the modifications.

If you most often retrieve rows, you must look further to define the criteria for retrieving rows and create indexes to improve the performance of these retrievals. Assume you have an employee database table and you will retrieve rows 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 83 for details).

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

- If your row retrievals are based on only one field at a time (for example, dept='D101'), create an index on these fields.
- If your row 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 rows 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 `dept_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.

7 Locking and Isolation Levels

This chapter discusses locking and isolation levels and how their settings can affect the data you retrieve. Different database systems support different locking and isolation levels.

NOTE: Refer to the section "Isolation and Lock Levels Supported" in the appropriate driver chapter in the *DataDirect Connect Series for ODBC User's Guide* for database-specific locking and isolation level information.

This chapter includes the following topics:

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 they use, 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 7-1 shows what data consistency behaviors can occur at each isolation level.

Table 7-1. 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.

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 use 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.

8 SSL Encryption Cipher Suites

The following tables list the SSL encryption cipher suites supported by the DataDirect Connect Series *for* ODBC drivers.

Refer to the section "Using Security" on page 99 in Chapter 3 "Advanced Features" of the *DataDirect Connect Series for ODBC User's Guide* for more information about SSL data encryption.

Table 8-1 shows the Encryption Cipher suite used by the driver if it cannot negotiate either SSL3 or TLS1 with the server.

Table 8-1. SSL Encryption Cipher Suite

DHE-RSA-AES256-SHA
DHE-DSS-AES256-SHA
AES256-SHA
EDH-RSA-DES-CBC3-SHA
EDH-DSS-DES-CBC3-SHA
DES-CBC3-SHA
DES-CBC3-MD5
DHE-RSA-AES128-SHA
DHE-DSS-AES128-SHA
AES128-SHA
RC2-CBC-MD5
RC4-SHA
RC4-MD5
EDH-RSA-DES-CBC-SHA
EDH-DSS-DES-CBC-SHA
DES-CBC-SHA
DES-CBC-MD5

Table 8-1. SSL Encryption Cipher Suite (cont.)
EXP-EDH-RSA-DES-CBC-SHA
EXP-EDH-DSS-DES-CBC-SHA
EXP-DES-CBC-SHA
EXP-RC2-CBC-MD5
EXP-RC4-MD5

Table 8-2 shows the SSL3 Encryption Cipher suite used by the driver if it can negotiate SSL3 with the server.

Table 8-2. Encryption Cipher Suite SSL3
DHE-RSA-AES256-SHA
DHE-DSS-AES256-SHA
AES256-SHA
EDH-RSA-DES-CBC3-SHA
EDH-DSS-DES-CBC3-SHA
DES-CBC3-SHA
DHE-RSA-AES128-SHA
DHE-DSS-AES128-SHA
AES128-SHA
RC4-SHA
RC4-MD5
EDH-RSA-DES-CBC-SHA
EDH-DSS-DES-CBC-SHA
DES-CBC-SHA
EXP-EDH-RSA-DES-CBC-SHA
EXP-EDH-DSS-DES-CBC-SHA
EXP-DES-CBC-SHA

Table 8-2. Encryption Cipher Suite SSL3 (cont.)
EXP-RC2-CBC-MD5
EXP-RC4-MD5

Table 8-3 shows the TLS1 Encryption Cipher suite used by the driver if it can negotiate TLS1 with the server.

Table 8-3. Encryption Cipher Suite TLS1
DHE-RSA-AES256-SHA
DHE-DSS-AES256-SHA
AES256-SHA
EDH-RSA-DES-CBC3-SHA
EDH-DSS-DES-CBC3-SHA
DES-CBC3-SHA
DHE-RSA-AES128-SHA
DHE-DSS-AES128-SHA
AES128-SHA
RC4-SHA
RC4-MD5
EDH-RSA-DES-CBC-SHA
EDH-DSS-DES-CBC-SHA
DES-CBC-SHA
EXP-EDH-RSA-DES-CBC-SHA
EXP-EDH-DSS-DES-CBC-SHA
EXP-DES-CBC-SHA
EXP-RC2-CBC-MD5
EXP-RC4-MD5

9 DataDirect Bulk Load

This chapter contains detailed information about the functions and statement attributes associated with DataDirect Bulk Load. This chapter includes the following topics:

- "DataDirect Bulk Load Functions"
- "DataDirect Bulk Load Statement Attributes"

For a full discussion of the features and operation of DataDirect Bulk Load, refer to "Using DataDirect Bulk Load" in Chapter 3 of the *DataDirect Connect Series for ODBC User's Guide*.

DataDirect Bulk Load Functions

The following DataDirect functions and parameters are not part of the standard ODBC API. They include functions for returning errors and warnings on bulk operations as well as functions for bulk export, loading, and verification.

NOTE: For your application to use DataDirect Bulk Load functionality, it must obtain driver connection handles and function pointers, as follows:

1. Use SQLGetInfo with the parameter SQL_DRIVER_HDBC to obtain the driver's connection handle from the Driver Manager.
2. Use SQLGetInfo with the parameter SQL_DRIVER_HLIB to obtain the driver's shared library or DLL handle from the Driver Manager.

3 Obtain function pointers to the bulk load functions using the function name resolution method specific to your operating system. The bulk.c source file shipped with the drivers contains the function resolveName that illustrates how to obtain function pointers to the bulk load functions.

All of this is detailed in the code examples shown in the following sections. All of these functions can be found in the commented bulk.c source file that ships with the drivers. This file is located in the \example\bulk subdirectory of the product installation directory along with a text file named bulk.txt. Please consult bulk.txt for instructions about the bulk.c file.

Utility Functions

The example code in this section shows utility functions to which the DataDirect functions for bulk exporting, verification, and bulk loading refer, as well as the DataDirect functions GetBulkDiagRec and GetBulkDiagRecW.

GetBulkDiagRec and GetBulkDiagRecW

Syntax

```
SQLReturn
GetBulkDiagRec     (SQLSMALLINT  HandleType,
                    SQLHANDLE    Handle,
                    SQLSMALLINT  RecNumber,
                    SQLCHAR*     Sqlstate,
                    SQLINTEGER*  NativeError,
                    SQLCHAR*     MessageText,
                    SQLSMALLINT  BufferLength,
                    SQLSMALLINT* TextLength);
```

```
GetBulkDiagRecW    (SQLSMALLINT  HandleType,
                    SQLHANDLE     Handle,
                    SQLSMALLINT   RecNumber,
                    SQLWCHAR*     Sqlstate,
                    SQLINTEGER*   NativeError,
                    SQLWCHAR*     MessageText,
                    SQLSMALLINT   BufferLength,
                    SQLSMALLINT*  TextLength);
```

The standard ODBC return codes are returned: SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_INVALID_HANDLE, SQL_NO_DATA, and SQL_ERROR.

