



# Microsoft® Data Access Components 2.5 SDK

## PART I – Microsoft ODBC API Specification

### Introduction to ODBC

This PDF file contains contents from the Microsoft® Data Access SDK online *ODBC Programmer's Reference* at the following web site:

<http://msdn.microsoft.com/isapi/msdnlib2.idc?theURL=/library/psdk/dasdk/odin8w4s.htm>

For your convenience, the main sections of the Microsoft documentation are provided in three PDF files that are available on the SOLID Web site:

- The Part I Microsoft ODBC API Specification PDF file contains introductory information on ODBC and information on developing applications and drivers.
- The Part II Microsoft ODBC API Specification PDF file contains the ODBC API Reference, descriptions of each ODBC function.
- The Part III Microsoft ODBC API Specification PDF file contains information on installing and configuring ODBC software, and setup, installer, and translation DLL functions.

NOTE: Refer to Chapter 2, “Using SOLID ODBC API,” in the **SOLID Programmer Guide** for SOLID-specific usage. This PDF file also refers you to SOLID manuals for information on SOLID usage and to the Microsoft Web site (noted above) for those portions of the documentation that are not included in the PDF file.

Copyright © 2000 Microsoft Corporation, All rights reserved.

Information in this document is subject to change without notice and does not represent a commitment on the part of Solid Information Technology.

# Contents

|  |           |
|--|-----------|
| <b>About Part I: Introduction to ODBC</b>                  | <b>5</b>  |
| <b>Section 1: Introduction to ODBC</b>                     | <b>6</b>  |
| <b>Chapter 1: Introduction</b>                             | <b>6</b>  |
| Why Was ODBC Created?                                      | 6         |
| What Is ODBC?  | 7         |
| ODBC and the Standard CLI                                  | 8         |
| <b>Chapter 2: An Introduction to SQL and ODBC</b>          | <b>9</b>  |
| Structured Query Language (SQL)                            | 9         |
| Processing an SQL Statement                                | 10        |
| Dynamic SQL  | 12        |
| Call-Level Interfaces                                      | 13        |
| Network Database Access                                    | 13        |
| The ODBC Solution  | 15        |
| <b>Chapter 3: ODBC Architecture</b>                        | <b>17</b> |
| Applications   | 18        |
| The Driver Manager   | 19        |
| Drivers  | 20        |
| Driver Tasks   | 20        |
| Driver Architecture  | 20        |
| DBMS-Based Drivers   | 21        |
| Types of Data Sources                                      | 22        |
| Using Data Sources   | 23        |
| <b>Section 2: Developing Applications and Drivers</b>      | <b>25</b> |
| <b>Chapter 4: ODBC Fundamentals</b>                        | <b>26</b> |
| Handles  | 26        |
| Buffers  | 31        |
| Using Data Buffers   | 34        |
| Data Types in ODBC   | 39        |
| C Data Types in ODBC                                       | 40        |
| Data Type Conversions                                      | 41        |
| Conformance Levels   | 41        |
| SQL Conformance Levels                                     | 51        |
| Environment, Connection, and Statement Attributes          | 51        |
| Tables and Views   | 52        |
| <b>Chapter 5: Overview of Basic ODBC Application Steps</b> | <b>52</b> |
| Step 1: Connect to the Data Source                         | 54        |
| Step 2: Initialize the Application                         | 54        |
| Step 3: Build and Execute an SQL Statement                 | 55        |
| Step 4a: Fetch the Results                                 | 55        |
| Step 4b: Fetch the Row Count                               | 56        |

|   |            |
|---|------------|
| Step 5: Commit the Transaction                                      | 56         |
| Step 6: Disconnect from the Data Source                             | 57         |
| <b>Chapter 6: Overview of Connecting to a Data Source or Driver</b> | <b>58</b>  |
| Allocating the Environment Handle                                   | 58         |
| Declaring the Application's ODBC Version                            | 58         |
| Choosing a Data Source or Driver                                    | 59         |
| Allocating a Connection Handle                                      | 61         |
| Connection Attributes   | 61         |
| Establishing a Connection   | 62         |
| <b>Chapter 7: Overview of Catalog Functions</b>                     | <b>73</b>  |
| Uses of Catalog Data  | 73         |
| Catalog Functions in ODBC   | 74         |
| Schema Views  | 79         |
| <b>Chapter 8: Overview of SQL Statements</b>                        | <b>80</b>  |
| Constructing SQL Statements   | 80         |
| Interoperability of SQL Statements                                  | 84         |
| Constructing Interoperable SQL Statements                           | 85         |
| Escape Sequences in ODBC  | 89         |
| <b>Chapter 9: Overview of Executing Statements</b>                  | <b>95</b>  |
| Allocating a Statement Handle                                       | 95         |
| Executing a Statement   | 96         |
| Procedures  | 101        |
| Batches of SQL Statements   | 103        |
| Result-Generating and Result-Free Statements                        | 104        |
| Executing Catalog Functions   | 106        |
| Statement Parameters  | 106        |
| Procedure Parameters  | 114        |
| Asynchronous Execution  | 121        |
| <b>Chapter 10: Overview of Retrieving Results (Basic)</b>           | <b>126</b> |
| Was a Result Set Created?   | 126        |
| Result Set Metadata   | 127        |
| Binding Columns   | 128        |
| Fetching Data   | 133        |
| Closing the Cursor  | 137        |
| <b>Chapter 11: Overview of Retrieving Results (Advanced)</b>        | <b>138</b> |
| Block Cursors   | 138        |
| Using Block Cursors   | 143        |
| SQLGetData and Block Cursors  | 144        |
| Row Status Array  | 144        |
| Using Scrollable Cursors  | 148        |
| Cursor Characteristics and Cursor Type                              | 149        |
| Relative and Absolute Scrolling                                     | 153        |
| The ODBC Cursor Library   | 156        |
| Multiple Results  | 156        |

|   |            |
|---|------------|
| <b>Chapter 12: Overview of Updating Data</b>  | <b>158</b> |
| UPDATE, DELETE, and INSERT Statements   | 158        |
| Positioned Update and Delete Statements   | 158        |
| Simulating Positioned Update and Delete Statements  | 161        |
| Determining the Number of Affected Rows   | 163        |
| Updating Data with SQLSetPos  | 163        |
| Updating Data with SQLBulkOperations  | 167        |
| Long Data and SQLSetPos and SQLBulkOperations   | 170        |
| <b>Chapter 13: Overview of Descriptors</b>  | <b>172</b> |
| Descriptor Fields   | 173        |
| Allocating and Freeing Descriptors  | 176        |
| Getting and Setting Descriptor Fields   | 178        |
| <b>Chapter 14: Overview of Transactions</b>   | <b>180</b> |
| Transactions in ODBC  | 180        |
| <b>Chapter 15: Overview of Diagnostics</b>  | <b>191</b> |
| Return Codes  | 191        |
| Diagnostic Records  | 192        |
| Status Records  | 193        |
| Using SQLGetDiagRec and SQLGetDiagField   | 196        |
| Diagnostic Handling Examples  | 199        |
| <b>Chapter 16: Overview of Interoperability</b>   | <b>202</b> |
| Is ODBC the Answer?   | 202        |
| Choosing a Level of Interoperability  | 202        |
| Length of the Product Cycle   | 206        |
| Testing Interoperable Applications  | 208        |
| <b>Chapter 17: Overview of Programming Considerations</b>   | <b>210</b> |
| Multithreading  | 210        |
| Alignment   | 210        |
| Unicode   | 211        |
| Translation DLLs  | 216        |
| Enabling Tracing  | 217        |
| Dynamic Tracing   | 218        |
| Driver-Specific Data Types, Descriptor Types, Information Types, Diagnostic Types, and Attributes | 218        |
| Application/Driver Compatibility  | 220        |
| New Features  | 223        |
| Block Cursors, Scrollable Cursors, and Backward Compatibility for ODBC 3.x Applications           | 225        |
| Duplicated Features   | 228        |
| Writing ODBC 3.x Applications   | 233        |
| Mapping Replacement Functions for Backward Compatibility of Applications                          | 236        |
| Writing ODBC 3.x Drivers  | 249        |
| ODBC in Windows   | 251        |

# About Part I: Introduction to ODBC

This PDF file is an excerpt from the Microsoft *ODBC Programmer's Reference* and is divided into the following parts:

[Section 1: Introduction to ODBC](#), providing a brief history of Structured Query Language and ODBC and conceptual information about the ODBC interface

[Section 2: Developing Applications and Drivers](#), containing information about developing applications that use the ODBC interface and drivers that implement it.

# Section 1: Introduction to ODBC

Part 1 of the *ODBC Programmer's Reference* offers a brief history of Structured Query Language and ODBC, and includes conceptual information about the ODBC interface. This section contains the following chapters:

[Chapter 1: Introduction](#)

[Chapter 2: An Introduction to SQL and ODBC](#)

[Chapter 3: ODBC Architecture](#)

## Chapter 1: Introduction

Open Database Connectivity (ODBC) is a widely accepted application programming interface (API) for database access. It is based on the Call-Level Interface (CLI) specifications from X/Open and ISO/IEC for database APIs and uses Structured Query Language (SQL) as its database access language.

ODBC is designed for maximum *interoperability*—that is, the ability of a single application to access different database management systems (DBMSs) with the same source code. Database applications call functions in the ODBC interface, which are implemented in database-specific modules called *drivers*. The use of drivers isolates applications from database-specific calls in the same way that printer drivers isolate word processing programs from printer-specific commands. Because drivers are loaded at run time, a user only has to add a new driver to access a new DBMS; it is not necessary to recompile or relink the application.

### Why Was ODBC Created?

Historically, companies used a single DBMS. All database access was done either through the front end of that system or through applications written to work exclusively with that system. However, as the use of computers grew and more computer hardware and software became available, companies started to acquire different DBMSs. The reasons were many: People bought what was cheapest, what was fastest, what they already knew, what was latest on the market, what worked best for a single application. Other reasons were reorganizations and mergers, where departments that previously had a single DBMS now had several.

The issue grew even more complex with the advent of personal computers. These computers brought in a host of tools for querying, analyzing, and displaying data, along with a number of inexpensive, easy-to-use databases. From then on, a single corporation often had data scattered across a myriad of desktops, servers, and minicomputers, stored in a variety of incompatible databases, and accessed by a vast number of different tools, few of which could get at all of the data.

The final challenge came with the advent of client/server computing, which seeks to make the most efficient use of computer resources. Inexpensive personal computers (the clients) sit on the desktop and provide both a graphical front end to the data and a number of inexpensive tools, such as spreadsheets, charting programs, and report builders. Minicomputers and mainframe computers (the servers) host the DBMSs, where they can use their computing power and central location to provide quick, coordinated data access. How then was the front-end software to be connected to the back-end databases?

A similar problem faced independent software vendors (ISVs). Vendors writing database software for minicomputers and mainframes were usually forced to write one version of an application for each DBMS or write DBMS-specific code for each DBMS they wanted to access. Vendors writing software for personal computers had to write data access routines for each different DBMS with which they wanted to work. This often meant a huge amount of resources were spent writing and maintaining data access routines rather than applications, and applications were often sold not on their quality but on whether they could access data in a given DBMS.

What both sets of developers needed was a way to access data in different DBMSs. The mainframe and minicomputer group needed a way to merge data from different DBMSs in a single application, while the personal computer group needed this ability as well as a way to write a single application that was independent of any one DBMS. In short, both groups needed an interoperable way to access data; they needed open database connectivity.

## **What Is ODBC?**

Many misconceptions about ODBC exist in the computing world. To the end user, it is an icon in the Microsoft® Windows® Control Panel. To the application programmer, it is a library containing data access routines. To many others, it is the answer to all database access problems ever imagined.

First and foremost, ODBC is a specification for a database API. This API is independent of any one DBMS or operating system; although this manual uses C, the ODBC API is language-independent. The ODBC API is based on the CLI specifications from X/Open and ISO/IEC. ODBC 3.x fully implements both of these specifications—earlier versions of ODBC were based on preliminary versions of these specifications but did not fully implement them—and adds features commonly needed by developers of screen-based database applications, such as scrollable cursors.

The functions in the ODBC API are implemented by developers of DBMS-specific drivers. Applications call the functions in these drivers to access data in a DBMS-independent manner. A Driver Manager manages communication between applications and drivers.

Although Microsoft provides a Driver Manager for computers running Microsoft Windows NT® Server/Windows 2000 Server, Microsoft Windows NT Workstation/Windows 2000 Professional, and Microsoft Windows® 95/98, has written several ODBC drivers, and calls ODBC functions from some of its applications, anybody can write ODBC applications and drivers. In fact, the vast majority of ODBC applications and drivers available for computers running Windows NT Server/Windows 2000 Server, Windows NT Workstation/Windows 2000 Professional, and Windows 95/98 are produced by companies other than Microsoft. Furthermore, ODBC drivers and applications exist on the Macintosh® and a variety of UNIX platforms.

To help application and driver developers, Microsoft offers an ODBC Software Development Kit (SDK) for computers running Windows NT Server/Windows 2000 Server, Windows NT Workstation/Windows 2000 Professional, and Windows 95/98 that provides the Driver Manager, installer DLL, test tools, and sample applications. Microsoft has teamed with Visigenic Software to port these SDKs to the Macintosh and a variety of UNIX platforms.

It is important to understand that ODBC is designed to expose database capabilities, not supplement them. Thus, application writers should not expect that using ODBC will suddenly transform a simple database into a fully featured relational database engine. Nor are driver writers expected to implement functionality

not found in the underlying database. An exception to this is that developers who write drivers that directly access file data (such as data in an Xbase file) are required to write a database engine that supports at least minimal SQL functionality. Another exception is that the ODBC component of the Microsoft® Data Access Components (MDAC) SDK provides a cursor library that simulates scrollable cursors for drivers that implement a certain level of functionality.

Applications that use ODBC are responsible for any cross-database functionality. For example, ODBC is not a heterogeneous join engine, nor is it a distributed transaction processor. However, because it is DBMS-independent, it can be used to build such cross-database tools.

## ODBC and the Standard CLI

ODBC aligns with the following specifications and standards that deal with the Call-Level Interface (CLI). (The ODBC features are a superset of each of these standards.)

The X/Open CAE Specification "Data Management: SQL Call-Level Interface (CLI)"  
ISO/IEC 9075-3:1995 (E) Call-Level Interface (SQL/CLI)

As a result of this alignment, the following are true:

An application written to the X/Open and ISO CLI specifications will work with an ODBC 3.x driver or a standards-compliant driver when it is compiled with the ODBC 3.x header files and linked with ODBC 3.x libraries, and when it gains access to the driver through the ODBC 3.x Driver Manager.

A driver written to the X/Open and ISO CLI specifications will work with an ODBC 3.x application or a standards-compliant application when it is compiled with the ODBC 3.x header files and linked with ODBC 3.x libraries, and when the application gains access to the driver through the ODBC 3.x Driver Manager. (For more information, see "[Standards-Compliant Applications and Drivers](#)" in Chapter 17, "Programming Considerations.")

The Core interface conformance level encompasses all the features in the ISO CLI and all the nonoptional features in the X/Open CLI. Optional features of the X/Open CLI appear in higher interface conformance levels. Because all ODBC 3.x drivers are required to support the features in the Core interface conformance level, the following are true:

An ODBC 3.x driver will support all the features used by a standards-compliant application.

An ODBC 3.x application using only the features in ISO CLI and the nonoptional features of the X/Open CLI will work with any standards-compliant driver.

In addition to the call-level interface specifications contained in the ISO/IEC and X/Open CLI standards, ODBC implements the following features. (Some of these features existed in versions of ODBC prior to ODBC 3.x.)

- Multirow fetches by a single function call
- Binding to an array of parameters
- Bookmark support including fetching by bookmark, variable-length bookmarks, and bulk update and delete by bookmark operations on discontinuous rows
- Row-wise binding
- Binding offsets



- Support for batches of SQL statements, either in a stored procedure or as a sequence of SQL statements executed through **SQLExecute** or **SQLExecDirect**
- Exact or approximate cursor row counts
- Positioned update and delete operations and batched updates and deletes by function call (**SQLSetPos**)
- Catalog functions that extract information from the information schema without the need for supporting information schema views
- Escape sequences for outer joins, scalar functions, datetime literals, interval literals, and stored procedures
- Code-page translation libraries
- Reporting of a driver's ANSI-conformance level and SQL support
- On-demand automatic population of implementation parameter descriptor
- Enhanced diagnostics and row and parameter status arrays
- Datetime, interval, numeric/decimal, and 64-bit integer application buffer types
- Asynchronous execution
- Stored procedure support, including escape sequences, output parameter binding mechanisms, and catalog functions
- Connection enhancements including support for connection attributes and attribute browsing

## Chapter 2: An Introduction to SQL and ODBC

ODBC was created to provide a uniform method of access to different, or heterogeneous, database management systems. This chapter discusses some of the concepts and history behind the development of ODBC, including:

- SQL and the various ways that applications use it.
- How network database access is done in the real world and what parts of this process can easily be standardized.
- Strategies used by ODBC and why those strategies were chosen.