Description

GetBulkDiagRec (ANSI application) and GetBulkDiagRecW (Unicode application) return errors and warnings generated by bulk operations. The argument definition, return values, and function behavior is the same as for the standard ODBC SQLGetDiagRec and SQLGetDiagRecW functions with the following exceptions:

- The GetBulkDiagRec and GetBulkDiagRecW functions can be called after a bulk load, export or validate function is invoked to retrieve any error messages generated by the bulk operation. Calling these functions after any function except a bulk function is not recommended.
- The values returned in the Sqlstate and MessageText buffers by the GetBulkDiagRecW function are encoded as UTF-16 on Windows platforms. On UNIX and Linux platforms, the values returned for Sqlstate and MessageText are UTF-16 if the value of the SQL_ATTR_APP_UNICODE_TYPE is SQL_DD_CP_UTF16 and UTF-8 if the value of SQL_ATTR_APP_UNICODE_TYPE is SQL_DD_CP_UTF8.
- The handle passed as the Handle argument must be a driver connection handle obtained by calling SQLGetInfo (<ODBC Conn Handle>, SQL_DRIVER_HDBC).
- SQL_HANDLE_DBC is the only value accepted for HandleType. Any other value causes an error to be returned.

Example

```
#include "qesqlext.h"

#ifndef NULL
#define NULL 0
#endif

#if (! defined (_WIN32)) && (! defined (_WIN64))
typedef void * HMODULE;
#endif

/* Get the address of a routine in a shared library or DLL. */
void * resolveName (
  HMODULE hmod,
  const char *name)
{
#if defined (_WIN32) || defined (_WIN64)

  return GetProcAddress (hmod, name);
#elif defined (hpux)
  void *routine = shl_findsym (hmod, name);

  shl_findsym (hmod, name, TYPE_PROCEDURE, &routine);

  return routine;
#else
  return dlsym (hmod, name);
#endif
}
```

```
/* Get errors directly from the driver's connection handle. */
void driverError (void *driverHandle, HMODULE hmod)
{
  UCHAR sqlstate[16];
  UCHAR errmsg[SQL_MAX_MESSAGE_LENGTH * 2];
  SDWORD nativeerr;
  SWORD actualmsglen;
  RETCODE rc;
  SQLSMALLINT  i;
  PGetBulkDiagRec getBulkDiagRec;

  getBulkDiagRec = (PGetBulkDiagRec)
    resolveName (hmod, "GetBulkDiagRec");

  if (! getBulkDiagRec) {
    printf ("Cannot find GetBulkDiagRec!\n");
    return;
  }


  i = 1;
loop:   rc = (*getBulkDiagRec) (SQL_HANDLE_DBC,
          driverHandle, i++,
          sqlstate, &nativeerr, errmsg,
          SQL_MAX_MESSAGE_LENGTH - 1, &actualmsglen);

  if (rc == SQL_ERROR) {
    printf ("GetBulkDiagRec failed!\n");
    return;
  }

  if (rc == SQL_NO_DATA_FOUND) return;

  printf ("SQLSTATE = %s\n", sqlstate);
  printf ("NATIVE ERROR = %d\n", nativeerr);
  errmsg[actualmsglen] = '\0';
  printf ("MSG = %s\n\n", errmsg);
  goto loop;
}
```

Export, Validate, and Load Functions

The example code in this section shows the DataDirect functions for bulk exporting, verification, and bulk loading.

ExportTableToFile and ExportTableToFileW

Syntax

```
SQLReturn
ExportTableToFile  (HDBC      hdbc,
                    SQLCHAR*  TableName,
                    SQLCHAR*  FileName,
                    SQLLEN    IANAAppCodePage,
                    SQLLEN    ErrorTolerance,
                    SQLLEN    WarningTolerance,
                    SQLCHAR*  LogFile)

ExportTableToFileW (HDBC      hdbc,
                    SQLWCHAR* TableName,
                    SQLWCHAR* FileName,
                    SQLLEN    IANAAppCodePage,
                    SQLLEN    ErrorTolerance,
                    SQLLEN    WarningTolerance,
                    SQLWCHAR* LogFile)
```

The standard ODBC return codes are returned: SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_INVALID_HANDLE, and SQL_ERROR.

Description

ExportTableToFile (ANSI application) and ExportTableToFileW (Unicode application) bulk export a table to a physical file. Both a bulk data file and a bulk configuration file are produced by this operation. The configuration file has the same name as the data file, but with an XML extension. The bulk export operation can create a log file and can also export to external files. Refer to "External Overflow Files" in Chapter 3 of the *DataDirect Connect Series for ODBC User's Guide* for more information. The export

operation can be configured such that if any errors or warnings occur:

- The operation always completes
- The operation always terminates
- The operation terminates after a certain threshold of warnings or errors is exceeded.

Parameters

hdbc is the driver's connection handle, which is not the handle returned by SQLAllocHandle or SQLAllocConnect. To obtain the driver's connection handle, the application must then use the standard ODBC function SQLGetInfo (*ODBC Conn Handle*, *SQL_DRIVER_HDBC*).

TableName is a null-terminated string that specifies the name of the source database table that contains the data to be exported.

FileName is a null-terminated string that specifies the path (relative or absolute) and file name of the bulk load data file to which the data is to be exported. It also specifies the file name of the bulk configuration file. This file must not already exist. If the file already exists, an error is returned.

IANAAppCodePage specifies the code page value to which the driver must convert all data for storage in the bulk data file. Refer to "Character Set Conversions" in Chapter 3 of the *DataDirect Connect Series for ODBC User's Guide* for more information.

The default value on Windows is the current code page of the machine. On UNIX/Linux, the default value is 4.

ErrorTolerance specifies the number of errors to tolerate before an operation terminates. A value of 0 indicates that no errors are tolerated; the operation fails when the first error is encountered.

The default of -1 means that an infinite number of errors is tolerated.

WarningTolerance specifies the number of warnings to tolerate before an operation terminates. A value of 0 indicates that no warnings are tolerated; the operation fails when the first warning is encountered.

The default of -1 means that an infinite number of warnings is tolerated.

LogFile is a null-terminated character string that specifies the path (relative or absolute) and file name of the bulk log file. Events logged to this file are:

- Total number of rows fetched
- A message for each row that failed to export
- Total number of rows that failed to export
- Total number of rows successfully exported

Information about the load is written to this file, preceded by a header. Information about the next load is appended to the end of the file.

If LogFile is NULL, no log file is created.

Example

```
HDBC      hdbc;
HENV      henv;
void      *driverHandle;
HMODULE      hmod;
PExportTableToFile exportTableToFile;

char      tableName[128];
char      fileName[512];
char      logFile[512];
int       errorTolerance;
int       warningTolerance;
int       codePage;
```

```
/* Get the driver's connection handle from the DM.
   This handle must be used when calling directly into the driver. */

rc = SQLGetInfo (hdbc, SQL_DRIVER_HDBC, &driverHandle, 0, NULL);
if (rc != SQL_SUCCESS) {
    ODBC_error (henv, hdbc, SQL_NULL_HSTMT);
    EnvClose (henv, hdbc);
    exit (255);
}

/* Get the DM's shared library or DLL handle to the driver. */

rc = SQLGetInfo (hdbc, SQL_DRIVER_HLIB, &hmod, 0, NULL);
if (rc != SQL_SUCCESS) {
    ODBC_error (henv, hdbc, SQL_NULL_HSTMT);
    EnvClose (henv, hdbc);
    exit (255);
}

exportTableToFile = (PExportTableToFile)
    resolveName (hmod, "ExportTableToFile");
if (! exportTableToFile) {
    printf ("Cannot find ExportTableToFile!\n");
    exit (255);
}

rc = (*exportTableToFile) (
      driverHandle,
      (const SQLCHAR *) tableName,
      (const SQLCHAR *) fileName,
      codePage,
      errorTolerance, warningTolerance,
      (const SQLCHAR *) logFile);
if (rc == SQL_SUCCESS) {
      printf ("Export succeeded.\n");
}
else {
      driverError (driverHandle, hmod);
}
```

ValidateTableFromFile and ValidateTableFromFileW

Syntax

```
SQLReturn
ValidateTableFromFile  (HDBC      hdbc,
                        SQLCHAR*  TableName,
                        SQLCHAR*  ConfigFile,
                        SQLCHAR*  MessageList,
                        SQLULEN   MessageListSize,
                        SQLULEN*  NumMessages)

ValidateTableFromFileW (HDBC      hdbc,
                        SQLCHAR*  TableName,
                        SQLCHAR*  ConfigFile,
                        SQLCHAR*  MessageList,
                        SQLULEN   MessageListSize,
                        SQLULEN*  NumMessages)
```

The standard ODBC return codes are returned: SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_INVALID_HANDLE, and SQL_ERROR.

Description

ValidateTableFromFile (ANSI application) and ValidateTablefromFileW (Unicode application) verify the metadata in the configuration file against the data structure of the target database table. Refer to "Verification of the Bulk Load Configuration File" in Chapter 3 of the *DataDirect Connect Series for ODBC User's Guide* for more detailed information.

Parameters

hdbc is the driver's connection handle, which is not the handle returned by SQLAllocHandle or SQLAllocConnect. To obtain the driver's connection handle, the application must then use the standard ODBC function SQLGetInfo (*ODBC Conn Handle*, *SQL_DRIVER_HDBC*).

TableName is a null-terminated character string that specifies the name of the target database table into which the data is to be loaded.

ConfigFile is a null-terminated character string that specifies the path (relative or absolute) and file name of the bulk configuration file.

MessageList specifies a pointer to a buffer used to record any of the errors and warnings. MessageList must not be null.

MessageListSize specifies the maximum number of characters that can be written to the buffer to which MessageList points. If the buffer to which MessageList points is not big enough to hold all of the messages generated by the validation process, the validation is aborted and SQL_ERROR is returned.

NumMessages contains the number of messages that were added to the buffer. This method reports the following criteria:

- Check data types - Each column data type is checked to ensure no loss of data occurs. If a data type mismatch is detected, the driver adds an entry to the MessageList in the following format: `Risk of data conversion loss: Destination column_number is of type x, and source column_number is of type y.`
- Check column sizes - Each column is checked for appropriate size. If column sizes are too small in destination tables, the driver adds an entry to the MessageList in the following format: `Possible Data Truncation: Destination column_number is of size x while source column_number is of size y.`
- Check codepages - Each column is checked for appropriate code page alignment between the source and destination. If a mismatch occurs, the driver adds an entry to the MessageList in the following format: `Destination column code page for column_number risks data corruption if transposed without correct character conversion from source column_number.`

- Check Config Col Info - The destination metadata and the column metadata in the configuration file are checked for consistency of items such as length for character and binary data types, the character encoding code page for character types, precision and scale for numeric types, and nullablity for all types. If any inconsistency is found, the driver adds an entry to the MessageList in the following format: `Destination column metadata for column_number has column info mismatches from source column_number`.
- Check Column Names and Mapping - The columns defined in the configuration file are compared to the destination table columns based on the order of the columns. If the number of columns in the configuration file and/or import file does not match the number of columns in the table, the driver adds an entry to the MessageList in the following format: `The number of destination columns number does not match the number of source columns number`.

The function returns an array of null-terminated strings in the buffer to which MessageList points with an entry for each of these checks. If the driver determines that the information in the bulk load configuration file matches the metadata of the destination table, a return code of SQL_SUCCESS is returned and the MessageList remains empty.

If the driver determines that there are minor differences in the information in the bulk load configuration file and the destination table, then SQL_SUCCESS_WITH_INFO is returned and the MessageList is populated with information on the cause of the potential problems.

If the driver determines that the information in the bulk load information file cannot successfully be loaded into the destination table, then a return code of SQL_ERROR is returned and the MessageList is populated with information on the problems and mismatches between the source and destination.

Example

```
HDBC      hdbc;
HENV      henv;
void      *driverHandle;
HMODULE   hmod;
PValidateTableFromFile validateTableFromFile;

char      tableName[128];
char      configFile[512];
char      messageList[10240];
SQLLEN    numMessages;

/* Get the driver's connection handle from the DM.
   This handle must be used when calling directly into the driver. */

rc = SQLGetInfo (hdbc, SQL_DRIVER_HDBC, &driverHandle, 0, NULL);
if (rc != SQL_SUCCESS) {
    ODBC_error (henv, hdbc, SQL_NULL_HSTMT);
    EnvClose (henv, hdbc);
    exit (255);
}

/* Get the DM's shared library or DLL handle to the driver. */

rc = SQLGetInfo (hdbc, SQL_DRIVER_HLIB, &hmod, 0, NULL);
if (rc != SQL_SUCCESS) {
    ODBC_error (henv, hdbc, SQL_NULL_HSTMT);
    EnvClose (henv, hdbc);
    exit (255);
}

validateTableFromFile = (PValidateTableFromFile)
  resolveName (hmod, "ValidateTableFromFile");
if (!validateTableFromFile) {
  printf ("Cannot find ValidateTableFromFile!\n");
  exit (255);
}
```

```
messageList[0] = 0;
numMessages = 0;

rc = (*validateTableFromFile) (
      driverHandle,
      (const SQLCHAR *) tableName,
      (const SQLCHAR *) configFile,
      (SQLCHAR *) messageList,
      sizeof (messageList),
      &numMessages);
printf ("%d message%s%s\n", numMessages,
       (numMessages == 0) ? "s" :
       ((numMessages == 1) ? " : " : "s : "),
       (numMessages > 0) ? messageList : "");
if (rc == SQL_SUCCESS) {
       printf ("Validate succeeded.\n");
}
else {
    driverError (driverHandle, hmod);
}
```

LoadTableFromFile and LoadTableFromFileW

Syntax

```
SQLReturn
LoadTableFromFile  (HDBC      hdbc,
                    SQLCHAR*  TableName,
                    SQLCHAR*  FileName,
                    SQLLEN    ErrorTolerance,
                    SQLLEN    WarningTolerance,
                    SQLCHAR*  ConfigFile,
                    SQLCHAR*  LogFile,
                    SQLCHAR*  DiscardFile,
                    SQLULEN   LoadStart,
                    SQLULEN   LoadCount,
                    SQLULEN   ReadBufferSize)
```

```
LoadTableFromFileW (HDBC      hdbc,
                    SQLWCHAR*  TableName,
                    SQLWCHAR*  FileName,
                    SQLLEN     ErrorTolerance,
                    SQLLEN     WarningTolerance,
                    SQLWCHAR*  ConfigFile,
                    SQLWCHAR*  LogFile,
                    SQLWCHAR*  DiscardFile,
                    SQLULEN    LoadStart,
                    SQLULEN    LoadCount,
                    SQLULEN    ReadBufferSize)
```

The standard ODBC return codes are returned: SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_INVALID_HANDLE, and SQL_ERROR.

Description

LoadTableFromFile (ANSI application) and LoadTablefromFileW (Unicode application) bulk load data from a file to a table. The load operation can create a log file and can also create a discard file that contains rows rejected during the load. The discard file is in the same format as the bulk load data file. After fixing reported issues in the discard file, the bulk load can be reissued using the discard file as the bulk load data file.