### Structured Query Language (SQL)

A typical DBMS allows users to store, access, and modify data in an organized, efficient way. Originally, the users of DBMSs were programmers. Accessing the stored data required writing a program in a programming language such as COBOL. While these programs were often written to present a friendly

interface to a nontechnical user, access to the data itself required the services of a knowledgeable programmer. Casual access to the data was not practical.

Users were not entirely happy with this situation. While they could access data, it often required convincing a DBMS programmer to write special software. For example, if a sales department wanted to see the total sales in the previous month by each of its salespeople and wanted this information ranked in order by each salesperson's length of service in the company, it had two choices: Either a program already existed that allowed the information to be accessed in exactly this way, or the department had to ask a programmer to write such a program. In many cases, this was more work than it was worth, and it was always an expensive solution for one-time, or ad hoc, inquiries. As more and more users wanted easy access, this problem grew larger and larger.

Allowing users to access data on an ad hoc basis required giving them a language in which to express their requests. A single request to a database is defined as a query; such a language is called a query language. Many query languages were developed for this purpose, but one of these became the most popular: Structured Query Language, invented at IBM in the 1970s. It is more commonly known by its acronym, SQL, and is pronounced both as "ess-cue-ell" and as "sequel"; this manual uses the former pronunciation. SQL became an ANSI standard in 1986 and an ISO standard in 1987; it is used today in a great many database management systems.

Although SQL solved the ad hoc needs of users, the need for data access by computer programs did not go away. In fact, most database access still was (and is) programmatic, in the form of regularly scheduled reports and statistical analyses, data entry programs such as those used for order entry, and data manipulation programs, such as those used to reconcile accounts and generate work orders.

These programs also use SQL, using one of the following three techniques:

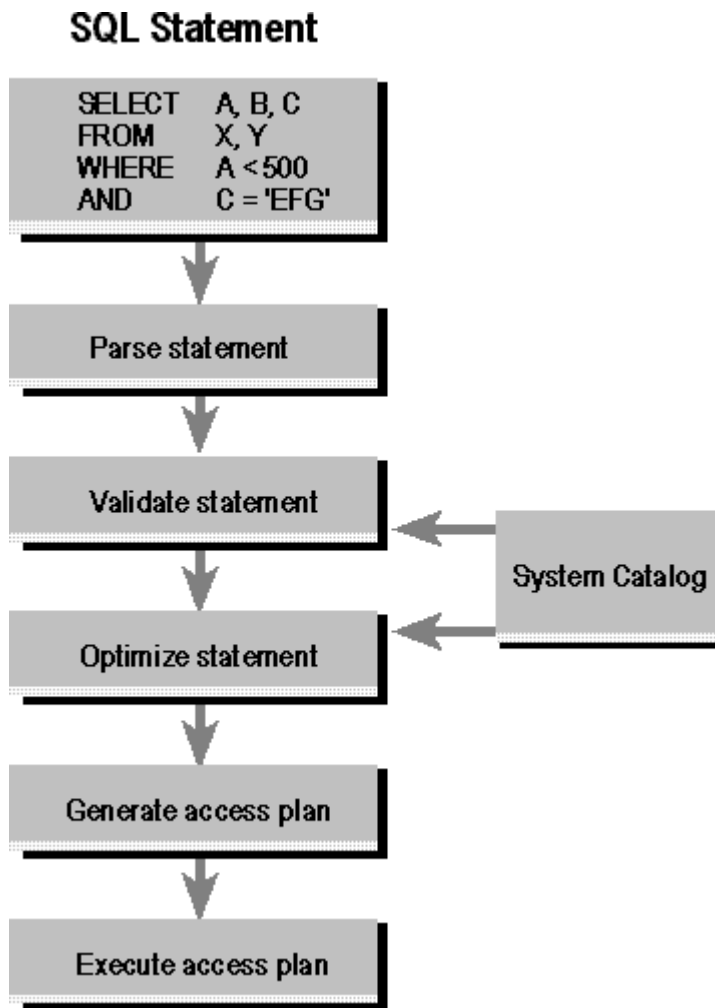
- **Embedded SQL**, in which SQL statements are embedded in a host language such as C or COBOL.
- **SQL Modules**, in which SQL statements are compiled on the DBMS and called from a host language.
- **Call-Level Interface**, or CLI, which consists of functions called to pass SQL statements to the DBMS and to retrieve results from the DBMS.

**Note** It is a historical accident that the term Call-Level Interface is used instead of Application Programming Interface (API), another term for the same thing. In the database world, API is used to describe SQL itself: SQL is the API to a DBMS.

Of these choices, embedded SQL is the most commonly used, although most major DBMSs support proprietary CLIs.

## Processing an SQL Statement

Before discussing the techniques for using SQL programmatically, it is necessary to discuss how an SQL statement is processed. The steps involved are common to all three techniques, although each technique performs them at different times. The following illustration shows the steps involved in processing an SQL statement, which are discussed throughout the rest of this section.



To process an SQL statement, a DBMS performs the following five steps:

- The DBMS first parses the SQL statement. It breaks the statement up into individual words, called tokens, makes sure that the statement has a valid verb and valid clauses, and so on. Syntax errors and misspellings can be detected in this step.
- The DBMS validates the statement. It checks the statement against the system catalog. Do all the tables named in the statement exist in the database? Do all of the columns exist and are the column names unambiguous? Does the user have the required privileges to execute the statement? Certain semantic errors can be detected in this step.
- The DBMS generates an access plan for the statement. The access plan is a binary representation of the steps that are required to carry out the statement; it is the DBMS equivalent of executable code.

- The DBMS optimizes the access plan. It explores various ways to carry out the access plan. Can an index be used to speed a search? Should the DBMS first apply a search condition to Table A and then join it to Table B, or should it begin with the join and use the search condition afterward? Can a sequential search through a table be avoided or reduced to a subset of the table? After exploring the alternatives, the DBMS chooses one of them.
- The DBMS executes the statement by running the access plan.

The steps used to process an SQL statement vary in the amount of database access they require and the amount of time they take. Parsing an SQL statement does not require access to the database and can be done very quickly. Optimization, on the other hand, is a very CPU-intensive process and requires access to the system catalog. For a complex, multitable query, the optimizer may explore thousands of different ways of carrying out the same query. However, the cost of executing the query inefficiently is usually so high that the time spent in optimization is more than regained in increased query execution speed. This is even more significant if the same optimized access plan can be used over and over to perform repetitive queries.

## Dynamic SQL

Although static SQL works well in many situations, there is a class of applications in which the data access cannot be determined in advance. For example, suppose a spreadsheet allows a user to enter a query, which the spreadsheet then sends to the DBMS to retrieve data. The contents of this query obviously cannot be known to the programmer when the spreadsheet program is written.

To solve this problem, the spreadsheet uses a form of embedded SQL called dynamic SQL. Unlike static SQL statements, which are hard-coded in the program, dynamic SQL statements can be built at run time and placed in a string host variable. They are then sent to the DBMS for processing. Because the DBMS must generate an access plan at run time for dynamic SQL statements, dynamic SQL is generally slower than static SQL. When a program containing dynamic SQL statements is compiled, the dynamic SQL statements are not stripped from the program, as in static SQL. Instead, they are replaced by a function call that passes the statement to the DBMS; static SQL statements in the same program are treated normally.

The simplest way to execute a dynamic SQL statement is with an EXECUTE IMMEDIATE statement. This statement passes the SQL statement to the DBMS for compilation and execution.

One disadvantage of the EXECUTE IMMEDIATE statement is that the DBMS must go through each of the five steps of processing an SQL statement each time the statement is executed. The overhead involved in this process can be significant if many statements are executed dynamically, and it is wasteful if those statements are similar. To address this situation, dynamic SQL offers an optimized form of execution called prepared execution, which uses the following steps:

The program constructs an SQL statement in a buffer, just as it does for the EXECUTE IMMEDIATE statement. Instead of host variables, a question mark (?) can be substituted for a constant anywhere in the statement text to indicate that a value for the constant will be supplied later. The question mark is called as a parameter marker.

The program passes the SQL statement to the DBMS with a PREPARE statement, which requests that the DBMS parse, validate, and optimize the statement and generate an execution plan for it. The program then uses an EXECUTE statement (not an EXECUTE IMMEDIATE statement) to execute the PREPARE

statement at a later time. It passes parameter values for the statement through a special data structure called the SQL Data Area or SQLDA.

The program can use the EXECUTE statement repeatedly, supplying different parameter values each time the dynamic statement is executed.

Prepared execution is still not the same as static SQL. In static SQL, the first four steps of processing an SQL statement take place at compile time. In prepared execution, these steps still take place at run time, but they are performed only once; execution of the plan takes place only when EXECUTE is called. This helps eliminate some of the performance disadvantages inherent in the architecture of dynamic SQL. The next illustration shows the differences between static SQL, dynamic SQL with immediate execution, and dynamic SQL with prepared execution.

## **Call-Level Interfaces**

The final technique for sending SQL statements to the DBMS is through a call-level interface (CLI). A call-level interface provides a library of DBMS functions that can be called by the application program. Thus, instead of trying to blend SQL with another programming language, a call-level interface is similar to the routine libraries most programmers are accustomed to using, such as the string, I/O, or math libraries in C. Note that DBMSs that support embedded SQL already have a call-level interface, the calls to which are generated by the precompiler. However, these calls are undocumented and subject to change without notice.

Call-level interfaces are commonly used in client/server architectures, in which the application program (the client) resides on one computer and the DBMS (the server) resides on a different computer. The application calls CLI functions on the local system, and those calls are sent across the network to the DBMS for processing.

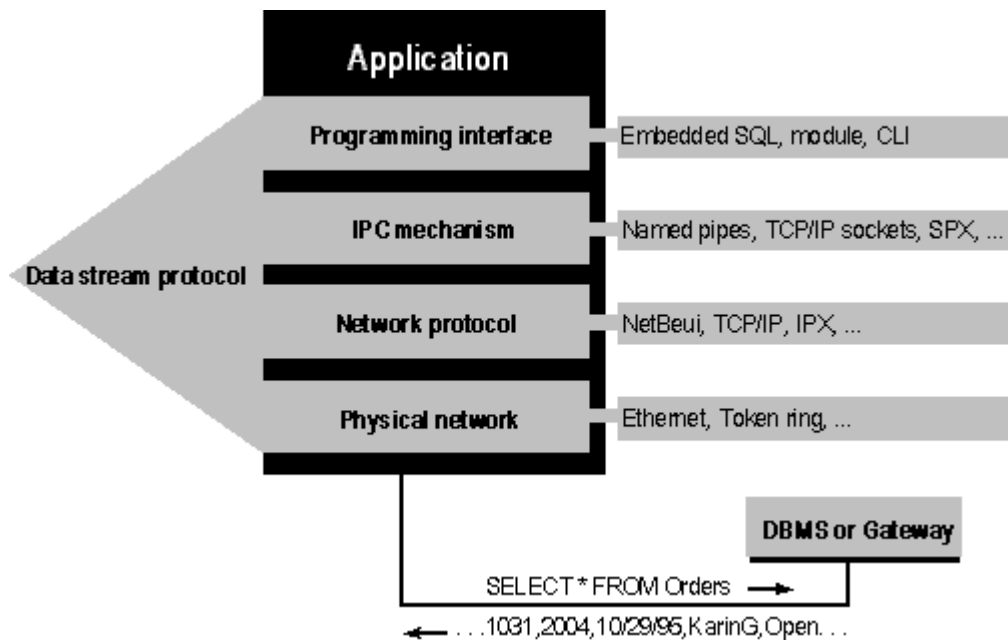
A call-level interface is similar to dynamic SQL, in that SQL statements are passed to the DBMS for processing at run time, but it differs from embedded SQL as a whole in that there are no embedded SQL statements and no precompiler is required.

Using a call-level interface typically involves the following steps:

- The application calls a CLI function to connect to the DBMS.
- The application builds an SQL statement and places it in a buffer. It then calls one or more CLI functions to send the statement to the DBMS for preparation and execution.
- If the statement is a SELECT statement, the application calls a CLI function to return the results in application buffers. Typically, this function returns one row or one column of data at a time.
- The application calls a CLI function to disconnect from the DBMS.

## **Network Database Access**

Accessing a database across a network requires a number of components, each of which is independent of, and resides beneath, the programming interface. These components are shown in the following illustration.



A further description of each component follows:

- **Programming Interface.** As described earlier in this chapter, the programming interface contains the calls made by the application. These interfaces (embedded SQL, SQL modules, and call-level interfaces) are generally specific to each DBMS, although they are usually based on an ANSI or ISO standard.
- **Data Stream Protocol.** The data stream protocol describes the stream of data transferred between the DBMS and its client. For example, the protocol might require the first byte to describe what the rest of the stream contains: an SQL statement to be executed, a returned error value, or returned data. The format of the rest of the data in the stream would then depend on this flag. For example, an error stream might contain the flag, a 2-byte integer error code, a 2-byte integer error message length, and an error message.

The data stream protocol is a logical protocol and is independent of the protocols used by the underlying network. Thus, a single data stream protocol can generally be used on a number of different networks. Data stream protocols are typically proprietary and have been optimized to work with a particular DBMS.

- **Interprocess Communication Mechanism.** The interprocess communication (IPC) mechanism is the process by which one process communicates with another. Examples include named pipes, TCP/IP sockets, and DECnet sockets. The choice of IPC mechanism is constrained by the operating system and network being used.
- **Network Protocol.** The network protocol is used to transport the data stream over a network. It can be considered the plumbing that supports the IPC mechanisms used to implement the data stream protocol, as well as supporting basic network operations such as file transfers and print

sharing. Network protocols include NetBEUI, TCP/IP, DECnet, and SPX/IPX and are specific to each network.

## The ODBC Solution

The question, then, is how does ODBC standardize database access? There are two architectural requirements:

- Applications must be able to access multiple DBMSs using the same source code without recompiling or relinking.
- Applications must be able to access multiple DBMSs simultaneously.

And there is one more question, due to marketplace reality:

Which DBMS features should ODBC expose? Only features that are common to all DBMSs or any feature that is available in any DBMS?

ODBC solves these problems in the following manner:

- **ODBC is a call-level interface.** To solve the problem of how applications access multiple DBMSs using the same source code, ODBC defines a standard CLI. This contains all of the functions in the CLI specifications from X/Open and ISO/IEC and provides additional functions commonly required by applications.

A different library, or driver, is required for each DBMS that supports ODBC. The driver implements the functions in the ODBC API. To use a different driver, the application does not need to be recompiled or relinked. Instead, the application simply loads the new driver and calls the functions in it. To access multiple DBMSs simultaneously, the application loads multiple drivers. How drivers are supported is operating system–specific. For example, on the Microsoft® Windows® operating system, drivers are dynamic-link libraries (DLLs).

- **ODBC defines a standard SQL grammar.** In addition to a standard call-level interface, ODBC defines a standard SQL grammar. This grammar is based on the X/Open SQL CAE specification. Differences between the two grammars are minor and primarily due to the differences between the SQL grammar required by embedded SQL (X/Open) and a CLI (ODBC). There are also some extensions to the grammar to expose commonly available language features not covered by the X/Open grammar.

Applications can submit statements using ODBC or DBMS-specific grammar. If a statement uses ODBC grammar that is different from DBMS-specific grammar, the driver converts it before sending it to the data source. However, such conversions are rare because most DBMSs already use standard SQL grammar.

- **ODBC provides a Driver Manager to manage simultaneous access to multiple DBMSs.** Although the use of drivers solves the problem of accessing multiple DBMSs simultaneously, the code to do this can be complex. Applications that are designed to work with all drivers cannot be statically linked to any drivers. Instead, they must load drivers at run time and call the functions in them through a table of function pointers. The situation becomes more complex if the application uses multiple drivers simultaneously.

Rather than forcing each application to do this, ODBC provides a Driver Manager. The Driver Manager implements all of the ODBC functions—mostly as pass-through calls to ODBC functions in drivers—and is statically linked to the application or loaded by the application at run time. Thus, the application calls ODBC functions by name in the Driver Manager, rather than by pointer in each driver.

When an application needs a particular driver, it first requests a connection handle with which to identify the driver and then requests that the Driver Manager load the driver. The Driver Manager loads the driver and stores the address of each function in the driver. To call an ODBC function in the driver, the application calls that function in the Driver Manager and passes the connection handle for the driver. The Driver Manager then calls the function by using the address it stored earlier.

- **ODBC exposes a significant number of DBMS features but does not require drivers to support all of them.** If ODBC exposed only features that are common to all DBMSs, it would be of little use; after all, the reason so many different DBMSs exist today is that they have different features. If ODBC exposed every feature that is available in any DBMS, it would be impossible for drivers to implement.

Instead, ODBC exposes a significant number of features—more than are supported by most DBMSs—but requires drivers to implement only a subset of those features. Drivers implement the remaining features only if they are supported by the underlying DBMS or if they choose to emulate them. Thus, applications can be written to exploit the features of a single DBMS as exposed by the driver for that DBMS, to use only those features used by all DBMSs, or to check for support of a particular feature and react accordingly.

So that an application can determine what features a driver and DBMS support, ODBC provides two functions (**SQLGetInfo** and **SQLGetFunctions**) that return general information about the driver and DBMS capabilities and a list of functions the driver supports. ODBC also defines API and SQL grammar conformance levels, which specify broad ranges of features supported by the driver. For more information, see "[Conformance Levels](#)" in Chapter 4, "ODBC Fundamentals."

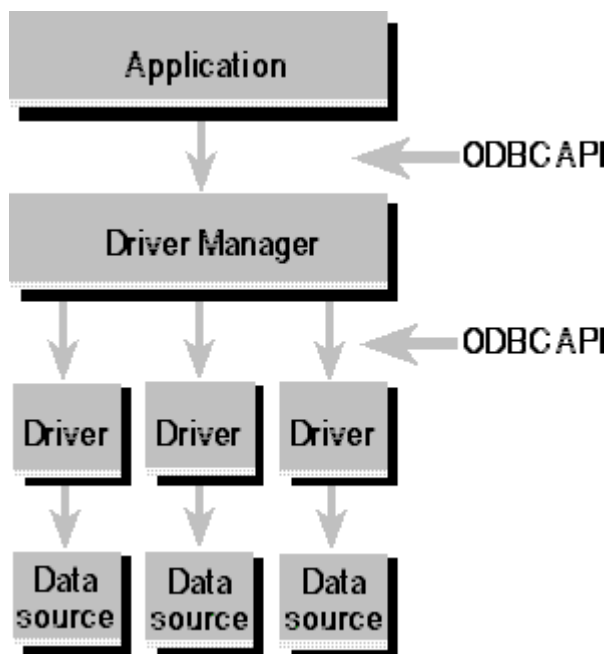
It is important to remember that ODBC defines a common interface for all of the features it exposes. Because of this, applications contain feature-specific code, not DBMS-specific code, and can use any drivers that expose those features. One advantage of this is that applications do not need to be updated when the features supported by a DBMS are enhanced; instead, when an updated driver is installed, the application automatically uses the features because its code is feature-specific, not driver-specific or DBMS-specific.



## Chapter 3: ODBC Architecture

The ODBC architecture has four components:

- **Application.** Performs processing and calls ODBC functions to submit SQL statements and retrieve results.
- **Driver Manager.** Loads and unloads drivers on behalf of an application. Processes ODBC function calls or passes them to a driver.
- **Driver.** Processes ODBC function calls, submits SQL requests to a specific data source, and returns results to the application. If necessary, the driver modifies an application's request so that the request conforms to syntax supported by the associated DBMS.
- **Data source.** Consists of the data the user wants to access and its associated operating system, DBMS, and network platform (if any) used to access the DBMS.
- The following illustration shows the relationship between these four components.



Note the following about this diagram. First, multiple drivers and data sources can exist, which allows the application to simultaneously access data from more than one data source. Second, the ODBC API is used in two places: between the application and the Driver Manager, and between the Driver Manager and each driver. The interface between the Driver Manager and the drivers is sometimes referred to as the *service provider interface*, or *SPI*. For ODBC, the application programming interface (API) and the service provider interface (SPI) are the same; that is, the Driver Manager and each driver have the same interface to the same functions.

## Applications

An *application* is a program that calls the ODBC API to access data. Although many types of applications are possible, most fall into three categories, which are used as examples throughout this guide.

- **Generic Applications.** These are also referred to as shrink-wrapped applications or off-the-shelf applications. Generic applications are designed to work with a variety of different DBMSs. Examples include a spreadsheet or statistics package that uses ODBC to import data for further analysis and a word processor that uses ODBC to get a mailing list from a database.

An important subcategory of generic applications is application development environments, such as PowerBuilder or Microsoft® Visual Basic®. Although the applications constructed with these environments will probably work only with a single DBMS, the environment itself needs to work with multiple DBMSs.

What all generic applications have in common is that they are highly interoperable among DBMSs and they need to use ODBC in a relatively generic manner. For more information about interoperability, see "[Choosing a Level of Interoperability](#)" in Chapter 16, "Interoperability."

- **Vertical Applications.** Vertical applications perform a single type of task, such as order entry or tracking manufacturing data, and work with a database schema that is controlled by the developer of the application. For a particular customer, the application works with a single DBMS. For example, a small business might use the application with dBase, while a large business might use it with Oracle.

The application uses ODBC in such a manner that the application is not tied to any one DBMS, although it might be tied to a limited number of DBMSs that provide similar functionality. Thus, the application developer can sell the application independently from the DBMS. Vertical applications are interoperable when they are developed but are sometimes modified to include noninteroperable code once the customer has chosen a DBMS.

- **Custom Applications.** Custom applications are used to perform a specific task in a single company. For example, an application in a large company might gather sales data from several divisions (each of which uses a different DBMS) and create a single report. ODBC is used because it is a common interface and saves programmers from having to learn multiple interfaces. Such applications are generally not interoperable and are written to specific DBMSs and drivers.

A number of tasks are common to all applications, no matter how they use ODBC. Taken together, they largely define the flow of any ODBC application. The tasks are:

- Selecting a data source and connecting to it.
- Submitting an SQL statement for execution.
- Retrieving results (if any).
- Processing errors.
- Committing or rolling back the transaction enclosing the SQL statement.

- Disconnecting from the data source.

Because most data access work is done with SQL, the primary task for which applications use ODBC is to submit SQL statements and retrieve the results (if any) generated by those statements. Other tasks for which applications use ODBC include determining and adjusting to driver capabilities and browsing the database catalog.

## The Driver Manager

The *Driver Manager* is a library that manages communication between applications and drivers. For example, on Microsoft® Windows® platforms, the Driver Manager is a dynamic-link library (DLL) that is written by Microsoft and can be redistributed by users of the Microsoft Data Access Software Development Kit (SDK).

The Driver Manager exists mainly as a convenience to application writers and solves a number of problems common to all applications. These include determining which driver to load based on a data source name, loading and unloading drivers, and calling functions in drivers.

To see why the latter is a problem, consider what would happen if the application called functions in the driver directly. Unless the application was linked directly to a particular driver, it would have to build a table of pointers to the functions in that driver and call those functions by pointer. Using the same code for more than one driver at a time would add yet another level of complexity. The application would first have to set a function pointer to point to the correct function in the correct driver, and then call the function through that pointer.

The Driver Manager solves this problem by providing a single place to call each function. The application is linked to the Driver Manager and calls ODBC functions in the Driver Manager, not the driver. The application identifies the target driver and data source with a *connection handle*. When it loads a driver, the Driver Manager builds a table of pointers to the functions in that driver. It uses the connection handle passed by the application to find the address of the function in the target driver and calls that function by address.

For the most part, the Driver Manager just passes function calls from the application to the correct driver. However, it also implements some functions (**SQLDataSources**, **SQLDrivers**, and **SQLGetFunctions**) and performs basic error checking. For example, the Driver Manager checks that handles are not null pointers, that functions are called in the correct order, and that certain function arguments are valid. For a complete description of the errors checked by the Driver Manager, see the reference section for each function and the “ODBC State Transition Tables” contained in the Part II Microsoft SDK PDF File available on the Solid Web site.

The final major role of the Driver Manager is loading and unloading drivers. The application loads and unloads only the Driver Manager. When it wants to use a particular driver, it calls a connection function (**SQLConnect**, **SQLDriverConnect**, or **SQLBrowseConnect**) in the Driver Manager and specifies the name of a particular data source or driver, such as "Accounting" or "SQL Server." Using this name, the Driver Manager searches the data source information for the driver's file name, such as `Sqldrver.dll`. It then loads the driver (assuming it is not already loaded), stores the address of each function in the driver, and calls the connection function in the driver, which then initializes itself and connects to the data source.

When the application is done using the driver, it calls **SQLDisconnect** in the Driver Manager. The Driver Manager calls this function in the driver, which disconnects from the data source. However, the Driver Manager keeps the driver in memory in case the application reconnects to it. It unloads the driver only when the application frees the connection used by the driver or uses the connection for a different driver, and no other connections use the driver. For a complete description of the Driver Manager's role in loading and unloading drivers, see "[Driver Manager's Role in the Connection Process](#)" in Chapter 6, "Connecting to a Data Source or Driver."

## Drivers

*Drivers* are libraries that implement the functions in the ODBC API. Each is specific to a particular DBMS; for example, a driver for Oracle cannot directly access data in an Informix DBMS. Drivers expose the capabilities of the underlying DBMSs; they are not required to implement capabilities not supported by the DBMS. For example, if the underlying DBMS does not support outer joins, then neither should the driver. The only major exception to this is that drivers for DBMSs that do not have stand-alone database engines, such as Xbase, must implement a database engine that at least supports a minimal amount of SQL.

## Driver Tasks

Specific tasks performed by drivers include:

- Connecting to and disconnecting from the data source.
- Checking for function errors not checked by the Driver Manager.
- Initiating transactions; this is transparent to the application.
- Submitting SQL statements to the data source for execution. The driver must modify ODBC SQL to DBMS-specific SQL; this is often limited to replacing escape clauses defined by ODBC with DBMS-specific SQL.
- Sending data to and retrieving data from the data source, including converting data types as specified by the application.
- Mapping DBMS-specific errors to ODBC SQLSTATEs.

## Driver Architecture

Driver architecture falls into two categories, depending on which software processes SQL statements:

- **File-based drivers.** The driver accesses the physical data directly. In this case, the driver acts as both driver and data source; that is, it processes ODBC calls and SQL statements. For example, dBASE drivers are file-based drivers because dBASE does not provide a stand-alone database engine the driver can use. It is important to note that developers of file-based drivers must write their own database engines.
- **DBMS-based drivers.** The driver accesses the physical data through a separate database engine. In this case the driver processes only ODBC calls; it passes SQL statements to the database engine for processing. For example, Oracle drivers are DBMS-based drivers because Oracle has a stand-

alone database engine the driver uses. Where the database engine resides is immaterial. It can reside on the same machine as the driver or a different machine on the network; it might even be accessed through a gateway.

Driver architecture is generally interesting only to driver writers; that is, driver architecture generally makes no difference to the application. However, the architecture can affect whether an application can use DBMS-specific SQL. For example, Microsoft Access provides a stand-alone database engine. If a Microsoft Access driver is DBMS-based—it accesses the data through this engine—the application can pass Microsoft Access–SQL statements to the engine for processing.

However, if the driver is file-based—that is, it contains a proprietary engine that accesses the Microsoft® Access .mdb file directly—any attempts to pass Microsoft Access–specific SQL statements to the engine are likely to result in syntax errors. The reason is that the proprietary engine is likely to implement only ODBC SQL.

## **DBMS-Based Drivers**

DBMS-based drivers are used with data sources such as Oracle or SQL Server that provide a stand-alone database engine for the driver to use. These drivers access the physical data through the stand-alone engine; that is, they submit SQL statements to and retrieve results from the engine.

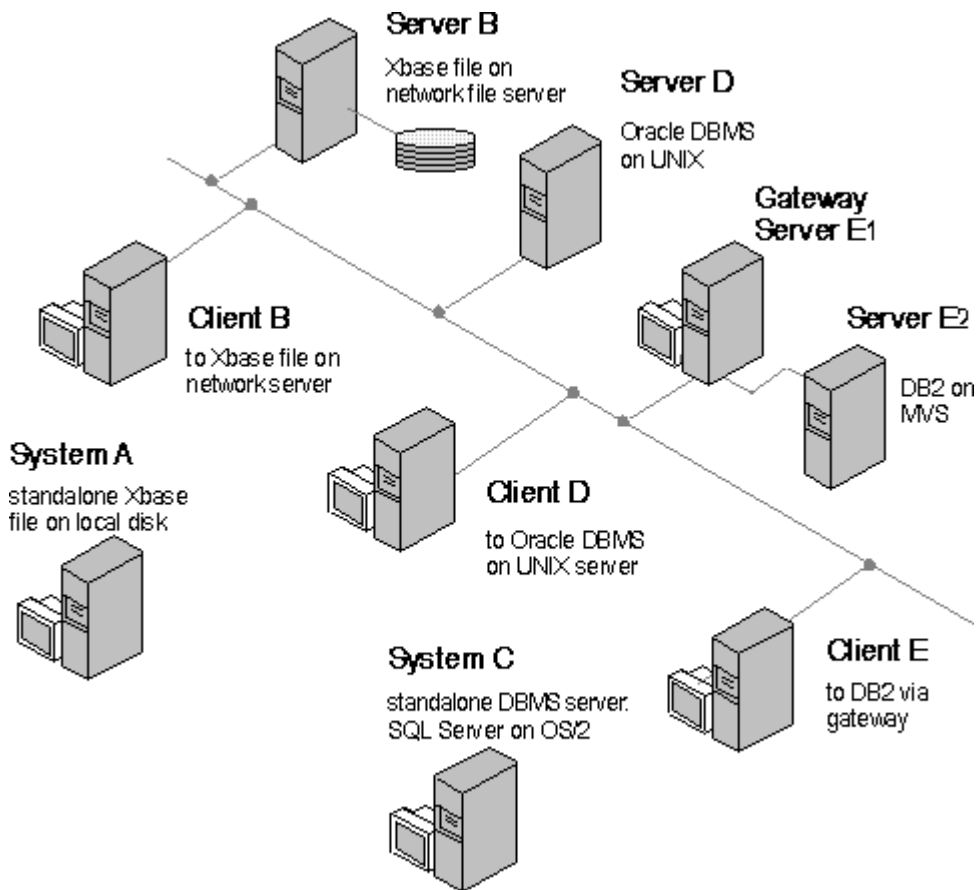
Because DBMS-based drivers use an existing database engine, they are usually easier to write than file-based drivers. Although a DBMS-based driver can be easily implemented by translating ODBC calls to native API calls, this results in a slower driver. A better way to implement a DBMS-based driver is to use the underlying data stream protocol, which is usually what the native API does. For example, a SQL Server driver should use TDS (the data stream protocol for SQL Server) rather than DB Library (the native API for SQL Server). An exception to this rule is when ODBC is the native API. For example, Watcom SQL is a stand-alone engine that resides on the same machine as the application and is loaded directly as the driver.

DBMS-based drivers act as the client in a client/server configuration where the data source acts as the server. In most cases, the client (driver) and server (data source) reside on different machines, although both could reside on the same machine running a multitasking operating system. A third possibility is a *gateway*, which sits between the driver and data source. A gateway is a piece of software that causes one DBMS to look like another. For example, applications written to use SQL Server can also access DB2 data through the Micro Decisionware DB2 Gateway; this product causes DB2 to look like SQL Server.

The following illustration shows three different configurations of DBMS-based drivers. In the first configuration, the driver and data source reside on the same machine. In the second, the driver and data source reside on different machines. In the third, the driver and data source reside on different machines and a gateway sits between them, residing on yet another machine.

## **Network Example**

This illustration shows how each of the preceding configurations could appear in a single network.



## Types of Data Sources

There are two types of data sources: machine data sources and file data sources. Although both contain similar information about the source of the data, they differ in the way this information is stored. Because of these differences, they are used in somewhat different manners.

### Machine Data Sources

*Machine data sources* are stored on the system with a user-defined name. Associated with the data source name is all of the information the Driver Manager and driver need to connect to the data source. For an Xbase data source, this might be the name of the Xbase driver, the full path of the directory containing the Xbase files, and some options that tell the driver how to use those files, such as single-user mode or read-only. For an Oracle data source, this might be the name of the Oracle driver, the server where the Oracle DBMS resides, the SQL\*Net connection string that identifies the SQL\*Net driver to use, and the system ID of the database on the server.

## Using Data Sources

Data sources usually are created by the end user or a technician with a program called the *ODBC Administrator*. The ODBC Administrator prompts the user for the driver to use and then calls that driver. The driver displays a dialog box that requests the information it needs to connect to the data source. After the user enters the information, the driver stores it on the system.

Later, the application calls the Driver Manager and passes it the name of a machine data source or the path of a file containing a file data source. When passed a machine data source name, the Driver Manager searches the system to find the driver used by the data source. It then loads the driver and passes the data source name to it. The driver uses the data source name to find the information it needs to connect to the data source. Finally, it connects to the data source, typically prompting the user for a user ID and password, which generally are not stored.