The load operation can be configured such that if any errors or warnings occur:

- The operation always completes
- The operation always terminates
- The operation terminates after a certain threshold of warnings or errors is exceeded.

If a load fails, the *LoadStart* and *LoadCount* parameters can be used to control which rows are loaded when a load is restarted after a failure.

Parameters

hdbc is the driver's connection handle, which is not the handle returned by SQLAllocHandle or SQLAllocConnect. To obtain the driver's connection handle, the application must then use the standard ODBC function SQLGetInfo (*ODBC Conn Handle*, *SQL_DRIVER_HDBC*).

TableName is a null-terminated character string that specifies the name of the target database table into which the data is to be loaded.

FileName is a null-terminated string that specifies the path (relative or absolute) and file name of the bulk data file from which the data is to be loaded.

ErrorTolerance specifies the number of errors to tolerate before an operation terminates. A value of 0 indicates that no errors are tolerated; the operation fails when the first error is encountered.

The default of -1 means that an infinite number of errors is tolerated.

WarningTolerance specifies the number of warnings to tolerate before an operation terminates. A value of 0 indicates that no warnings are tolerated; the operation fails when the first warning is encountered.

The default of -1 means that an infinite number of warnings is tolerated.

ConfigFile is a null-terminated character string that specifies the path (relative or absolute) and file name of the bulk configuration file.

LogFile is a null-terminated character string that specifies the path (relative or absolute) and file name of the bulk log file. Events logged to this file are:

- Total number of rows read
- Message for each row that failed to load.
- Total number of rows that failed to load
- Total number of rows successfully loaded

Information about the load is written to this file, preceded by a header. Information about the next load is appended to the end of the file.

If LogFile is NULL, no log file is created.

DiscardFile is a null-terminated character string that specifies the path (relative or absolute) and file name of the bulk discard file. Any row that cannot be inserted into database as result of bulk load is added to this file, with the last row to be rejected added to the end of the file.

Information about the load is written to this file, preceded by a header. Information about the next load is appended to the end of the file.

If DiscardFile is NULL, no discard file is created.

LoadStart specifies the first row to be loaded from the data file. Rows are numbered starting with 1. For example, when LoadStart=10, the first 9 rows of the file are skipped and the first row loaded is row 10. This parameter can be used to restart a load after a failure.

LoadCount specifies the number of rows to be loaded from the data file. The bulk load operation loads rows up to the value of LoadCount from the file to the database. It is valid for LoadCount to specify more rows than exist in the data file. The bulk load operation completes successfully when either the LoadCount value has been loaded or the end of the data file is reached. This parameter can be used in conjunction with *LoadStart* to restart a load after a failure.

ReadBufferSize specifies the size, in KB, of the buffer that is used to read the bulk data file for a bulk load operation. The default is 2048.

Example

```
HDBC      hdbc;
HENV      henv;
void      *driverHandle;
HMODULE   hmod;
PLoadTableFromFile loadTableFromFile;

char      tableName[128];
char      fileName[512];
char      configFile[512];
char      logFile[512];
char      discardFile[512];
int       errorTolerance;
int       warningTolerance;
int       loadStart;
int       loadCount;
int       readBufferSize;

/* Get the driver's connection handle from the DM.
   This handle must be used when calling directly into the driver. */

rc = SQLGetInfo (hdbc, SQL_DRIVER_HDBC, &driverHandle, 0, NULL);
if (rc != SQL_SUCCESS) {
  ODBC_error (henv, hdbc, SQL_NULL_HSTMT);
  EnvClose (henv, hdbc);
  exit (255);
}

/* Get the DM's shared library or DLL handle to the driver. */

rc = SQLGetInfo (hdbc, SQL_DRIVER_HLIB, &hmod, 0, NULL);
if (rc != SQL_SUCCESS) {
  ODBC_error (henv, hdbc, SQL_NULL_HSTMT);
  EnvClose (henv, hdbc);
  exit (255);
}
```

```
loadTableFromFile = (PLoadTableFromFile)
  resolveName (hmod, "LoadTableFromFile");
if (! loadTableFromFile) {
  printf ("Cannot find LoadTableFromFile!\n");
  exit (255);
}

rc = (*loadTableFromFile) (
     driverHandle,
     (const SQLCHAR *) tableName,
     (const SQLCHAR *) fileName,
     errorTolerance, warningTolerance,
     (const SQLCHAR *) configFile,
     (const SQLCHAR *) logFile,
     (const SQLCHAR *) discardFile,
     loadStart, loadCount,
     readBufferSize);
if (rc == SQL_SUCCESS) {
     printf ("Load succeeded.\n");
}
else {
     driverError (driverHandle, hmod);
}
```

DataDirect Bulk Load Statement Attributes

In addition to exporting tables with the ExportTableToFile methods, result sets can be exported to a bulk load data file through the use of two DataDirect statement attributes, SQL_BULK_EXPORT_PARAMS and SQL_BULK_EXPORT. SQL_BULK_EXPORT_PARAMS is used to configure information about where and how the data is to be exported. SQL_BULK_EXPORT begins the bulk export operation.

SQL_BULK_EXPORT_PARAMS

The ValuePtr argument to SQLSetStmtAttr or SQLSetStmtAttrW when the attribute argument is SQL_BULK_EXPORT_PARAMS is a pointer to a BulkExportParams structure. The definitions of the fields in the BulkExportParams structure are the same as the corresponding arguments in the ExportTableToFile and ExportTableToFileW methods except that the generation of the log file is controlled by the EnableLogging field. When EnableLogging is set to 1, the driver writes events that occur during the export to a log file. Events logged to this file are:

- A message for each row that failed to export.
- Total number of rows fetched
- Total number of rows successfully exported
- Total number of rows that failed to export

The log file is located in the same directory as the bulk load data file and has the same base name as the bulk load data file with a .log extension. When EnableLogging is set to 0, no logging takes place

If the bulk export parameters are not set prior to setting the SQL_BULK_EXPORT attribute, the driver uses the current driver code page value, defaults EnableLogging to 1 (enabled), and defaults ErrorTolerance and WarningTolerance to -1 (infinite)

The SQL_BULK_EXPORT_PARAMS structure is as follows:

```
struct BulkExportParams {
  SQLLEN  Version;               /* Must be the value 1 */
  SQLLEN  IANAAppCodePage;
  SQLLEN  EnableLogging;
  SQLLEN  ErrorTolerance;
  SQLLEN  WarningTolerance;
};
```

SQL_BULK_EXPORT

The ValuePtr argument to SQLSetStmtAttr or SQLSetStmtAttrW when the attribute argument is SQL_BULK_EXPORT is a pointer to a string that specifies the file name of the bulk load data file to which the data in the result set will be exported.

Result set export occurs when the SQL_BULK_EXPORT statement attribute is set. If using the SQL_BULK_EXPORT_PARAMS attribute to set values for the bulk export parameters, the SQL_BULK_EXPORT_PARAMS attribute must be set prior to setting the SQL_BULK_EXPORT attribute. Once set, the bulk export parameters remain set for the life of the statement. If the bulk export parameters are not set prior to setting the SQL_BULK_EXPORT attribute, the driver uses the current driver code page value, defaults EnableLogging to 1 (enabled), and defaults ErrorTolerance and WarningTolerance to -1 (infinite)

Both a bulk load data file and a bulk load configuration file are produced by this operation. The configuration file has the same base name as the bulk load data file, but with an XML extension. The configuration file is created in the same directory as the bulk load data file.