When passed a file data source, the Driver Manager opens the file and loads the specified driver. If the file also contains a connection string, it passes this to the driver. Using the information in the connection string, the driver connects to the data source. If no connection string was passed, the driver generally prompts the user for the necessary information.

## Data Source Example

On computers running Microsoft® Windows NT® Server/Windows 2000 Server, Microsoft Windows NT Workstation/Windows 2000 Professional, or Microsoft Windows® 95/98, machine data source information is stored in the registry. Depending on which registry key the information is stored under, the data source is known as a *user data source* or a *system data source*. User data sources are stored under the HKEY\_CURRENT\_USER key and are available only to the current user. System data sources are stored under the HKEY\_LOCAL\_MACHINE key and can be used by more than one user on one machine. They can also be used by systemwide services, which can then gain access to the data source even if no user is logged on to the machine. For more information about user and system data sources, see “SQLManageDataSources” in the “Installer DLL Function Reference” contained in the Part III PDF file available on the Solid Web site.

Suppose a user has three user data sources: Personnel and Inventory, which use an Oracle DBMS; and Payroll, which uses a Microsoft SQL Server DBMS. The registry values for data sources might be:

```
HKEY_CURRENT_USER
SOFTWARE
  ODBC
    Odbc.ini
      ODBC Data Sources
        Personnel : REG_SZ : Oracle
        Inventory : REG_SZ : Oracle
        Payroll : REG_SZ : SQL Server
```

and the registry values for the Payroll data source might be:

```
HKEY_CURRENT_USER
SOFTWARE
  ODBC
```

Odbc.ini

Payroll

Driver : REG\_SZ : C:\WINDOWS\SYSTEM\Sqlsrvr.dll

Description : REG\_SZ : Payroll database

Server : REG\_SZ : PYRLL1

FastConnectOption : REG\_SZ : No

UseProcForPrepare : REG\_SZ : Yes

OEMTOANSI : REG\_SZ : No

LastUser : REG\_SZ : smithjo

Database : REG\_SZ : Payroll

Language : REG\_SZ :



## Section 2: Developing Applications and Drivers

Part 2 of the *ODBC Programmer's Reference* contains information about developing applications that use the ODBC interface and drivers that implement it. This section contains the following chapters:

- [Chapter 4: ODBC Fundamentals](#)
- [Chapter 5: Overview of Basic ODBC Application Steps](#)
- [Chapter 6: Overview of Connecting to a Data Source or Driver](#)
- [Chapter 7: Overview of Catalog Functions](#)
- [Chapter 8: Overview of SQL Statements](#)
- [Chapter 9: Overview of Executing Statements](#)
- [Chapter 10: Overview of Retrieving Results \(Basic\)](#)
- [Chapter 11: Overview of Retrieving Results \(Advanced\)](#)
- [Chapter 12: Overview of Updating Data](#)
- [Chapter 13: Overview of Descriptors](#)
- [Chapter 14: Overview of Transactions](#)
- [Chapter 15: Overview of Diagnostics](#)
- [Chapter 16: Overview of Interoperability](#)
- [Chapter 17: Overview of Programming Considerations](#)

## Chapter 4: ODBC Fundamentals

This chapter covers a number of concepts fundamental to writing ODBC applications and drivers:

Handles

Buffers

Data types

Conformance levels

Environment, connection, and statement attributes

### Handles

Handles are opaque, 32-bit values that identify a particular item; in ODBC, this item can be an environment, connection, statement, or descriptor. When the application calls **SQLAllocHandle**, the Driver Manager or driver creates a new item of the specified type and returns its handle to the application. The application later uses the handle to identify that item when calling ODBC functions. The Driver Manager and driver use the handle to locate information about the item.

For example, the following code uses two statement handles (*hstmtOrder* and *hstmtLine*) to identify the statements on which to create result sets of sales orders and sales order line numbers. It later uses these handles to identify which result set to fetch data from.

```
SQLHSTMT      hstmtOrder, hstmtLine; // Statement handles.
SQLINTEGER    OrderID;
SQLINTEGER    OrderIDInd = 0;
SQLRETURN     rc;

// Prepare the statement that retrieves line number information.
SQLPrepare(hstmtLine, "SELECT * FROM Lines WHERE OrderID = ?", SQL_NTS);

// Bind OrderID to the parameter in the preceding statement.
SQLBindParameter(hstmtLine, 1, SQL_PARAM_INPUT, SQL_C_ULONG, SQL_INTEGER, 5, 0,
                 &OrderID, 0, &OrderIDInd);

// Bind the result sets for the Order table and the Lines table. Bind
// OrderID to the OrderID column in the Orders table. When each row is
// fetched, OrderID will contain the current order ID, which will then be
// passed as a parameter to the statement to fetch line number
// information. Code not shown.

// Create a result set of sales orders.
SQLExecDirect(hstmtOrder, "SELECT * FROM Orders", SQL_NTS);

// Fetch and display the sales order data. Code to check if rc equals
// SQL_ERROR or SQL_SUCCESS_WITH_INFO not shown.
while ((rc = SQLFetch(hstmtOrder)) != SQL_NO_DATA) {
    // Display the sales order data. Code not shown.

    // Create a result set of line numbers for the current sales order.
    SQLExecute(hstmtLine);

    // Fetch and display the sales order line number data. Code to check
```

```

    // if rc equals SQL_ERROR or SQL_SUCCESS_WITH_INFO not shown.
    while ((rc = SQLFetch(hstmtLine)) != SQL_NO_DATA) {
        // Display the sales order line number data. Code not shown.
    }

    // Close the sales order line number result set.
    SQLCloseCursor(hstmtLine);
}

// Close the sales order result set.
SQLCloseCursor(hstmtOrder);

```

Handles are meaningful only to the ODBC component that created them; that is, only the Driver Manager can interpret Driver Manager handles and only a driver can interpret its own handles.

For example, suppose the driver in the preceding example allocates a structure to store information about a statement and returns the pointer to this structure as the statement handle. When the application calls **SQLPrepare**, it passes an SQL statement and the handle of the statement used for sales order line numbers. The driver sends the SQL statement to the data source, which prepares it and returns an access plan identifier. The driver uses the handle to find the structure in which to store this identifier.

Later, when the application calls **SQLExecute** to generate the result set of line numbers for a particular sales order, it passes the same handle. The driver uses the handle to retrieve the access plan identifier from the structure. It sends the identifier to the data source to tell it which plan to execute.

ODBC has two levels of handles: Driver Manager handles and driver handles. The application uses Driver Manager handles when calling ODBC functions because it calls those functions in the Driver Manager. The Driver Manager uses this handle to find the corresponding driver handle and uses the driver handle when calling the function in the driver. For an example of how driver and Driver Manager handles are used, see "[Driver Manager's Role in the Connection Process](#)" in Chapter 6, "Connecting to a Data Source or Driver."

That there are two levels of handles is an artifact of the ODBC architecture; in most cases, it is not relevant to either the application or driver. Although there is usually no reason to do so, it is possible for the application to determine the driver handles by calling **SQLGetInfo**.

## Environment Handles

An *environment* is a global context in which to access data; associated with an environment is any information that is global in nature, such as:

- The environment's state
- The current environment-level diagnostics
- The handles of connections currently allocated on the environment
- The current settings of each environment attribute

Within a piece of code that implements ODBC (the Driver Manager or a driver), an environment handle identifies a structure to contain this information.

Environment handles are not frequently used in ODBC applications. They are always used in calls to **SQLDataSources** and **SQLDrivers** and sometimes used in calls to **SQLAllocHandle**, **SQLEndTran**, **SQLFreeHandle**, **SQLGetDiagField**, and **SQLGetDiagRec**.

Each piece of code that implements ODBC (the Driver Manager or a driver) contains one or more environment handles. For example, the Driver Manager maintains a separate environment handle for each application that is connected to it. Environment handles are allocated with **SQLAllocHandle** and freed with **SQLFreeHandle**.

## Connection Handles

A *connection* consists of a driver and a data source. A connection handle identifies each connection. The connection handle defines not only which driver to use, but which data source to use with that driver. Within a piece of code that implements ODBC (the Driver Manager or a driver), the connection handle identifies a structure that contains connection information, such as:

- The connection's state
- The current connection-level diagnostics
- The handles of statements and descriptors currently allocated on the connection
- The current settings of each connection attribute

ODBC does not prevent multiple simultaneous connections, if the driver supports them. Thus, in a particular ODBC environment, multiple connection handles might point to a variety of drivers and data sources, the same driver and a variety of data sources, or even multiple connections to the same driver and data source. Some drivers limit the number of active connections they support; the **SQL\_MAX\_DRIVER\_CONNECTIONS** option in **SQLGetInfo** specifies how many active connections a particular driver supports.

Connection handles are used primarily when connecting to the data source (**SQLConnect**, **SQLDriverConnect**, or **SQLBrowseConnect**), disconnecting from the data source (**SQLDisconnect**), getting information about the driver and data source (**SQLGetInfo**), retrieving diagnostics (**SQLGetDiagField** and **SQLGetDiagRec**) and performing transactions (**SQLEndTran**). They are also used when setting and getting connection attributes (**SQLSetConnectAttr** and **SQLGetConnectAttr**) and when getting the native format of an SQL statement (**SQLNativeSql**).

Connection handles are allocated with **SQLAllocHandle** and freed with **SQLFreeHandle**.

## Statement Handles

A *statement* is most easily thought of as an SQL statement, such as **SELECT \* FROM Employee**. However, a statement is more than just an SQL statement—it consists of all of the information associated with that SQL statement, such as any result sets created by the statement and parameters used in the execution of the statement. A statement does not even need to have an application-defined SQL statement. For example, when a catalog function such as **SQLTables** is executed on a statement, it executes a predefined SQL statement that returns a list of table names.

Each statement is identified by a statement handle. A statement is associated with a single connection, and there can be multiple statements on that connection. Some drivers limit the number of active statements they support; the `SQL_MAX_CONCURRENT_ACTIVITIES` option in **SQLGetInfo** specifies how many active statements a driver supports on a single connection. A statement is defined to be *active* if it has results pending, where results are either a result set or the count of rows affected by an **INSERT**, **UPDATE**, or **DELETE** statement, or data is being sent with multiple calls to **SQLPutData**.

Within a piece of code that implements ODBC (the Driver Manager or a driver), the statement handle identifies a structure that contains statement information, such as:

- The statement's state
- The current statement-level diagnostics
- The addresses of the application variables bound to the statement's parameters and result set columns
- The current settings of each statement attribute

Statement handles are used in most ODBC functions. Notably, they are used in the functions to bind parameters and result set columns (**SQLBindParameter** and **SQLBindCol**), prepare and execute statements (**SQLPrepare**, **SQLExecute**, and **SQLExecDirect**), retrieve metadata (**SQLColAttribute** and **SQLDescribeCol**), fetch results (**SQLFetch**), and retrieve diagnostics (**SQLGetDiagField** and **SQLGetDiagRec**). They are also used in catalog functions (**SQLColumns**, **SQLTables**, and so on) and a number of other functions.

Statement handles are allocated with **SQLAllocHandle** and freed with **SQLFreeHandle**.

## Descriptor Handles

A *descriptor* is a collection of metadata that describes the parameters of an SQL statement or the columns of a result set, as seen by the application or driver (also known as the *implementation*). Thus, a descriptor can fill any of four roles:

**Application Parameter Descriptor (APD)**. Contains information about the application buffers bound to the parameters in an SQL statement, such as their addresses, lengths, and C data types.

**Implementation Parameter Descriptor (IPD)**. Contains information about the parameters in an SQL statement, such as their SQL data types, lengths, and nullability.

**Application Row Descriptor (ARD)**. Contains information about the application buffers bound to the columns in a result set, such as their addresses, lengths, and C data types.

**Implementation Row Descriptor (IRD)**. Contains information about the columns in a result set, such as their SQL data types, lengths, and nullability.

Four descriptors (one filling each role) are allocated automatically when a statement is allocated. These are known as *automatically allocated descriptors* and are always associated with that statement. Applications can also allocate descriptors with **SQLAllocHandle**. These are known as *explicitly allocated descriptors*. They are allocated on a connection and can be associated with one or more statements on that connection to fulfill the role of an APD or ARD on those statements.

Most operations in ODBC can be performed without explicit use of descriptors by the application. However, descriptors provide a convenient shortcut for some operations. For example, suppose an application wants to insert data from two different sets of buffers. To use the first set of buffers, it would repeatedly call **SQLBindParameter** to bind them to the parameters in an **INSERT** statement, and then execute the statement. To use the second set of buffers, it would repeat this process. Alternatively, it could set up bindings to the first set of buffers in one descriptor and to the second set of buffers in another descriptor. To switch between the sets of bindings, it would simply call **SQLSetStmtAttr** and associate the correct descriptor with the statement as the APD.

For more information about descriptors, see “[Types of Descriptors](#)” in Chapter 13, “Descriptors.”

## State Transitions

ODBC defines discrete *states* for each environment, each connection, and each statement. For example, the environment has three possible states: Unallocated (in which no environment is allocated), Allocated (in which an environment is allocated but no connections are allocated), and Connection (in which an environment and one or more connections are allocated). Connections have 7 possible states; statements have 13 possible states.

A particular item, as identified by its handle, moves from one state to another when the application calls a certain function or functions and passes the handle to that item. Such movement is called a *state transition*. For example, allocating an environment handle with **SQLAllocHandle** moves the environment from Unallocated to Allocated, and freeing that handle with **SQLFreeHandle** returns it from Allocated to Unallocated. ODBC defines a limited number of legal state transitions, which is another way of saying that functions must be called in a certain order.

Some functions, such as **SQLGetConnectAttr**, do not affect state at all. Other functions affect the state of a single item. For example, **SQLDisconnect** moves a connection from a Connection state to an Allocated state. Finally, some functions affect the state of more than one item. For example, allocating a connection handle with **SQLAllocHandle** moves a connection from an Unallocated to an Allocated state and moves the environment from an Allocated to a Connection state.

If an application calls a function out of order, the function returns a *state transition error*. For example, if an environment is in a Connection state and the application calls **SQLFreeHandle** with that environment handle, **SQLFreeHandle** returns SQLSTATE HY010 (Function sequence error), because it can be called only when the environment is in an Allocated state. By defining this as an invalid state transition, ODBC prevents the application from freeing the environment while there are active connections.

Some state transitions are inherent in the design of ODBC. For example, it is not possible to allocate a connection handle without first allocating an environment handle, because the function that allocates a connection handle requires an environment handle. Other state transitions are enforced by the Driver Manager and the drivers. For example, **SQLExecute** executes a prepared statement. If the statement handle passed to it is not in a Prepared state, **SQLExecute** returns SQLSTATE HY010 (Function sequence error).

From the application's point of view, state transitions are usually straightforward: Legal state transitions tend to go hand-in-hand with the flow of a well-written application. State transitions are more complex for the Driver Manager and the drivers because they must track the state of the environment, each connection, and each statement. Most of this work is done by the Driver Manager; most of the work that must be done by drivers occurs with statements with pending results.

Parts 1 and 2 of this manual ("Introduction to ODBC" and "Developing Applications and Drivers") tend not to explicitly mention state transitions. Instead, they describe the order in which functions must be called. For example, Chapter 9, "Executing SQL Statements," states that a statement must be prepared with **SQLPrepare** before it can be executed with **SQLExecute**. For a complete description of states and state transitions, including which transitions are checked by the Driver Manager and which must be checked by drivers, see Appendix B, "ODBC State Transition Tables" contained on the Microsoft Web site (ODBC Programmer's Reference).

## Buffers

A buffer is any piece of application memory used to pass data between the application and the driver. For example, application buffers can be associated with, or *bound to*, result set columns with **SQLBindCol**. As each row is fetched, the data is returned for each column in these buffers. *Input buffers* are used to pass data from the application to the driver; *output buffers* are used to return data from the driver to the application.

**Note** If an ODBC function returns SQL\_ERROR, the contents of any output arguments to that function are undefined.

This discussion concerns itself primarily with buffers of indeterminate type. The addresses of these buffers appear as arguments of type SQLPOINTER, such as the *TargetValuePtr* argument in **SQLBindCol**. However, some of the items discussed here, such as the arguments used with buffers, also apply to arguments used to pass strings to the driver, such as the *TableName* argument in **SQLTables**.

These buffers usually come in pairs. *Data buffers* are used to pass the data itself, while *length/indicator buffers* are used to pass the length of the data in the data buffer or a special value such as SQL\_NULL\_DATA, which indicates that the data is NULL. The length of the data in a data buffer is different from the length of the data buffer itself. The following illustration shows the relationship between the data buffer and the length/indicator buffer.

A length/indicator buffer is required whenever the data buffer contains variable-length data, such as character or binary data. If the data buffer contains fixed-length data, such as an integer or date structure, a length/indicator buffer is needed only to pass indicator values because the length of the data is already known. If an application uses a length/indicator buffer with fixed-length data, the driver ignores any lengths passed in it.