10 SQL for Flat-File Drivers

This chapter describes the SQL statements that you can use with the flat-file drivers (Btrieve, dBASE, Paradox, and Text). Any exceptions to the supported SQL functionality described in this chapter are documented in the individual flat-file driver chapters in the *DataDirect Connect Series for ODBC User's Guide*.

The database drivers parse SQL statements and translate them into a form that the database can understand. The SQL statements described in this chapter let you:

- Read, insert, update, and delete rows from a database
- Create new tables
- Drop existing tables

These SQL statements allow your application to be portable across other databases. This chapter includes information about the following topics:

Select Statement

The form of the Select statement supported by the flat-file drivers is:

```
SELECT [DISTINCT] {* | column_expression, ...}
FROM table_names [table_alias] ...
[ WHERE expr1 rel_operator expr2 ]
[ GROUP BY {column_expression, ...} ]
[ HAVING expr1 rel_operator expr2 ]
[ UNION [ALL] (SELECT...) ]
[ ORDER BY {sort_expression [DESC | ASC]}, ... ]
[ FOR UPDATE [OF {column_expression, ...}] ]
```

Select Clause

Follow Select with a list of column expressions you want to retrieve or an asterisk (*) to retrieve all fields.

```
SELECT [DISTINCT] {* | column_expression, [[AS]
column_alias]. . .}
```

column_expression can be simply a field name (for example, LAST_NAME). More complex expressions may include mathematical operations or string manipulation (for example, SALARY * 1.05). See "SQL Expressions" on page 128 for details.

column_alias can be used to give the column a descriptive name. For example, to assign the alias DEPARTMENT to the column DEP:

```
SELECT dep AS department FROM emp
```

Separate multiple column expressions with commas (for example, LAST_NAME, FIRST_NAME, HIRE_DATE).

Field names can be prefixed with the table name or alias. For example, EMP.LAST_NAME or E.LAST_NAME, where E is the alias for the table EMP.

The Distinct operator can precede the first column expression. This operator eliminates duplicate rows from the result of a query. For example:

```
SELECT DISTINCT dep FROM emp
```

Aggregate Functions

Aggregate functions can also be a part of a Select clause. Aggregate functions return a single value from a set of rows. An aggregate can be used with a field name (for example, AVG(SALARY)) or in combination with a more complex column expression (for example, AVG(SALARY * 1.07)). The column expression can be preceded by the Distinct operator. The Distinct operator eliminates duplicate values from an aggregate expression. For example:

```
COUNT (DISTINCT last_name)
```

In this example, only distinct last name values are counted.

Table 10-1 lists valid aggregate functions.

Table 10-1. Aggregate Functions

Aggregate	Returns
SUM	The total of the values in a numeric field expression. For example, SUM(SALARY) returns the sum of all salary field values.
AVG	The average of the values in a numeric field expression. For example, AVG(SALARY) returns the average of all salary field values.

Table 10-1. Aggregate Functions

COUNT	The number of values in any field expression. For example, COUNT(NAME) returns the number of name values. When using COUNT with a field name, COUNT returns the number of non-NULL field values. A special example is COUNT(*), which returns the number of rows in the set, including rows with NULL values.
MAX	The maximum value in any field expression. For example, MAX(SALARY) returns the maximum salary field value.
MIN	The minimum value in any field expression. For example, MIN(SALARY) returns the minimum salary field value.

From Clause

The From clause indicates the tables to be used in the Select statement. The format of the From clause is:

```
FROM table_names [table_alias]
```

table_names can be one or more simple table names in the current working directory or complete path names.

table_alias is a name used to refer to a table in the rest of the Select statement. Database field names may be prefixed by the table alias. Given the table specification:

```
FROM emp E
```

you may refer to the LAST_NAME field as E.LAST_NAME. Table aliases must be used if the Select statement joins a table to itself. For example:

```
SELECT * FROM emp E, emp F WHERE E.mgr_id = F.emp_id
```

The equal sign (=) includes only matching rows in the results.

If you are joining more than one table, you can use LEFT OUTER JOIN, which includes non-matching rows in the first table you name. For example:

```
SELECT * FROM T1 LEFT OUTER JOIN T2 on T1.key = T2.key
```

Where Clause

The Where clause specifies the conditions that rows must meet to be retrieved. The Where clause contains conditions in the form:

```
WHERE expr1 rel_operator expr2
```

expr1 and *expr2* can be field names, constant values, or expressions.

rel_operator is the relational operator that links the two expressions. See "SQL Expressions" on page 128 for details.

For example, the following Select statement retrieves the names of employees that make at least $20,000.

```
SELECT last_name,first_name FROM emp WHERE salary >= 20000
```

Group By Clause

The Group By clause specifies the names of one or more fields by which the returned values should be grouped. This clause is used to return a set of aggregate values. It has the following form:

```
GROUP BY column_expressions
```

column_expressions must match the column expression used in the Select clause. A column expression can be one or more field names of the database table, separated by a comma (,) or one or more expressions, separated by a comma (,). See "SQL Expressions" on page 128 for details.

The following example sums the salaries in each department:

```
SELECT dept_id, sum(salary) FROM emp GROUP BY dept_id
```

This statement returns one row for each distinct department ID. Each row contains the department ID and the sum of the salaries of the employees in the department.

Having Clause

The Having clause enables you to specify conditions for groups of rows (for example, display only the departments that have salaries totaling more than $200,000). This clause is valid only if you have already defined a Group By clause. It has the following form:

```
HAVING expr1 rel_operator expr2
```

expr1 and *expr2* can be field names, constant values, or expressions. These expressions do not have to match a column expression in the Select clause.

rel_operator is the relational operator that links the two expressions. See "SQL Expressions" on page 128 for details.

The following example returns only the departments whose sums of salaries are greater than $200,000:

```
SELECT dept_id, sum(salary) FROM emp
GROUP BY dept_id HAVING sum(salary) > 200000
```

Union Operator

The Union operator combines the results of two Select statements into a single result. The single result is all the returned rows from both Select statements. By default, duplicate rows are

not returned. To return duplicate rows, use the All keyword (UNION ALL). The form is:

```
SELECT statement
UNION ALL
SELECT statement
```

When using the Union operator, the Select lists for each Select statement must have the same number of column expressions with the same data types, and must be specified in the same order. For example:

```
SELECT last_name, salary, hire_date FROM emp
UNION
SELECT name, pay, birth_date FROM person
```

This example has the same number of column expressions, and each column expression, in order, has the same data type.

The following example is *not* valid because the data types of the column expressions are different (salary from emp has a different data type than last_name from raises). This example does have the same number of column expressions in each Select statement but the expressions are not in the same order by data type.

```
SELECT last_name, salary FROM emp
UNION
SELECT salary, last_name FROM raises
```

Order By Clause

The Order By clause indicates how the rows are to be sorted. The form is:

```
ORDER BY {sort_expression [DESC | ASC]}, ...
```

sort_expression can be field names, expressions, or the positioned number of the column expression to use.

The default is to perform an ascending (ASC) sort.

For example, to sort by last_name and then by first_name, you could use either of the following Select statements:

```
SELECT emp_id, last_name, first_name FROM emp
ORDER BY last_name, first_name
```

or

```
SELECT emp_id, last_name, first_name FROM emp
ORDER BY 2,3
```

In the second example, last_name is the second column expression following Select, so Order By 2 sorts by last_name.