The length of both the data buffer and the data it contains is measured in bytes, as opposed to characters. This distinction is unimportant for programs that use ANSI strings because lengths in bytes and characters are the same.

When the data buffer represents a driver-defined descriptor field, diagnostic field, or attribute, the application should indicate to the Driver Manager the nature of the function argument that indicates the value for the field or attribute. The application does this by setting the length argument in any function call that sets the field or attribute to one of the following values. (The same is true for functions that retrieve the values of the field or attribute, with the exception that the argument points to the values that for the setting function are in the argument itself.)

If the function argument that indicates the value for the field or attribute is a pointer to a character string, the *length* argument is the length of the string or SQL\_NTS.

If the function argument that indicates the value for the field or attribute is a pointer to a binary buffer, the application places the result of the `SQL_LEN_BINARY_ATTR(length)` macro in the *length* argument. This places a negative value in the *length* argument.

If the function argument that indicates the value for the field or attribute is a pointer to a value other than a character string or a binary string, the *length* argument should have the value `SQL_IS_POINTER`.

If the function argument that indicates the value for the field or attribute contains a fixed-length value, the *length* argument is `SQL_IS_INTEGER`, `SQL_IS_UNSIGNED_INTEGER`, `SQL_SMALLINT`, or `SQL_USMALLINT`, as appropriate.

## Deferred Buffers

A *deferred buffer* is one whose value is used at some time *after* it is specified in a function call. For example, **SQLBindParameter** is used to associate, or *bind*, a data buffer with a parameter in an SQL statement. The application specifies the number of the parameter and passes the address, byte length, and type of the buffer. The driver saves this information but does not examine the contents of the buffer. Later, when the application executes the statement, the driver retrieves the information and uses it to retrieve the parameter data and send it to the data source. Therefore, the input of data in the buffer is deferred. Because deferred buffers are specified in one function and used in another, it is an application programming error to free a deferred buffer while the driver still expects it to exist; for more information, see "

[Allocating and Freeing Buffers](#)," in the next section.

Both input and output buffers can be deferred. The following table summarizes the uses of deferred buffers. Note that deferred buffers bound to result set columns are specified with **SQLBindCol**, and deferred buffers bound to SQL statement parameters are specified with **SQLBindParameter**.

| Buffer use   | Type            | Specified with   | Used by                                 |
|--|-----------------|------------------|---|
| Sending data for input parameters                      | Deferred input  | SQLBindParameter | SQLExecute<br>SQLExecDirect             |
| Sending data to update or insert a row in a result set | Deferred input  | SQLBindCol       | SQLSetPos                               |
| Returning data for output and input/output parameters  | Deferred output | SQLBindParameter | SQLExecute<br>SQLExecDirect             |
| Returning result set data                              | Deferred output | SQLBindCol       | SQLFetch<br>SQLFetchScroll<br>SQLSetPos |

## Allocating and Freeing Buffers

All buffers are allocated and freed by the application. If a buffer is not deferred, it need only exist for the duration of the call to a function. For example, **SQLGetInfo** returns the value associated with a particular option in the buffer pointed to by the *InfoValuePtr* argument. This buffer can be freed immediately after the call to **SQLGetInfo**, as shown in the following code example:

```
SQLSMALLINT    InfoValueLen;
SQLCHAR *      InfoValuePtr = malloc(50);    // Allocate InfoValuePtr.
```



```
SQLGetInfo(hdbc, SQL_DBMS_NAME, (SQLPOINTER)InfoValuePtr, 50,
          &InfoValueLen);
```

```
free(InfoValuePtr); // OK to free InfoValuePtr.
```

Because deferred buffers are specified in one function and used in another, it is an application programming error to free a deferred buffer while the driver still expects it to exist. For example, the address of the *\*ValuePtr* buffer is passed to **SQLBindCol** for later use by **SQLFetch**. This buffer cannot be freed until the column is unbound, such as with a call to **SQLBindCol** or **SQLFreeStmt** as shown in the following code example:

```
SQLRETURN    rc;
SQLINTEGER    ValueLenOrInd;
SQLHSTMT    hstmt;

// Allocate ValuePtr
SQLCHAR * ValuePtr = malloc(50);

// Bind ValuePtr to column 1. It is an error to free ValuePtr here.
SQLBindCol(hstmt, 1, SQL_C_CHAR, ValuePtr, 50, &ValueLenOrInd);

// Fetch each row of data and place the value for column 1 in *ValuePtr.
// Code to check if rc equals SQL_ERROR or SQL_SUCCESS_WITH_INFO
// not shown.
while ((rc = SQLFetch(hstmt)) != SQL_NO_DATA) {
    // It is an error to free ValuePtr here.
}

// Unbind ValuePtr from column 1. It is now OK to free ValuePtr.
SQLFreeStmt(hstmt, SQL_UNBIND);
free(ValuePtr);
```

Such an error is easily made by declaring the buffer locally in a function; the buffer is freed when the application leaves the function. For example, the following code causes undefined and probably fatal behavior in the driver:

```
SQLRETURN    rc;
SQLHSTMT    hstmt;

BindAColumn(hstmt);

// Fetch each row of data and try to place the value for column 1 in
// *ValuePtr. Because ValuePtr has been freed, the behavior is undefined
// and probably fatal. Code to check if rc equals SQL_ERROR or
// SQL_SUCCESS_WITH_INFO not shown.
while ((rc = SQLFetch(hstmt)) != SQL_NO_DATA) {}

.
.
.

void BindAColumn(SQLHSTMT hstmt) // WARNING! This function won't work!
{
    // Declare ValuePtr locally.
```

```

SQLCHAR      ValuePtr[50];
SQLINTEGER   ValueLenOrInd;

// Bind rgbValue to column.
SQLBindCol(hstmt, 1, SQL_C_CHAR, ValuePtr, sizeof(ValuePtr),
           &ValueLenOrInd);

// ValuePtr is freed when BindAColumn exits.
}

```

## Using Data Buffers

Data buffers are described by three pieces of information: their type, address, and byte length. Whenever a function needs one of these pieces of information and does not already know it, it has an argument with which the application passes it.

## Data Buffer Type

The C data type of a buffer is specified by the application. In the case of a single variable, this occurs when the application allocates the variable. In the case of generic memory—that is, memory pointed to by a pointer of type `void`—this occurs when the application casts the memory to a particular type. The driver discovers this type in two ways:

**Data buffer type argument.** Buffers used to transfer parameter values and result set data, such as the buffer bound with *TargetValuePtr* in **SQLBindCol**, usually have an associated type argument, such as the *TargetType* argument in **SQLBindCol**. In this argument, the application passes the C type identifier corresponding to the type of the buffer. For example, in the following call to **SQLBindCol**, the value `SQL_C_TYPE_DATE` tells the driver that the *Date* buffer is an `SQL_DATE_STRUCT`:

```

SQL_DATE_STRUCT Date;
SQLINTEGER DateInd;
SQLBindCol(hstmt, 1, SQL_C_TYPE_DATE, &Date, 0, &DateInd);

```

For more information on type identifiers, see the "Data Types in ODBC" section, later in this chapter.

**Predefined type.** Buffers used to send and retrieve options or attributes, such as the buffer pointed to by the *InfoValuePtr* argument in **SQLGetInfo**, have a fixed type that depends on the option specified. The driver assumes that the data buffer is of this type; it is the application's responsibility to allocate a buffer of this type. For example, in the following call to **SQLGetInfo**, the driver assumes the buffer is a 32-bit integer because this is what the `SQL_STRING_FUNCTIONS` option requires:

- `SQLINTEGER StringFuncs;`
- `SQLGetInfo(hdbc, SQL_STRING_FUNCTIONS, (SQLPOINTER) &StringFuncs, 0,`
- `NULL);`

The driver uses the C data type to interpret the data in the buffer.

## Data Buffer Address

The application passes the address of the data buffer to the driver in an argument, often named *ValuePtr* or a similar name. For example, in the following call to **SQLBindCol**, the application specifies the address of the *Date* variable:

```

SQL_DATE_STRUCT Date;

```

```
SQLINTEGER DateInd;
SQLBindCol(hstmt, 1, SQL_C_TYPE_DATE, &dsDate, 0, &DateInd);
As mentioned in the “
```

Allocating and Freeing Buffers” section in this chapter, the address of a deferred buffer must remain valid until the buffer is unbound.

Unless it is specifically prohibited, the address of a data buffer can be a null pointer. For buffers used to send data to the driver, this causes the driver to ignore the information normally contained in the buffer. For buffers used to retrieve data from the driver, this causes the driver to not return a value. In both cases, the driver ignores the corresponding data buffer length argument.

## Data Buffer Length

The application passes the byte length of the data buffer to the driver in an argument, named *BufferLength* or a similar name. For example, in the following call to **SQLBindCol**, the application specifies the length of the *ValuePtr* buffer (**sizeof(ValuePtr)**):

```
SQLCHAR      ValuePtr[50];
SQLINTEGER    ValueLenOrInd;
SQLBindCol(hstmt, 1, SQL_C_CHAR, ValuePtr, sizeof(ValuePtr), &ValueLenOrInd);
```

A driver will always return the number of bytes, not the number of characters, in the buffer length argument of any function that has an output string argument.

Data buffer lengths are required only for output buffers; the driver uses them to avoid writing past the end of the buffer. However, the driver checks the data buffer length only when the buffer contains variable-length data, such as character or binary data. If the buffer contains fixed-length data, such as an integer or date structure, the driver ignores the data buffer length and assumes the buffer is large enough to hold the data; that is, it never truncates fixed-length data. It is therefore important for the application to allocate a large enough buffer for fixed-length data.

When a truncation of non-data output strings occurs (such as the cursor name returned for **SQLGetCursorName**), the returned length in the buffer length argument is the maximum byte length possible. That means that it is twice the number of characters, because the string could contain DBCS characters. For example, if the cursor name is 10 ANSI characters long (10 bytes) and an application calls **SQLGetCursorName** to try to retrieve the cursor name into a buffer that is 6 bytes long, the value returned in *\*NameLengthPtr* will be 20.

Data buffer lengths are not required for input buffers because the driver does not write to these buffers.

## Using Length/Indicator Values

The length/indicator buffer is used to pass the byte length of the data in the data buffer or a special indicator such as **SQL\_NULL\_DATA**, which indicates that the data is NULL. Depending on the function in which it is used, a length/indicator buffer is defined to be an **SQLINTEGER** or an **SQLSMALLINT**. Therefore, a single argument is needed to describe it. If the data buffer is a nondeferred input buffer, this argument contains the byte length of the data itself or an indicator value. It is often named *StrLen\_or\_Ind* or a similar name. For example, the following code calls **SQLPutData** to pass a buffer full of data; the byte length (*ValueLen*) is passed directly because the data buffer (*ValuePtr*) is an input buffer.

```
SQLCHAR      ValuePtr[50];
```

```
SQLINTEGER    ValueLen;
```

```
// Call local function to place data in ValuePtr. In ValueLen, return the
// number of bytes of data placed in ValuePtr. If there is not enough
// data, this will be less than 50.
```

```
FillBuffer(ValuePtr, sizeof(ValuePtr), &ValueLen);
```

```
// Call SQLPutData to send the data to the driver.
```

```
SQLPutData(hstmt, ValuePtr, ValueLen);
```

If the data buffer is a deferred input buffer, a nondeferred output buffer, or an output buffer, the argument contains the address of the length/indicator buffer. It is often named *StrLen\_or\_IndPtr* or a similar name. For example, the following code calls **SQLGetData** to retrieve a buffer full of data; the byte length is returned to the application in the length/indicator buffer (*ValueLenOrInd*), whose address is passed to **SQLGetData** because the corresponding data buffer (*ValuePtr*) is a nondeferred output buffer.

```
SQLCHAR        ValuePtr[50];
```

```
SQLINTEGER    ValueLenOrInd;
```

```
SQLGetData(hstmt, 1, SQL_C_CHAR, ValuePtr, sizeof(ValuePtr), &ValueLenOrInd);
```

Unless it is specifically prohibited, a length/indicator buffer argument can be 0 (if nondeferred input) or a null pointer (if output or deferred input). For input buffers, this causes the driver to ignore the byte length of the data. This returns an error when passing variable-length data but is common when passing non-null, fixed-length data, because neither a length nor an indicator value is needed. For output buffers, this causes the driver to not return the byte length of the data or an indicator value. This is an error if the data returned by the driver is NULL but is common when retrieving fixed-length, non-nullable data, because neither a length nor an indicator value is needed.

As when the address of a deferred data buffer is passed to the driver, the address of a deferred length/indicator buffer must remain valid until the buffer is unbound.

The following lengths are valid as length/indicator values:

- $n$ , where  $n > 0$ .
- 0.
- **SQL\_NTS**. A string sent to the driver in the corresponding data buffer is null-terminated; this is a convenient way for C programmers to pass strings without having to calculate their byte length. This value is legal only when the application sends data to the driver. When the driver returns data to the application, it always returns the actual byte length of the data.

The following values are valid as length/indicator values. **SQL\_NULL\_DATA** is stored in the **SQL\_DESC\_INDICATOR\_PTR** descriptor field; all other values are stored in the **SQL\_DESC\_OCTET\_LENGTH\_PTR** descriptor field.

**SQL\_NULL\_DATA**. The data is a NULL data value, and the value in the corresponding data buffer is ignored. This value is legal only for SQL data sent to or retrieved from the driver.

**SQL\_DATA\_AT\_EXEC**. The data buffer does not contain any data. Instead, the data will be sent with **SQLPutData** when the statement is executed or when **SQLBulkOperations** or **SQLSetPos** is called. This value is legal only for SQL data sent to the driver. For more information, see **SQLBindParameter**,

SQLBulkOperations, and SQLSetPos in the Part II PDF file, "ODBC API Reference" available on the Solid Web site.

Result of the SQL\_LEN\_DATA\_AT\_EXEC(*length*) macro. This value is similar to SQL\_DATA\_AT\_EXEC. For more information, see "[Sending Long Data](#)" in Chapter 9, "Executing Statements."

SQL\_NO\_TOTAL. The driver cannot determine the number of bytes of long data still available to return in an output buffer. This value is legal only for SQL data retrieved from the driver.

SQL\_DEFAULT\_PARAM. A procedure is to use the default value of an input parameter in a procedure instead of the value in the corresponding data buffer.

SQL\_COLUMN\_IGNORE. **SQLBulkOperations** or **SQLSetPos** is to ignore the value in the data buffer. When updating a row of data by a call to **SQLBulkOperations** or **SQLSetPos**, the column value is not changed. When inserting a new row of data by a call to **SQLBulkOperations**, the column value is set to its default or, if the column does not have a default, to NULL.

## Data Length, Buffer Length, and Truncation

The *data length* is the byte length of the data as it would be stored in the application's data buffer, not as it is stored in the data source. This distinction is important because the data is often stored in different types in the data buffer and in the data source. Thus, for data being sent to the data source, this is the byte length of the data before conversion to the data source's type. For data being retrieved from the data source, this is the byte length of the data after conversion to the data buffer's type and before any truncation is done.

For fixed-length data, such as an integer or a date structure, the byte length of the data is always the size of the data type. In general, applications allocate a data buffer that is the size of the data type. If the application allocates a smaller buffer, the consequences are undefined because the driver assumes the data buffer is the size of the data type and does not truncate the data to fit into a smaller buffer. If the application allocates a larger buffer, the extra space is never used.

For variable-length data, such as character or binary data, it is important to recognize that the byte length of the data is separate from and often different than the byte length of the buffer. The relation of these two lengths is shown in the "Buffers" section, earlier in this chapter. If the byte length of the data is greater than the byte length of the buffer, the driver truncates data being fetched to the byte length of the buffer and returns SQL\_SUCCESS\_WITH\_INFO with SQLSTATE 01004 (Data truncated). However, the returned byte length is the length of the untruncated data.

For example, suppose an application allocates 50 bytes for a binary data buffer. If the driver has 10 bytes of binary data to return, it returns those 10 bytes in the buffer. The byte length of the data is 10, and the byte length of the buffer is 50. If the driver has 60 bytes of binary data to return, it truncates the data to 50 bytes, returns those bytes in the buffer, and returns SQL\_SUCCESS\_WITH\_INFO. The byte length of the data is 60 (the length before truncation), and the byte length of the buffer is still 50.

A diagnostic record is created for each column that is truncated. Because it takes time for the driver to create these records and for the application to process them, truncation can degrade performance. Usually, an application can avoid this problem by allocating large enough buffers, although this might not be

possible when working with long data. When data truncation occurs, the application can sometimes allocate a larger buffer and refetch the data; this is not true in all cases. If truncation occurs while getting data with calls to **SQLGetData**, the application need not call **SQLGetData** for data that has already been returned; for more information, see “[Getting Long Data](#)” in Chapter 10, “Retrieving Results (Basic).”