For Update Clause

The For Update clause locks the rows of the database table selected by the Select statement. The form is:

```
FOR UPDATE OF column_expressions
```

column_expressions is a list of field names in the database table that you intend to update, separated by a comma (,).

The following example returns all rows in the employee database that have a salary field value of more than $20,000. When each record is fetched, it is locked. If the record is updated or deleted, the lock is held until you commit the change. Otherwise, the lock is released when you fetch the next record.

```
SELECT * FROM emp WHERE salary > 20000
  FOR UPDATE OF last_name, first_name, salary
```

SQL Expressions

Expressions are used in the Where clauses, Having clauses, and Order By clauses of SQL Select statements.

Expressions enable you to use mathematical operations as well as character string and date manipulation operators to form complex database queries.

The most common expression is a simple field name. You can combine a field name with other expression elements.

Valid expression elements are as follows:

- Field names
- Constants
- Exponential notation
- Numeric operators
- Character operators
- Date operators
- Relational operators
- Logical operators
- Functions

Constants

Constants are values that do not change. For example, in the expression PRICE * 1.05, the value 1.05 is a constant.

You must enclose character constants in pairs of single (') or double (") quotation marks. To include a single quotation mark in a character constant enclosed by single quotation marks, use two single quotation marks together (for example, 'Don''t'). Similarly, if the constant is enclosed by double quotation marks, use two double quotation marks to include one.

You must enclose date and time constants in braces ({}), for example, {01/30/89} and {12:35:10}. The form for date constants is MM/DD/YY or MM/DD/YYYY. The form for time constants is HH:MM:SS.

The logical constants are .T. and 1 for True and .F. and 0 for False. For portability, use 1 and 0.

Exponential Notation

You can include exponential notation in expression elements. For example:

```
SELECT col1, 3.4E+7 FROM table1 WHERE calc < 3.4E-6 * col2
```

Numeric Operators

You can include the following operators in numeric expressions:

Operator	Meaning
+	Addition
–	Subtraction
*	Multiplication
/	Division
**	Exponentiation
^	Exponentiation

The following table shows examples of numeric expressions. For these examples, assume salary is 20000.

Example	Resulting value
`salary + 10000`	30000
`salary * 1.1`	22000
`2 ** 3`	8

You can precede numeric expressions with a unary plus (+) or minus (–). For example, `-(salary * 1.1)` is -22000.

Character Operators

Character expressions can include the following operators:

Operator	Meaning
+	Concatenation, keeping trailing blanks.
–	Concatenation, moving trailing blanks to the end.

The following table shows examples of character expressions. In the examples, last_name is `'JONES  '` and first_name is `'ROBERT  '`.

Example	Resulting Value
`first_name + last_name`	`'ROBERT     JONES  '`
`first_name – last_name`	`'ROBERTJONES       '`

NOTE: Some flat-file drivers return character data with trailing blanks as shown in the table; however, you cannot rely on the driver to return blanks. If you want an expression that works regardless of whether the drivers return trailing blanks, use the TRIM function before concatenating strings to make the expression portable. For example:

```
TRIM(first_name) + '' + TRIM(last_name)
```

Date Operators

You can include the following operators in date expressions:

Operator	Meaning
+	Add a number of days to a date to produce a new date.
–	The number of days between two dates, or subtract a number of days from a date to produce a new date.

The following table shows examples of date expressions. In these examples, hire_date is {01/30/1990}.

Example	Resulting Value
`hire_date + 5`	`{02/04/1990}`
`hire_date - {01/01/1990}`	`29`
`hire_date - 10`	`{01/20/1990}`

Relational Operators

Relational operators separating any two expressions can be any one of those listed in Table 10-2.

Table 10-2. Relational Operators

Operator	Meaning
=	Equal.
<>	Not Equal.
>	Greater Than.
>=	Greater Than or Equal.
<	Less Than.
<=	Less Than or Equal.
Like	Matching a pattern.
Not Like	Not matching a pattern.
Is NULL	Equal to NULL.
Is Not NULL	Not Equal to NULL.
Between	Range of values between a lower and upper bound.
In	A member of a set of specified values or a member of a subquery.
Exists	True if a subquery returned at least one record.

Table 10-2. Relational Operators *(cont.)*

Operator	Meaning
Any	Compares a value to each value returned by a subquery. Any must be prefaced by =, <>, >, >=, <, or <=.
	=Any is equivalent to In.
All	Compares a value to each value returned by a subquery. All must be prefaced by =, <>, >, >=, <, or <=.

The following list shows some examples of relational operators:

```
salary <= 40000
dept = 'D101'
hire_date > {01/30/1989}
salary + commission >= 50000
last_name LIKE 'Jo%'
salary IS NULL
salary BETWEEN 10000 AND 20000
WHERE salary = ANY (SELECT salary FROM emp WHERE dept = 'D101')
WHERE salary > ALL (SELECT salary FROM emp WHERE dept = 'D101')
```

Logical Operators

Two or more conditions may be combined to form more complex criteria. When two or more conditions are present, they must be related by AND or OR. For example:

```
salary = 40000 AND exempt = 1
```

The logical NOT operator is used to reverse the meaning. For example:

```
NOT (salary = 40000 AND exempt = 1)
```

Operator Precedence

As expressions become more complex, the order in which the expressions are evaluated becomes important. Table 10-3 shows the order in which the operators are evaluated. The operators in the first line are evaluated first, then those in the second line, and so on. Operators in the same line are evaluated left to right in the expression.

Table 10-3. Operator Precedence

Precedence	Operator
1	Unary -, Unary +
2	**
3	*, /
4	+, -
5	=, <>, <, <=, >, >=, LIKE, NOT LIKE, IS NULL, IS NOT NULL, BETWEEN, IN, EXISTS, ANY, ALL
6	NOT
7	AND
8	OR

The following example shows the importance of precedence:

```
WHERE salary > 40000 OR
hire_date > {01/30/1989} AND
dept = 'D101'
```

Because AND is evaluated first, this query retrieves employees in department D101 hired after January 30, 1989, as well as every employee making more than $40,000, no matter what department or hire date.

To force the clause to be evaluated in a different order, use parentheses to enclose the conditions to be evaluated first. For example:

```
WHERE (salary > 40000 OR hire_date > {01/30/1989})
AND dept = 'D101'
```

retrieves employees in department D101 that either make more than $40,000 or were hired after January 30, 1989.

Functions

The flat-file drivers support a number of functions that you may use in expressions. In Table 10-4 through Table 10-6 on page 139, the functions are grouped according to the type of result they return.

Table 10-4. Functions that Return Character Strings

Function	Description
CHR	Converts an ASCII code into a one-character string. `CHR(67)` returns `C`.
RTRIM	Removes trailing blanks from a string. `RTRIM('ABC  ')` returns `ABC`.
TRIM	Removes trailing blanks from a string. `TRIM('ABC  ')` returns `ABC`.
LTRIM	Removes leading blanks from a string. `LTRIM('  ABC')` returns `ABC`.
UPPER	Changes each letter of a string to uppercase. `UPPER('Allen')` returns `ALLEN`.
LOWER	Changes each letter of a string to lowercase. `LOWER('Allen')` returns `allen`.
LEFT	Returns leftmost characters of a string. `LEFT('Mattson',3)` returns `Mat`.

Table 10-4. Functions that Return Character Strings (cont.)