## Character Data and C Strings

Input parameters that refer to variable-length character data (such as column names, dynamic parameters, and string attribute values) have an associated length parameter. If the application terminates strings with the null character, as is typical in C, it provides as an argument either the length in bytes of the string (not including the null-terminator) or **SQL\_NTS** (Null-Terminated String). A non-negative length argument specifies the actual length of the associated string. The length argument may be 0 to specify a zero-length string, which is distinct from a **NULL** value. The negative value **SQL\_NTS** directs the driver to determine the length of the string by locating the null-termination character.

When character data is returned from the driver to the application, the driver must always null-terminate it. This gives the application the choice of whether to handle the data as a string or a character array. If the application buffer is not large enough to return all of the character data, the driver truncates it to the byte length of the buffer less the number of bytes required by the null-termination character, null-terminates the truncated data, and stores it in the buffer. Therefore, applications must always allocate extra space for the null-termination character in buffers used to retrieve character data. For example, a 51-byte buffer is needed to retrieve 50 characters of data.

Special care must be taken by both the application and the driver when sending or retrieving long character data in parts with **SQLPutData** or **SQLGetData**. If the data is passed as a series of null-terminated strings, the null-termination characters on these strings must be stripped before the data can be reassembled.

A number of ODBC programmers have confused character data and C strings. That this has occurred is an artifact of using the C language when defining ODBC functions. If an ODBC driver or application uses another language—remember that ODBC is language-independent—this confusion is less likely to arise.

When C strings are used to hold character data, the null-termination character is not considered to be part of the data and is not counted as part of its byte length. For example, the character data "ABC" can be held as the C string "ABC\0" or the character array {'A', 'B', 'C'}. The byte length of the data is 3, whether or not it is treated as a string or a character array.

Although applications and drivers commonly use C strings (null-terminated arrays of characters) to hold character data, there is no requirement to do this. In C, character data can also be treated as an array of characters (without null-termination) and its byte length passed separately in the length/indicator buffer.

Because character data can be held in a non-null-terminated array and its byte length passed separately, it is possible to embed null characters in character data. However, the behavior of ODBC functions in this case is undefined and it is driver-specific whether a driver handles this correctly. Thus, interoperable applications should always handle character data that can contain embedded null characters as binary data.

## Data Types in ODBC

ODBC uses two sets of data types: SQL data types and C data types. SQL data types are used in the data source and C data types are used in C code in the application.

### Type Identifiers

To describe SQL and C data types, ODBC defines two sets of *type identifiers*. A type identifier describes the type of an SQL column or a C buffer. It is a **#define** value and is generally passed as a function argument or returned in metadata. For example, the following call to **SQLBindParameter** binds a variable of type `SQL_DATE_STRUCT` to a date parameter in an SQL statement. The C type identifier `SQL_C_TYPE_DATE` specifies the type of the *Date* variable, and the SQL type identifier `SQL_TYPE_DATE` specifies the type of the dynamic parameter.

```
SQL_DATE_STRUCT Date;
SQLINTEGER DateInd = 0;
SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_TYPE_DATE, SQL_TYPE_DATE, 0,
0,
                &Date, 0, &DateInd);
```

## SQL Data Types in ODBC

SQL data types are the types in which data is stored in the data source.

### SQL Type Identifiers

Each data source defines its own SQL data types. ODBC defines type identifiers and describes the general characteristics of the SQL data types that might be mapped to each type identifier. It is driver-specific how each data type in the underlying data source is mapped to an SQL type identifier of ODBC.

For example, `SQL_CHAR` is the type identifier for a character column with a fixed length, typically between 1 and 254 characters. These characteristics correspond to the `CHAR` data type found in many SQL data sources. Thus, when an application discovers that the type identifier for a column is `SQL_CHAR`, it can assume it is probably dealing with a `CHAR` column. However, it should still check the byte length of the column before assuming it is between 1 and 254 characters; the driver for a non-SQL data source, for example, might map a fixed-length character column of 500 characters to `SQL_CHAR` or `SQL_LONGVARCHAR`, since neither is an exact match.

ODBC defines a wide variety of SQL type identifiers. However, the driver is not required to use all of these identifiers. Instead, it uses only those identifiers it needs to expose the SQL data types supported by the underlying data source. If the underlying data source supports SQL data types to which no type identifier corresponds, the driver can define additional type identifiers. For more information, see [“Driver-Specific Data Types, Descriptor Types, Information Types, Diagnostic Types, and Attributes,”](#) in Chapter 17, “Programming Considerations.”

For a complete description of SQL type identifiers, see Appendix D, “Data Types” in the **SOLID Programmer Guide**.

## ***Retrieving Data Type Information with SQLGetTypeInfo***

Because the mappings from underlying SQL data types to ODBC type identifiers are approximate, ODBC provides a function (**SQLGetTypeInfo**) through which a driver can completely describe each SQL data type in the data source. This function returns a result set, each row of which describes the characteristics of a single data type, such as name, type identifier, precision, scale, and nullability.

This information generally is used by generic applications that allow the user to create and alter tables. Such applications call **SQLGetTypeInfo** to retrieve the data type information and then present some or all of it to the user. Such applications need to be aware of two things:

More than one SQL data type can map to a single type identifier, which can make it difficult to determine which data type to use. To solve this, the result set is ordered first by type identifier and second by closeness to the type identifier's definition. In addition, data source-defined data types take precedence over user-defined data types. For example, suppose that a data source defines the **INTEGER** and **COUNTER** data types to be the same except that **COUNTER** is auto-incrementing. Suppose also that the user-defined type **WHOLENUM** is a synonym of **INTEGER**. Each of these types maps to **SQL\_INTEGER**. In the **SQLGetTypeInfo** result set, **INTEGER** appears first, followed by **WHOLENUM** and then **COUNTER**. **WHOLENUM** appears after **INTEGER** because it is user-defined but before **COUNTER** because it more closely matches the definition of the **SQL\_INTEGER** type identifier.

ODBC does not define data type names for use in **CREATE TABLE** and **ALTER TABLE** statements. Instead, the application should use the name returned in the **TYPE\_NAME** column of the result set returned by **SQLGetTypeInfo**. The reason for this is that although most of SQL does not vary much across DBMSs, data type names vary tremendously. Rather than forcing drivers to parse SQL statements and replace standard data type names with DBMS-specific data type names, ODBC requires applications to use the DBMS-specific names in the first place.

Note that **SQLGetTypeInfo** does not necessarily describe all of the data types an application can encounter. In particular, result sets might contain data types not directly supported by the data source. For example, the data types of the columns in result sets returned by catalog functions are defined by ODBC and these data types might not be supported by the data source. To determine the characteristics of the data types in a result set, an application calls **SQLColAttribute**.

## **C Data Types in ODBC**

ODBC defines the C data types that are used by application variables and their corresponding type identifiers. Among other things, these are used by the buffers that are bound to result set columns and statement parameters. For example, suppose an application wants to retrieve data from a result set column in character format. It declares a variable with the **SQLCHAR \*** data type and binds this variable to the result set column with a type identifier of **SQL\_C\_CHAR**. For a complete list of C data types and type identifiers, see Appendix D, "Data Types" in the **SOLID Programmer Guide**.

ODBC also defines a default mapping from each SQL data type to a C data type. For example, a 2-byte integer in the data source is mapped to a 2-byte integer in the application. To use the default mapping, an application specifies the **SQL\_C\_DEFAULT** type identifier. However, use of this identifier is discouraged for interoperability reasons.



All integer C data types defined in ODBC 1.x were signed. Unsigned C data types and their corresponding type identifiers were added in ODBC 2.0. Because of this, applications and drivers need to be particularly careful when dealing with 1.x versions.

## Data Type Conversions

Data can be converted from one type to another at one of four times: when data is transferred from one application variable to another (C to C), when data in an application variable is sent to a statement parameter (C to SQL), when data in a result set column is returned in an application variable (SQL to C), and when data is transferred from one data source column to another (SQL to SQL).

Any conversion that occurs when data is transferred from one application variable to another is outside the scope of this document.

When an application binds a variable to a result set column or statement parameter, it implicitly specifies a data type conversion in its choice of the data type of the application variable. For example, suppose a column contains integer data. If the application binds an integer variable to the column, it specifies that no conversion be done; if it binds a character variable to the column, it specifies that the data be converted from integer to character.

ODBC defines how data is converted between each SQL and C data type. Basically, it supports all reasonable conversions, such as character to integer and integer to float, and does not support ill-defined conversions, such as float to date. Drivers are required to support all conversions for each SQL data type they support. For a complete list of conversions between SQL and C data types, see “Converting Data from SQL to C Data Types” and “Converting Data from C to SQL Data Types” in Appendix D, “Data Types” of the **SOLID Programmer Guide**.

ODBC also defines a scalar function for converting data from one SQL data type to another. The **CONVERT** scalar function is mapped by the driver to the underlying scalar function or functions defined to perform conversions in the data source. Because it is mapped to DBMS-specific functions, ODBC does not define how these conversions work or what conversions must be supported. An application discovers what conversions are supported by a particular driver and data source through the SQL\_CONVERT options in **SQLGetInfo**. For more information about the **CONVERT** scalar function, see “[Escape Sequences](#)” in Chapter 8, “SQL Statements” and Appendix E, “Scalar Functions” in the **SOLID Programmer Guide**.

## Conformance Levels

ODBC drivers give the application access to diverse data sources. Each driver lets the application determine at run time what ODBC capabilities and what SQL grammar the driver and each data source supports. Note that this is not a requirement of applications designed to work with a single driver or a small, known set of drivers, because these applications can simply be written to the capabilities of that driver or drivers. To help applications discover driver and data source capabilities, two areas of conformance are available: the ODBC interface and SQL grammar.

## Interface Conformance Levels

The purpose of leveling is to inform the application what features are available to it from the driver. A leveling scheme based on functions does not sufficiently achieve this goal. In ODBC 3.x, drivers are classified based on the features they possess. Supporting the feature may include supporting the function; it

may also include supporting a descriptor field, a statement attribute, a “Y” value for an information type returned by **SQLGetInfo**, and so on.

To simplify specification of interface conformance, ODBC defines three conformance levels. To meet a particular conformance level, a driver must satisfy all of the requirements of that conformance level. Conformance with a given level implies complete conformance with all lower levels.

Conformance levels do not always divide neatly into support for a specific list of ODBC functions, but specify supported features as listed in the following sections. To provide support for a feature, a driver must support some or all forms of calls to certain ODBC functions (for more information, see “[Function Conformance](#)”), setting certain attributes (for more information, see “Attribute Conformance”), and certain descriptor fields (for more information, see “[Descriptor Field Conformance](#)”).

The application discovers a driver’s interface conformance level by connecting to a data source, and calling **SQLGetInfo** with the SQL\_ODBC\_INTERFACE\_CONFORMANCE option.

Drivers are free to implement features beyond the level to which they claim complete conformance. Applications discover any such additional capabilities by calling **SQLGetFunctions** (to determine which ODBC functions are present) and **SQLGetInfo** (to query various other ODBC capabilities).

There are three ODBC interface conformance levels: Core, Level 1, and Level 2.

**Note** These conformance levels have different requirements than the ODBC API conformance levels of the same name in ODBC 2.x. In particular, all the features implied by ODBC 2.x API conformance Level 1 are now part of the Core interface conformance level. As a result, many ODBC drivers may report Core-level interface conformance.

### ***Core Interface Conformance***

All ODBC drivers must exhibit at least Core-level interface conformance. Because the features in the Core level are chosen such that they are what is required by most generic interoperable applications, this lets the driver work with such applications; it also corresponds to the features defined in the ISO CLI specification and the non-optional features defined in the X/Open CLI specification. A Core-level interface-conformant ODBC driver allows the application to do all of the following:

- Allocate and free all types of handles, by calling **SQLAllocHandle** and **SQLFreeHandle**.
- Use all forms of the **SQLFreeStmt** function.
- Bind result set columns, by calling **SQLBindCol**.
- Handle dynamic parameters, including arrays of parameters, in the input direction only, by calling **SQLBindParameter** and **SQLNumParams**. (Parameters in the output direction are feature 203 in “[Level 2 Interface Conformance](#).”)
- Specify a bind offset.
- Use the data-at-execution dialog, involving calls to **SQLParamData** and **SQLPutData**.

- Manage cursors and cursor names, by calling **SQLCloseCursor**, **SQLGetCursorName**, and **SQLSetCursorName**.
- Gain access to the description (metadata) of result sets, by calling **SQLColAttribute**, **SQLDescribeCol**, **SQLNumResultCols**, and **SQLRowCount**. (Use of these functions on column number 0 to retrieve bookmark metadata is feature 204 in “[Level 2 Interface Conformance](#).”)
- Query the data dictionary, by calling the catalog functions **SQLColumns**, **SQLGetTypeInfo**, **SQLStatistics**, and **SQLTables**. (The driver is not required to support multipart names of database tables and views. For more information, see feature 101 in “[Level 1 Interface Conformance](#)” and feature 201 in “[Level 2 Interface Conformance](#).”) However, certain features of the SQL92 specification, such as column qualification and names of indexes, are syntactically comparable to multipart naming. The present list of ODBC features is not intended to introduce new options into these aspects of SQL92.)
- Manage data sources and connections, by calling **SQLConnect**, **SQLDataSources**, **SQLDisconnect**, and **SQLDriverConnect**. Obtain information on drivers, no matter which ODBC level they support, by calling **SQLDrivers**.
- Prepare and execute SQL statements, by calling **SQLExecDirect**, **SQLExecute**, and **SQLPrepare**.
- Fetch one row of a result set or multiple rows, in the forward direction only, by calling **SQLFetch**, or by calling **SQLFetchScroll** with the *FetchOrientation* argument set to **SQL\_FETCH\_NEXT**.
- Obtain an unbound column in parts, by calling **SQLGetData**.
- Obtain current values of all attributes, by calling **SQLGetConnectAttr**, **SQLGetEnvAttr**, and **SQLGetStmtAttr**; and set all attributes to their default values and set certain attributes to non-default values by calling **SQLSetConnectAttr**, **SQLSetEnvAttr**, and **SQLSetStmtAttr**.
- Manipulate certain fields of descriptors, by calling **SQLCopyDesc**, **SQLGetDescField**, **SQLGetDescRec**, **SQLSetDescField**, and **SQLSetDescRec**.
- Obtain diagnostic information, by calling **SQLGetDiagField** and **SQLGetDiagRec**.
- Detect driver capabilities, by calling **SQLGetFunctions** and **SQLGetInfo**. Also, detect the result of any text substitutions made to an SQL statement before it is sent to the data source, by calling **SQLNativeSql**.
- Use the syntax of **SQLEndTran** to commit a transaction. A Core-level driver need not support true transactions; therefore, the application cannot specify **SQL\_ROLLBACK**, nor **SQL\_AUTOCOMMIT\_OFF** for the **SQL\_ATTR\_AUTOCOMMIT** connection attribute. (For more information, see feature 109 in “[Level 2 Interface Conformance](#).”)
- Call **SQLCancel** to cancel the data-at-execution dialog and, in multithread environments, to cancel an ODBC function executing in another thread. Core-level interface conformance does not mandate support for asynchronous execution of functions, nor the use of **SQLCancel** to cancel an ODBC function executing asynchronously. Neither the platform nor the ODBC driver need be

multithread for the driver to conduct independent activities at the same time. However, in multithread environments, the ODBC driver must be thread-safe. Serialization of requests from the application is a conformant way to implement this specification, even though it may create serious performance problems.

- Obtain the SQL\_BEST\_ROWID row-identifying column of tables, by calling **SQLSpecialColumns**. (Support for SQL\_ROWVER is feature 208 in “[Level 2 Interface Conformance](#).”)

**Important** ODBC Drivers must implement the functions in the Core interface conformance level.

### ***Level 1 Interface Conformance***

The Level 1 interface conformance level includes the Core interface conformance level functionality, plus additional features, such as transactions, that are usually available in an OLTP relational DBMS. A Level 1 interface-conformant driver lets the application do the following, in addition to the features in the Core interface conformance level:

|     |   |
|-----|---|
| 101 | Specify the schema of database tables and views (using two-part naming). (For more information, see the three-part naming feature 201 in “ <a href="#">Level 2 Interface Conformance</a> .”)  |
| 102 | Invoke true asynchronous execution of ODBC functions, where applicable ODBC functions are all synchronous or all asynchronous on a given connection.  |
| 103 | Use scrollable cursors, and thereby achieve access to a result set in methods other than forward-only, by calling <b>SQLFetchScroll</b> with the <i>FetchOrientation</i> argument other than SQL_FETCH_NEXT. (The SQL_FETCH_BOOKMARK <i>FetchOrientation</i> is in feature 204 in “ <a href="#">Level 2 Interface Conformance</a> .”) |
| 104 | Obtain primary keys of tables, by calling <b>SQLPrimaryKeys</b> .   |
| 105 | Use stored procedures, through the ODBC escape sequence for procedure calls; and query the data dictionary regarding stored procedures, by calling <b>SQLProcedureColumns</b> and <b>SQLProcedures</b> . (The process by which procedures are created and stored on the data source is outside the scope of this document.)           |
| 106 | Connect to a data source by interactively browsing the available servers, by calling <b>SQLBrowseConnect</b> .  |
| 107 | Use ODBC functions instead of SQL statements to perform certain database operations: <b>SQLSetPos</b> with SQL_POSITION and SQL_REFRESH.  |
| 108 | Gain access to the contents of multiple result sets generated by batches and stored procedures, by calling <b>SQLMoreResults</b> .  |
| 109 | Delimit transactions spanning several ODBC functions, with true atomicity and the ability to specify SQL_ROLLBACK in <b>SQLEndTran</b> .  |

### ***Level 2 Interface Conformance***

The Level 2 interface conformance level includes the Level 1 interface conformance-level functionality, plus the following features:

|     |  |
|-----|--|
| 201 | Use three-part names of database tables and views. (For more information, see the two-part naming support feature 101 in “ <a href="#">Level 1 Interface Conformance</a> .”) |
|-----|--|

|     |   |
|-----|---|
| 202 | Describe dynamic parameters, by calling <b>SQLDescribeParam</b> .   |
| 203 | Use not only input parameters, but also output and input/output parameters, and result values of stored procedures.   |
| 204 | Use bookmarks, including retrieving bookmarks by calling <b>SQLDescribeCol</b> and <b>SQLColAttribute</b> on column number 0; fetching based on a bookmark by calling <b>SQLFetchScroll</b> with the <i>FetchOrientation</i> argument set to SQL_FETCH_BOOKMARK; and update, delete, and fetch by bookmark operations by calling <b>SQLBulkOperations</b> with the <i>Operation</i> argument set to SQL_UPDATE_BY_BOOKMARK, SQL_DELETE_BY_BOOKMARK, or SQL_FETCH_BY_BOOKMARK. |
| 205 | Retrieve advanced information on the data dictionary, by calling <b>SQLColumnPrivileges</b> , <b>SQLForeignKeys</b> , and <b>SQLTablePrivileges</b> .   |
| 206 | Use ODBC functions instead of SQL statements to perform additional database operations, by calling <b>SQLBulkOperations</b> with SQL_ADD, or <b>SQLSetPos</b> with SQL_DELETE or SQL_UPDATE. (Support for calls to <b>SQLSetPos</b> with the <i>LockType</i> argument set to SQL_LOCK_EXCLUSIVE or SQL_LOCK_UNLOCK is not a part of the conformance levels, but is an optional feature.)  |
| 207 | Enable asynchronous execution of ODBC functions for specified individual statements.  |
| 208 | Obtain the SQL_ROWVER row-identifying column of tables, by calling <b>SQLSpecialColumns</b> . (For more information, see the support for <b>SQLSpecialColumns</b> with the <i>IdentifierType</i> argument set to SQL_BEST_ROWID as feature 20 in “ <a href="#">Core Interface Conformance</a> .”)   |
| 209 | Set the SQL_ATTR_CONCURRENCY statement attribute to at least one value other than SQL_CONCUR_READ_ONLY.   |
| 210 | The ability to time out login request and SQL queries (SQL_ATTR_LOGIN_TIMEOUT and SQL_ATTR_QUERY_TIMEOUT).  |
| 211 | The ability to change the default isolation level; the ability to execute transactions with the “serializable” level of isolation.  |

### ***Function Conformance***

The following table indicates the conformance level of each ODBC function, where this is well-defined.

| Function            | Conformance level |
|---------------------|-------------------|
| SQLAllocHandle      | Core              |
| SQLBindCol          | Core              |
| SQLBindParameter    | Core [1]          |
| SQLBrowseConnect    | Level 1           |
| SQLBulkOperations   | Level 1           |
| SQLCancel           | Core [1]          |
| SQLCloseCursor      | Core              |
| SQLColAttribute     | Core [1]          |
| SQLColumnPrivileges | Level 2           |
| SQLColumns          | Core              |
| SQLConnect          | Core              |

|                     |          |
|---------------------|----------|
| SQLCopyDesc         | Core     |
| SQLDataSources      | Core     |
| SQLDescribeCol      | Core [1] |
| SQLDescribeParam    | Level 2  |
| SQLDisconnect       | Core     |
| SQLDriverConnect    | Core     |
| SQLDrivers          | Core     |
| SQLEndTran          | Core [1] |
| SQLExecDirect       | Core     |
| SQLExecute          | Core     |
| SQLFetch            | Core     |
| SQLFetchScroll      | Core [1] |
| SQLForeignKeys      | Level 2  |
| SQLFreeHandle       | Core     |
| SQLFreeStmt         | Core     |
| SQLGetConnectAttr   | Core     |
| SQLGetCursorName    | Core     |
| SQLGetData          | Core     |
| SQLGetDescField     | Core     |
| SQLGetDescRec       | Core     |
| SQLGetDiagField     | Core     |
| SQLGetDiagRec       | Core     |
| SQLGetEnvAttr       | Core     |
| SQLGetFunctions     | Core     |
| SQLGetInfo          | Core     |
| SQLGetStmtAttr      | Core     |
| SQLGetTypeInfo      | Core     |
| SQLMoreResults      | Level 1  |
| SQLNativeSql        | Core     |
| SQLNumParams        | Core     |
| SQLNumResultCols    | Core     |
| SQLParamData        | Core     |
| SQLPrepare          | Core     |
| SQLPrimaryKeys      | Level 1  |
| SQLProcedureColumns | Level 1  |
| SQLProcedures       | Level 1  |
| SQLPutData          | Core     |
| SQLRowCount         | Core     |

|                    |             |
|--------------------|-------------|
| SQLSetConnectAttr  | Core [2]    |
| SQLSetCursorName   | Core        |
| SQLSetDescField    | Core [1]    |
| SQLSetDescRec      | Core        |
| SQLSetEnvAttr      | Core [2]    |
| SQLSetPos          | Level 1 [1] |
| SQLSetStmtAttr     | Core [2]    |
| SQLSpecialColumns  | Core [1]    |
| SQLStatistics      | Core        |
| SQLTablePrivileges | Level 2     |
| SQLTables          | Core        |

[1] Significant features of this function are available only at higher conformance levels.

[2] Setting certain attributes to non-default values depends on the conformance level. For more information, see the next section, “Attribute Conformance.”

### ***Attribute Conformance***

The following table indicates the conformance level of each ODBC environment attribute, where this is well-defined.

| Function                    | Conformance level |
|-----------------------------|-------------------|
| SQL_ATTR_CONNECTION_POOLING | -- [1]            |
| SQL_ATTR_CP_MATCH           | -- [1]            |
| SQL_ATTR_ODBC_VER           | Core              |
| SQL_ATTR_OUTPUT_NTS         | -- [1]            |

[1] This is an optional feature and as such is not part of the conformance levels.

The following table indicates the conformance level of each ODBC connection attribute, where this is well-defined.

| Function                    | Conformance level   |
|-----------------------------|---------------------|
| SQL_ATTR_ACCESS_MODE        | Core                |
| SQL_ATTR_ASYNC_ENABLE       | Level 1/Level 2 [1] |
| SQL_ATTR_AUTO_IPD           | Level 2             |
| SQL_ATTR_AUTOCOMMIT         | Level 1             |
| SQL_ATTR_CONNECTION_DEAD    | Level 1             |
| SQL_ATTR_CONNECTION_TIMEOUT | Level 2             |
| SQL_ATTR_CURRENT_CATALOG    | Level 2             |

|                           |                     |
|---------------------------|---------------------|
| SQL_ATTR_LOGIN_TIMEOUT    | Level 2             |
| SQL_ATTR_ODBC_CURSORS     | Core                |
| SQL_ATTR_PACKET_SIZE      | Level 2             |
| SQL_ATTR_QUIET_MODE       | Core                |
| SQL_ATTR_TRACE            | Core                |
| SQL_ATTR_TRACEFILE        | Core                |
| SQL_ATTR_TRANSLATE_LIB    | Core                |
| SQL_ATTR_TRANSLATE_OPTION | Core                |
| SQL_ATTR_TXN_ISOLATION    | Level 1/Level 2 [2] |

[1] Applications that support connection-level asynchrony (required for Level 1) must support setting this attribute to SQL\_TRUE by calling **SQLSetConnectAttr**; the attribute need not be settable to a value other than its default value through **SQLSetStmtAttr**. Applications that support statement-level asynchrony (required for Level 2) must support setting this attribute to SQL\_TRUE using either function.

[2] For Level 1 interface conformance, the driver must support one value in addition to the driver-defined default value (available by calling **SQLGetInfo** with the SQL\_DEFAULT\_TXN\_ISOLATION option). For Level 2 interface conformance, the driver must also support SQL\_TXN\_SERIALIZABLE.

The following table indicates the conformance level of each ODBC statement attribute, where this is well-defined.

| Function                       | Conformance level   |
|--------------------------------|---------------------|
| SQL_ATTR_APP_PARAM_DESC        | Core                |
| SQL_ATTR_APP_ROW_DESC          | Core                |
| SQL_ATTR_ASYNC_ENABLE          | Level 1/Level 2 [1] |
| SQL_ATTR_CONCURRENCY           | Level 1/Level 2 [2] |
| SQL_ATTR_CURSOR_SCROLLABLE     | Level 1             |
| SQL_ATTR_CURSOR_SENSITIVITY    | Level 2             |
| SQL_ATTR_CURSOR_TYPE           | Core/Level 2 [3]    |
| SQL_ATTR_ENABLE_AUTO_IPD       | Level 2             |
| SQL_ATTR_FETCH_BOOKMARK_PTR    | Level 2             |
| SQL_ATTR_IMP_PARAM_DESC        | Core                |
| SQL_ATTR_IMP_ROW_DESC          | Core                |
| SQL_ATTR_KEYSET_SIZE           | Level 2             |
| SQL_ATTR_MAX_LENGTH            | Level 1             |
| SQL_ATTR_MAX_ROWS              | Level 1             |
| SQL_ATTR_METADATA_ID           | Core                |
| SQL_ATTR_NOSCAN                | Core                |
| SQL_ATTR_PARAM_BIND_OFFSET_PTR | Core                |
| SQL_ATTR_PARAM_BIND_TYPE       | Core                |



|                               |         |
|-------------------------------|---------|
| SQL_ATTR_PARAM_OPERATION_PTR  | Core    |
| SQL_ATTR_PARAM_STATUS_PTR     | Core    |
| SQL_ATTR_PARAMS_PROCESSED_PTR | Core    |
| SQL_ATTR_PARAMSET_SIZE        | Core    |
| SQL_ATTR_QUERY_TIMEOUT        | Level 2 |
| SQL_ATTR_RETRIEVE_DATA        | Level 1 |
| SQL_ATTR_ROW_ARRAY_SIZE       | Core    |
| SQL_ATTR_ROW_BIND_OFFSET_PTR  | Core    |
| SQL_ATTR_ROW_BIND_TYPE        | Core    |
| SQL_ATTR_ROW_NUMBER           | Level 1 |
| SQL_ATTR_ROW_OPERATION_PTR    | Level 1 |
| SQL_ATTR_ROW_STATUS_PTR       | Core    |
| SQL_ATTR_ROWS_FETCHED_PTR     | Core    |
| SQL_ATTR_SIMULATE_CURSOR      | Level 2 |
| SQL_ATTR_USE_BOOKMARKS        | Level 2 |

[1] Applications that support connection-level asynchrony (required for Level 1) must support setting this attribute to SQL\_TRUE by calling **SQLSetConnectAttr**; the attribute need not be settable to a value other than its default value through **SQLSetStmtAttr**. Applications that support statement-level asynchrony (required for Level 2) must support setting this attribute to SQL\_TRUE using either function.

[2] For Level 2 interface conformance, the driver must support SQL\_CONCUR\_READ\_ONLY and at least one other value.

[3] For Level 1 interface conformance, the driver must support SQL\_CURSOR\_FORWARD\_ONLY and at least one other value. For Level 2 interface conformance, the driver must support all values defined in this document.

### ***Descriptor Field Conformance***

The following table indicates the conformance level of each ODBC descriptor header field, where this is well-defined.

| Function                    | Conformance level                               |
|-----------------------------|---|
| SQL_DESC_ALLOC_TYPE         | Core  |
| SQL_DESC_ARRAY_SIZE         | Core  |
| SQL_DESC_ARRAY_STATUS_PTR   | Core (for APD, IPR, and IRD); Level 1 (for ARD) |
| SQL_DESC_BIND_OFFSET_PTR    | Core  |
| SQL_DESC_BIND_TYPE          | Core  |
| SQL_DESC_COUNT              | Core  |
| SQL_DESC_ROWS_PROCESSED_PTR | Core  |

The following table indicates the conformance level of each ODBC descriptor record field, where this is well-defined.

| Function                             | Conformance level |
|--------------------------------------|-------------------|
| SQL_DESC_AUTO_UNIQUE_VALUE           | Level 2           |
| SQL_DESC_BASE_COLUMN_NAME            | Core              |
| SQL_DESC_BASE_TABLE_NAME             | Level 1           |
| SQL_DESC_CASE_SENSITIVE              | Core              |
| SQL_DESC_CATALOG_NAME                | Level 2           |
| SQL_DESC_CONCISE_TYPE                | Core              |
| SQL_DESC_DATA_PTR                    | Core              |
| SQL_DESC_DATETIME_INTERVAL_CODE      | Core [1]          |
| SQL_DESC_DATETIME_INTERVAL_PRECISION | Core [1]          |
| SQL_DESC_DISPLAY_SIZE                | Core              |
| SQL_DESC_FIXED_PREC_SCALE            | Core              |
| SQL_DESC_INDICATOR_PTR               | Core              |
| SQL_DESC_LABEL                       | Level 2           |
| SQL_DESC_LENGTH                      | Core              |
| SQL_DESC_LITERAL_PREFIX              | Core              |
| SQL_DESC_LITERAL_SUFFIX              | Core              |
| SQL_DESC_LOCAL_TYPE_NAME             | Core              |
| SQL_DESC_NAME                        | Core              |
| SQL_DESC_NULLABLE                    | Core              |
| SQL_DESC_OCTET_LENGTH                | Core              |
| SQL_DESC_OCTET_LENGTH_PTR            | Core              |
| SQL_DESC_PARAMETER_TYPE              | Core/Level 2 [2]  |
| SQL_DESC_PRECISION                   | Core              |
| SQL_DESC_ROWVER                      | Level 1           |
| SQL_DESC_SCALE                       | Core              |
| SQL_DESC_SCHEMA_NAME                 | Level 1           |
| SQL_DESC_SEARCHABLE                  | Core              |
| SQL_DESC_TABLE_NAME                  | Level 1           |
| SQL_DESC_TYPE                        | Core              |
| SQL_DESC_TYPE_NAME                   | Core              |
| SQL_DESC_UNNAMED                     | Core              |
| SQL_DESC_UNSIGNED                    | Core              |
| SQL_DESC_UPDATABLE                   | Core              |

[1]Support for these record fields is required only if the driver supports the applicable data types.

[2]For Core-level conformance, the driver must support `SQL_PARAM_INPUT`. For Level 2 interface conformance, the driver must also support `SQL_PARAM_INPUT_OUTPUT` and `SQL_PARAM_OUTPUT`.

## SQL Conformance Levels

The level of SQL92 grammar supported by a driver is indicated by the value returned by a call to **SQLGetInfo** with the `SQL_SQL_CONFORMANCE` information type. This indicates whether the driver conforms to the Entry, FIPS Transitional, Intermediate, or Full levels defined in SQL92.

All ODBC drivers must support the minimum SQL grammar described in Appendix C, “SQL Minimum Grammar” of the **SOLID Programmer Guide**. This grammar is a subset of the Entry level of SQL92. Drivers may support additional SQL and, in fact, be conformant to the SQL92 Entry, Intermediate, or Full level, or to the FIPS 127-2 Transitional level. Drivers that comply to a given level of SQL92 or FIPS 127-2 can support additional features in any of the higher levels, yet not be fully conformant to that level. To determine whether a feature is supported, an application should call **SQLGetInfo** with the appropriate information type. The conformance level of an SQL feature is described in the corresponding information type (see the **SQLGetInfo** function description in the Part II PDF file, “ODBC API Reference,” available on the Solid Web site.

## Environment, Connection, and Statement Attributes

ODBC defines a number of attributes that are associated with environments, connections, or statements.

Environment attributes affect the entire environment, such as whether connection pooling is enabled. Environment attributes are set with **SQLSetEnvAttr** and retrieved with **SQLGetEnvAttr**.