Function	Description
RIGHT	Returns rightmost characters of a string. `RIGHT('Mattson',4)` returns `tson`.
SUBSTR	Returns a substring of a string. Parameters are the string, the first character to extract, and the number of characters to extract (optional). `SUBSTR('Conrad',2,3)` returns `onr`. `SUBSTR('Conrad',2)` returns `onrad`.
SPACE	Generates a string of blanks. `SPACE(5)` returns `'  '`.
DTOC	Converts a date to a character string. An optional second parameter determines the format of the result: 0 (the default) returns MM/DD/YY. 1 returns DD/MM/YY. 2 returns YY/MM/DD. 10 returns MM/DD/YYYY. 11 returns DD/MM/YYYY. 12 returns YYYY/MM/DD. An optional third parameter specifies the date separator character. If not specified, a slash (/) is used. `DTOC({01/30/1997})` returns `01/30/97`. `DTOC({01/30/1997}, 0)` returns `01/30/97`. `DTOC({01/30/1997}, 1)` returns `30/01/97`. `DTOC({01/30/1997}, 2,'-')` returns `97-01-30`.
DTOS	Converts a date to a character string using the format YYYYMMDD. `DTOS({01/23/1990})` returns `19900123`.
IIF	Returns one of two values, true or false. Parameters are a logical expression, the true value, and the false value. If the logical expression evaluates to true, the function returns the true value. Otherwise, it returns the false value. `IIF(salary>20000,'BIG','SMALL')` returns `BIG` if `salary` is greater than 20000. If not, it returns `SMALL`.

***Table 10-4. Functions that Return Character Strings** (cont.)*

Function	Description

STR	Converts a number to a character string. Parameters are the number, the total number of output characters (including the decimal point), and optionally the number of digits to the right of the decimal point.
	`STR(12.34567,4)` returns `12.`
	`STR(12.34567,4,1)` returns `12.3.`
	`STR(12.34567,6,3)` returns `12.346.`

STRVAL	Converts a value of any type to a character string.
	`STRVAL('Woltman')` returns `Woltman.`
	`STRVAL({12/25/1953})` returns `12/25/1953.`
	`STRVAL (5 * 3)` returns `15.`
	`STRVAL (4 = 5)` returns `'False'.`

TIME	Returns the time of day as a character string. At 9:49 PM, `TIME()` returns `21:49:00.`
TTOC	NOTE: This function applies only to flat-file drivers that support SQL_TIMESTAMP: the Btrieve, dBASE (access to FoxPro 3.0), and Paradox drivers. Converts a timestamp to a character string. An optional second parameter determines the format of the result: When set to 0 or none (the default), MM/DD/YY HH:MM:SS AM is returned. When set to 1, YYYYMMDDHHMMSS is returned, which is a suitable format for indexing. `TTOC({1992-04-02 03:27:41})` returns 04/02/92 03:27:41 AM. `TTOC({1992-04-02 03:27:41, 1})` returns 19920402032741
USERNAME	For Btrieve, the logon ID specified at connect time is returned. For Paradox drivers, the user name specified during configuration is returned. For all other flat-file drivers, an empty string is returned.

Table 10-5. Functions that Return Numbers

Function	Description
MOD	Divides two numbers and returns the remainder of the division. `MOD(10,3)` returns `1`.
LEN	Returns the length of a string. `LEN('ABC')` returns `3`.
MONTH	Returns the month part of a date. `MONTH({01/30/1989})` returns `1`.
DAY	Returns the day part of a date. `DAY({01/30/1989})` returns `30`.
YEAR	Returns the year part of a date. `YEAR({01/30/1989})` returns `1989`.
MAX	Returns the larger of two numbers. `MAX(66,89)` returns `89`.
DAYOFWEEK	Returns the day of week (1-7) of a date expression. `DAYOFWEEK({05/01/1995})` returns `5`.
MIN	Returns the smaller of two numbers. `MIN(66,89)` returns `66`.
POW	Raises a number to a power. `POW(7,2)` returns `49`.
INT	Returns the integer part of a number. `INT(6.4321)` returns `6`.
ROUND	Rounds a number. `ROUND(123.456, 0)` returns `123`. `ROUND(123.456, 2)` returns `123.46`. `ROUND(123.456, -2)` returns `100`.

Table 10-5. Functions that Return Numbers (cont.)

Function	Description
NUMVAL	Converts a character string to a number. If the character string is not a valid number, a zero (0) is returned. `NUMVAL('123')` returns the number `123`.
VAL	Converts a character string to a number. If the character string is not a valid number, a zero (0) is returned. `VAL('123')` returns the number `123`.

Table 10-6. Functions that Return Dates

Function	Description
DATE	Returns today's date. If today is 12/25/1999, `DATE()` returns `{12/25/1999}`.
TODAY	Returns today's date. If today is 12/25/1999, `TODAY()` returns `{12/25/1999}`.
DATEVAL	Converts a character string to a date. `DATEVAL('01/30/1989')` returns `{01/30/1989}`.
CTOD	Converts a character string to a date. An optional second parameter specifies the format of the character string: 0 (the default) returns MM/DD/YY, 1 returns DD/MM/YY, and 2 returns YY/MM/DD. `CTOD('01/30/1989')` returns `{01/30/1989}`. `CTOD('01/30/1989',1)` returns `{30/01/1989}`.

The following examples use some of the number and date functions.

Retrieve all employees that have been with the company at least 90 days:

```
SELECT first_name, last_name FROM emp
  WHERE DATE() - hire_date >= 90
```

Retrieve all employees hired in January of this year or last year:

```
SELECT first_name, last_name FROM emp
  WHERE MONTH(hire_date) = 1
  AND (YEAR(hire_date) = YEAR(DATE())
  OR YEAR(hire_date) = YEAR(DATE()) - 1)
```

Create and Drop Table Statements

The flat-file drivers support SQL statements to create and delete database files. The Create Table statement is used to create files and the Drop Table statement is used to delete files.

Create Table

The form of the Create Table statement is:

```
CREATE TABLE table_name (col_definition[,col_definition,...])
```

table_name can be a simple table name or a full path name. A table name is preferred for portability to other SQL data sources. If a table name is used, the file is created in the directory you specified as the database directory in the connection string. If you did not specify a database directory in the connection string, the file is created in the directory specified as the database directory in .odbc.ini. If you did not specify a database directory in either place, the file is created in the current working directory at connect time.

col_definition is the column name, followed by the data type, followed by an optional column constraint definition. Values for column names are database specific. The data type specifies a column's data type.

The only column constraint definition currently supported by some flat-file drivers is "not NULL." Not all flat-file tables support

"not NULL" columns. In the cases where "not NULL" is not supported, this restriction is ignored and the driver returns a warning if "not NULL" is specified for a column. The "not NULL" column constraint definition is allowed in the driver so that you can write a database-independent application (and not be concerned about the driver raising an error on a Create Table statement with a "not NULL" restriction).

A sample Create Table statement to create an employee database table is:

```
CREATE TABLE emp (last_name CHAR(20) NOT NULL,
  first_name CHAR(12) NOT NULL,
  salary NUMERIC (10,2) NOT NULL,
  hire_date DATE NOT NULL)
```

Drop Table

The form of the Drop Table statement is:

```
DROP TABLE table_name
```

table_name can be a simple table name (emp) or a full path name. A table name is preferred for portability to other SQL data sources. If a table name is used, the file is dropped from the directory you specified as the database directory in the connection string. If you did not specify a database directory in the connection string, the file is deleted from the directory specified as the database directory in .odbc.ini. If you did not specify a database directory in either of these places, the file is dropped from the current working directory at connect time.

A sample Drop Table statement to delete the emp table is:

```
DROP TABLE emp
```

Insert Statement

The Insert statement is used to add new rows to a database table. With it, you can specify either of the following options:

- A list of values to be inserted as a new record
- A Select statement that copies data from another table to be inserted as a set of new rows

The form of the Insert statement is:

```
INSERT INTO table_name [(col_name, ...)]
{VALUES (expr, ...) | select_statement}
```

table_name can be a simple table name or a full path name. A table name is preferred for portability to other SQL data sources.

col_name is an optional list of column names giving the name and order of the columns whose values are specified in the Values clause. If you omit *col_name*, the value expressions (*expr*) must provide values for all columns defined in the file and must be in the same order that the columns are defined for the file.

expr is the list of expressions giving the values for the columns of the new record. Usually, the expressions are constant values for the columns. Character string values must be enclosed in single (') or double (") quotation marks, date values must be enclosed in braces {}, and logical values that are letters must be enclosed in periods (for example, .T. or .F.).

An example of an Insert statement that uses a list of expressions is:

```
INSERT INTO emp (last_name, first_name, emp_id, salary, hire_date)
VALUES ('Smith', 'John', 'E22345', 27500, {4/6/1999})
```

Each Insert statement adds one record to the database table. In this case a record has been added to the employee database table, emp. Values are specified for five columns. The remaining columns in the table are assigned a blank value, meaning NULL.

select_statement is a query that returns values for each *col_name* value specified in the column name list. Using a Select statement instead of a list of value expressions lets you select a set of rows from one table and insert it into another table using a single Insert statement.

An example of an Insert statement that uses a Select statement is:

```
INSERT INTO emp1 (first_name, last_name, emp_id, dept, salary)
SELECT first_name, last_name, emp_id, dept, salary from emp
WHERE dept = 'D050'
```

In this type of Insert statement, the number of columns to be inserted must match the number of columns in the Select statement. The list of columns to be inserted must correspond to the columns in the Select statement just as it would to a list of value expressions in the other type of Insert statement. That is, the first column inserted corresponds to the first column selected; the second inserted to the second, and so forth.

The size and data type of these corresponding columns must be compatible. Each column in the Select list should have a data type that the driver accepts on a regular Insert/Update of the corresponding column in the Insert list. Values are truncated when the size of the value in the Select list column is greater than the size of the corresponding Insert list column.

The Select statement is evaluated before any values are inserted. This query cannot be made on the table into which values are inserted.

Update Statement

The Update statement is used to change rows in a database file. The form of the Update statement supported for flat-file drivers is:

```
UPDATE table_name SET col_name = expr, ...
[ WHERE { conditions | CURRENT OF cursor_name } ]
```

table_name can be a simple table name or a full path name. A table name is preferred for portability to other SQL data sources.

col_name is the name of a column whose value is to be changed. Several columns can be changed in one statement.

expr is the new value for the column. The expression can be a constant value or a subquery. Character string values must be enclosed with single (') or double (") quotation marks, date values must be enclosed by braces {}, and logical values that are letters must be enclosed by periods (for example, .T. or .F.). Subqueries must be enclosed in parentheses.

The Where clause (any valid clause described in "Select Statement" on page 122) determines which rows are to be updated.

The Where Current Of *cursor_name* clause can be used only by developers coding directly to the ODBC API. It causes the row at which *cursor_name* is positioned to be updated. This is called a "positioned update." You must first execute a Select...For Update statement with a named cursor and fetch the row to be updated.

An example of an Update statement on the emp table is:

```
UPDATE emp SET salary=32000, exempt=1
WHERE emp_id = 'E10001'
```

The Update statement changes every record that meets the conditions in the Where clause. In this case, the salary and exempt status are changed for all employees having the

employee ID E10001. Because employee IDs are unique in the emp table, only one record is updated.

An example using a subquery is:

```
UPDATE emp SET salary = (SELECT avg(salary) from emp)
WHERE emp_id = 'E10001'
```

In this case, the salary is changed to the average salary in the company for the employee having employee ID E10001.

Delete Statement

The Delete statement is used to delete rows from a database table. The form of the Delete statement supported for flat-file drivers is:

```
DELETE FROM table_name
[ WHERE { conditions | CURRENT OF cursor_name } ]
```

table_name can be a simple table name or a full path name. A table name is preferred for portability to other SQL data sources.

The Where clause determines which rows are to be deleted. If you include only the keyword Where, all rows in the table are deleted, but the file is left intact.

The Where Current Of *cursor_name* clause can be used only by developers coding directly to the ODBC API. It causes the row at which *cursor_name* is positioned to be deleted. This is called a "positioned delete." You must first execute a Select...For Update statement with a named cursor and fetch the row to be deleted.

An example of a Delete statement on the emp table is:

```
DELETE FROM emp WHERE emp_id = 'E10001'
```

Each Delete statement removes every record that meets the conditions in the Where clause. In this case, every record having the employee ID E10001 is deleted. Because employee IDs are unique in the employee table, at most, one record is deleted.

Reserved Keywords

The following words are reserved for use in SQL statements. If they are used for file or column names in a database that you use, you must enclose them in double (") quotation marks in any SQL statement where they appear as file or column names.

- ALL
- AND
- BETWEEN
- COMPUTE
- CROSS
- DISTINCT
- FOR
- FROM
- FULL
- GROUP
- HAVING
- INNER
- INTO
- LEFT
- LIKE
- NATURAL
- NOT
- NULL
- ON
- OPTIONS
- OR
- ORDER
- RIGHT
- UNION
- WHERE

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.

client load balancing

Client load balancing distributes new connections in a computing environment so that no one server is overwhelmed with connection requests.

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 failover

Connection failover allows an application to connect to an alternate, or backup, database server if the primary database server is unavailable, for example, because of a hardware failure or traffic overload.

connection retry

Connection retry defines the number of times the driver attempts to connect to the primary and, if configured, alternate database servers after the initial unsuccessful connection attempt. Connection retry can be an important strategy for system recovery.

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/Linux. Applications deployed in the global assembly cache must have a strong name to handle name and version conflicts.

DTC (Distributed Transaction Coordinator) In Microsoft Windows NT, Windows 2000, Windows XP, Windows Vista, and the Windows Server 2003 family, the DTC is a system service that is part of COM+ services. COM+ components that use DTC can enlist ODBC connections in distributed transactions. This makes it possible to scale transactions from one to many computers without adding special code.

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)

load balancing

See client load balancing.

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 ensures that only one user at a time can affect the data.

MTS (Microsoft Transaction Server)

MTS is a component-based transaction processing system for developing, deploying, and managing high-performance, scalable, and robust enterprise, Internet, and intranet server applications. MTS was the precursor to COM+, the current version of this processing system (see DTC).

ODBC Administrator

The ODBC Data Source Administrator manages database drivers and configures DSNs. On computers running the Microsoft Windows 2000, XP, or Vista operating systems, this application is located in the Windows Control Panel under Administrative Tools. Its icon is named "Data Sources (ODBC)."

In UNIX/Linux environments, the DataDirect UNIX ODBC Data Source Administrator is located in the /tools directory of the product installation directory.

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

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W