Connection attributes affect each connection individually, such as how long a driver should wait while attempting to connect to a data source before timing out. Connection attributes are set with **SQLSetConnectAttr** and retrieved with **SQLGetConnectAttr**. For more information about connection attributes, see “[Connection Attributes](#)” in Chapter 6, “Connecting to a Data Source or Driver.”

Statement attributes affect each statement individually, such as whether a statement should be executed asynchronously. Statement attributes are set with **SQLSetStmtAttr** and retrieved with **SQLGetStmtAttr**. A few statement attributes are read-only attributes and cannot be set. For example, the `SQL_ATTR_ROW_NUMBER` statement attribute, which is used to retrieve the number of the current row in the cursor, is read-only. For more information about statement attributes, see “[Statement Attributes](#)” in Chapter 9, “Executing Statements.”

In addition to attributes defined by ODBC, a driver can define its own connection and statement attributes. Driver-defined attributes must be registered with X/Open to ensure that two driver vendors do not assign the same integer value to different, proprietary attributes. For more information, see “[Driver-Specific Data Types, Descriptor Types, Information Types, Diagnostic Types, and Attributes](#)” in Chapter 17, “Programming Considerations.”

For a complete list of attributes, see `SQLSetEnvAttr`, `SQLSetConnectAttr`, and `SQLSetStmtAttr`, in the Part II PDF file, “ODBC API Reference,” available on the Solid Web site. Most attributes are also described in the description of the ODBC function that they affect.

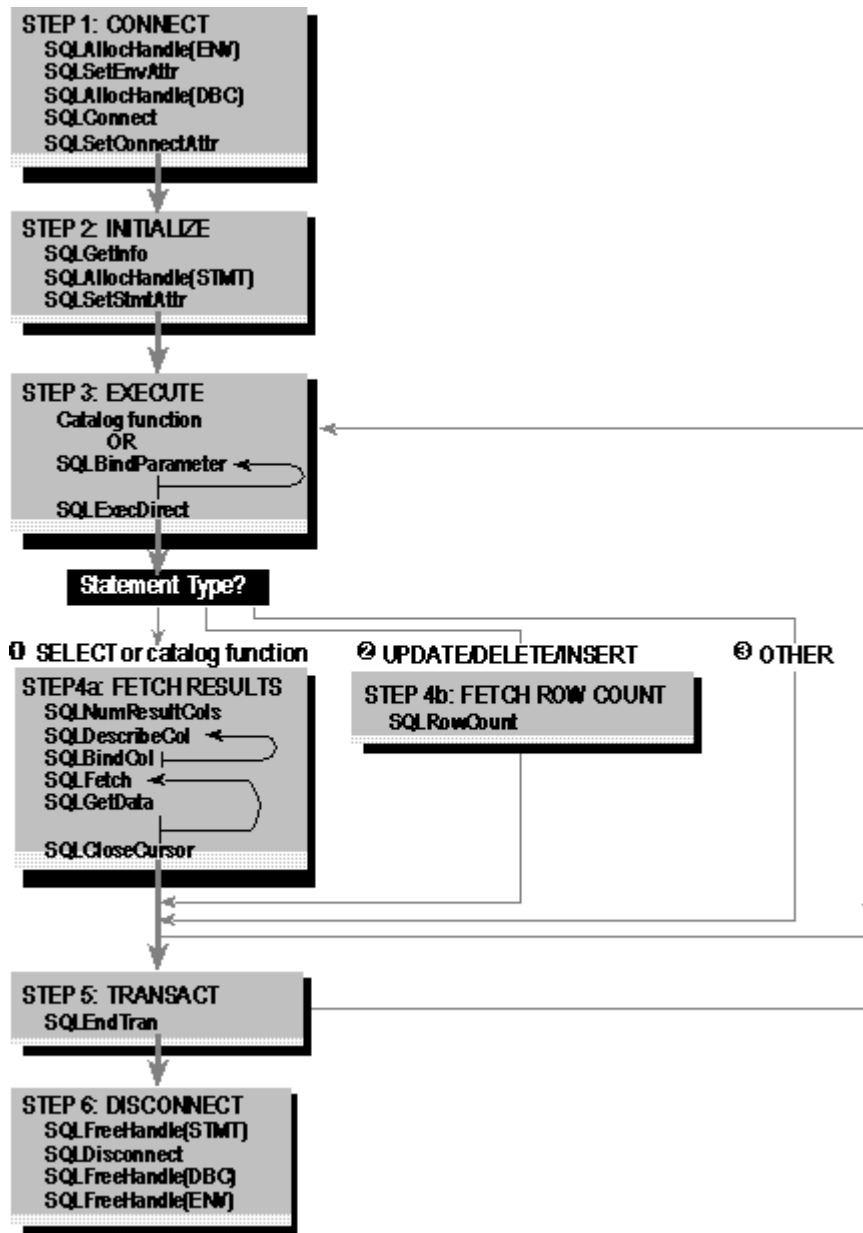
## Tables and Views

In ODBC functions, tables and views are interchangeable. The term *table* is used for both tables and views, except where the term *view* is used explicitly.

## Chapter 5: Overview of Basic ODBC Application Steps

This chapter describes the general flow of ODBC applications. It is unlikely that any application calls all of these functions in exactly this order. However, most applications use some variation of these steps. The basic application steps are shown in the following figure.

## Basic application steps



## Step 1: Connect to the Data Source

The first step in any application is to connect to the data source. This phase, including the functions it requires, is illustrated in the following figure.



The first step in connecting to the data source is to load the Driver Manager and allocate the environment handle with **SQLAllocHandle**. For more information, see [“Allocating the Environment Handle”](#) in Chapter 6, “Connecting to a Data Source or Driver.”

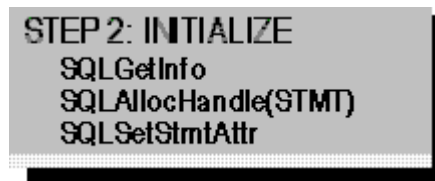
The application then registers the version of ODBC to which it conforms by calling **SQLSetEnvAttr** with the `SQL_ATTR_APP_ODBC_VER` environment attribute. For more information, see the [“Declaring the Application’s ODBC Version”](#) section in Chapter 6, “Connecting to a Data Source or Driver,” and the [“Backward Compatibility and Standards Compliance”](#) section in Chapter 17, “Programming Considerations.”

Next, the application allocates a connection handle with **SQLAllocHandle** and connects to the data source with **SQLConnect**, **SQLDriverConnect**, or **SQLBrowseConnect**. For more information, see [“Allocating a Connection Handle”](#) and “Establishing a Connection” in Chapter 6, “Connecting to a Data Source or Driver.”

The application then sets any connection attributes, such as whether to manually commit transactions. For more information, see [“Connection Attributes”](#) in Chapter 6, “Connecting to a Data Source or Driver.”

## Step 2: Initialize the Application

The second step is to initialize the application, as illustrated in the following figure.



Exactly what is done here varies with the application.

At this point, it is common to use **SQLGetInfo** to discover the capabilities of the driver. For more information, see “Considering Database Features to Use” in Chapter 16, “Interoperability.”

All applications need to allocate a statement handle with **SQLAllocHandle**, and many applications set statement attributes, such as the cursor type, with **SQLSetStmtAttr**. For more information, see [“Allocating a Connection Handle”](#) and [“Statement Attributes”](#) in Chapter 9, “Executing Statements.”

### Step 3: Build and Execute an SQL Statement

The third step is to build and execute an SQL statement, as shown in the following figure. The methods used to perform this step are likely to vary tremendously. The application might prompt the user to enter an SQL statement, build an SQL statement based on user input, or use a hard-coded SQL statement. For more information, see [“Constructing SQL Statements”](#) in Chapter 8, “SQL Statements.”

If the SQL statement contains parameters, the application binds them to application variables by calling **SQLBindParameter** for each parameter. For more information, see [“Statement Parameters”](#) in Chapter 9, “Executing Statements.”

After the SQL statement is built and any parameters are bound, the statement is executed with **SQLExecDirect**. If the statement will be executed multiple times, it can be prepared with **SQLPrepare** and executed with **SQLExecute**. For more information, see [“Executing a Statement”](#) in Chapter 9, “Executing Statements.”

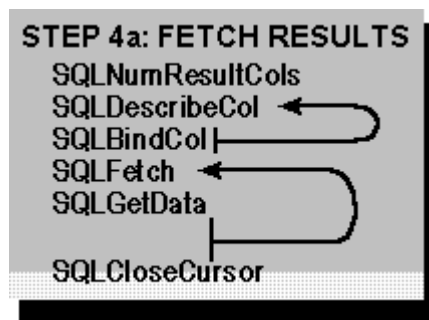
The application might also forgo executing an SQL statement altogether and instead call a function to return a result set containing catalog information, such as the available columns or tables. For more information, see [“Uses of Catalog Data”](#) in Chapter 7, “Catalog Functions.”

The application’s next action depends on the type of SQL statement executed.

| Type of SQL statement             | Proceed to  |
|-----------------------------------|---|
| <b>SELECT</b> or catalog function | “Step 4a: Fetch the Results”  |
| UPDATE, DELETE, or INSERT         | “Step 4b: Fetch the Row Count”  |
| All other SQL statements          | “Step 3: Build and Execute an SQL Statement” (this topic) or <a href="#">“Step 5: Commit the Transaction”</a> |

### Step 4a: Fetch the Results

The next step is to fetch the results, as shown in the following figure.



If the statement executed in Step 3, “Build and Execute an SQL Statement,” was a **SELECT** statement or a catalog function, the application first calls **SQLNumResultCols** to determine the number of columns in the result set. This step is not necessary if the application already knows the number of result set columns, such as when the SQL statement is hard-coded in a vertical or custom application.

Next, the application retrieves the name, data type, precision, and scale of each result set column with **SQLDescribeCol**. Again, this is not necessary for applications such as vertical and custom applications that already know this information. It passes this information to **SQLBindCol**, which binds an application variable to a column in the result set.

The application now calls **SQLFetch** to retrieve the first row of data and place the data from that row in the variables bound with **SQLBindCol**. If there is any long data in the row, it then calls **SQLGetData** to retrieve that data. The application continues to call **SQLFetch** and **SQLGetData** to retrieve additional data. After it has finished fetching data, it calls **SQLCloseCursor** to close the cursor.

For a complete description of retrieving results, see “[Chapter 10: Overview of Retrieving Results \(Basic\)](#)” and “[Chapter 11: Overview of Retrieving Results \(Advanced\)](#).”

The application now returns to Step 3, “Build and Execute an SQL Statement,” to execute another statement in the same transaction; or proceeds to Step 5, “Commit the Transaction,” to commit or roll back the transaction.

## Step 4b: Fetch the Row Count

If the statement executed in Step 3 was an **UPDATE**, **DELETE**, or **INSERT** statement, the application retrieves the count of affected rows with **SQLRowCount**. For more information, see “[Determining the Number of Affected Rows](#)” in Chapter 12, “Updating Data.”

The application now returns to Step 3 to execute another statement in the same transaction or proceeds to Step 5 to commit or roll back the transaction.

## Step 5: Commit the Transaction

The fifth step is to call **SQLEndTran** to commit or roll back the transaction. The application performs this step only if it set the transaction commit mode to manual commit; if the transaction commit mode is auto-commit, which is the default, the transaction is automatically committed when the statement is executed. or more information, see “[Chapter 14: Overview of Transactions](#).”

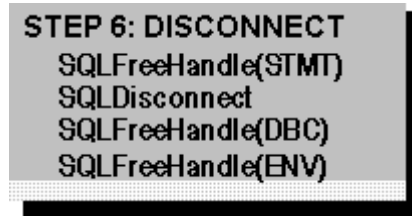
**STEP 5: TRANSACT**  
**SQLEndTran**

To execute a statement in a new transaction, the application returns to Step 3. To disconnect from the data source, the application proceeds to Step 6.



## Step 6: Disconnect from the Data Source

The final step is to disconnect from the data source, as illustrated in the following figure. First, the application frees any statement handles by calling **SQLFreeHandle**. For more information, see “[Freeing a Statement Handle](#)” in Chapter 9, “Executing Statements.”



Next, the application disconnects from the data source with **SQLDisconnect** and frees the connection handle with **SQLFreeHandle**. For more information, see “[Disconnecting from a Data Source or Driver](#)” in Chapter 6, “Connecting to a Data Source or Driver.”

Finally, the application frees the environment handle with **SQLFreeHandle** and unloads the Driver Manager. For more information, see “[Allocating a Connection Handle](#)” in Chapter 6, “Connecting to a Data Source or Driver.”

## Chapter 6: Overview of Connecting to a Data Source or Driver

An application can be connected to any number of drivers and data sources. These can be a variety of drivers and data sources, the same driver and a variety of data sources, or even multiple connections to the same driver and data source.

### Allocating the Environment Handle

The first task for any ODBC application is to load the Driver Manager; how this is done is operating-system dependent. For example, on a computer running Windows NT Server, Windows NT Workstation, or Windows 95, the application either links to the Driver Manager library or calls `LoadLibrary` to load the Driver Manager DLL.

The next task, which must be done before an application can call any other ODBC function, is to initialize the ODBC environment and allocate an environment handle. To do this:

The application declares a variable of type `SQLHENV`. It then calls **SQLAllocHandle** and passes the address of this variable and the `SQL_HANDLE_ENV` option. For example:

```
SQLHENV henv1;  
SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv1);
```

The Driver Manager allocates a structure in which to store information about the environment, and returns the environment handle in the variable.

The Driver Manager does not call **SQLAllocHandle** in the driver at this time because it does not know which driver to call. It delays calling **SQLAllocHandle** in the driver until the application calls a function to connect to a data source. For more information, see “[Driver Manager’s Role in the Connection Process](#),” later in this chapter.

When the application has finished using ODBC, it frees the environment handle with **SQLFreeHandle**. After freeing the environment, it is an application programming error to use the environment’s handle in a call to an ODBC function; doing so has undefined but probably fatal consequences.

When **SQLFreeHandle** is called, the driver releases the structure used to store information about the environment. Note that **SQLFreeHandle** cannot be called for an environment handle until after all connection handles on that environment handle have been freed.

For more information about the environment handle, see “[Environment Handles](#)” in Chapter 4, “ODBC Fundamentals.”

### Declaring the Application’s ODBC Version

Before an application allocates a connection, it must set the `SQL_ATTR_ODBC_VERSION` environment attribute. This attribute states that the application follows the ODBC 2.x or ODBC 3.x specification when using the following items:

**SQLSTATEs.** Many `SQLSTATE` values are different in ODBC 2.x and ODBC 3.x.

**Date, Time, and Timestamp Type Identifiers.** The following table shows the type identifiers for date, time, and timestamp data in ODBC 2.x and ODBC 3.x.

| ODBC 2.x             | ODBC 3.x             |
|----------------------|----------------------|
| SQL Type Identifiers |                      |
| SQL_DATE             | SQL_TYPE_DATE        |
| SQL_TIME             | SQL_TYPE_TIME        |
| SQL_TIMESTAMP        | SQL_TYPE_TIMESTAMP   |
| C Type Identifiers   |                      |
| SQL_C_DATE           | SQL_C_TYPE_DATE      |
| SQL_C_TIME           | SQL_C_TYPE_TIME      |
| SQL_C_TIMESTAMP      | SQL_C_TYPE_TIMESTAMP |

**CatalogName Argument in SQLTables.** In ODBC 2.x, the wild card characters (“%” and “\_”) in the *CatalogName* argument are treated literally. In ODBC 3.x, they are treated as wild cards. Thus, an application that follows the ODBC 2.x specification cannot use these as wild cards and does not escape them when using them as literals. An application that follows the ODBC 3.x specification can use these as wild cards or escape them and use them as literals. For more information, see “[Arguments in Catalog Functions](#)” in Chapter 7, “Catalog Functions.”

The ODBC 3.x Driver Manager and ODBC 3.x drivers check the version of the ODBC specification to which an application is written and respond accordingly. For example, if the application follows the ODBC 2.x specification and calls **SQLExecute** before calling **SQLPrepare**, the ODBC 3.x Driver Manager returns SQLSTATE S1010 (Function sequence error). If the application follows the ODBC 3.x specification, the Driver Manager returns SQLSTATE HY010 (Function sequence error). For more information, see “[Backward Compatibility and Standards Compliance](#)” in Chapter 17, “Programming Considerations.”

**Important** Applications that follow the ODBC 3.x specification must use conditional code to avoid using functionality new to ODBC 3.x when working with ODBC 2.x drivers. ODBC 2.x drivers do not support functionality new to ODBC 3.x just because the application declares that it follows the ODBC 3.x specification. Furthermore, ODBC 3.x drivers do not cease to support functionality new to ODBC 3.x just because the application declares that it follows the ODBC 2.x specification.

## Choosing a Data Source or Driver

The data source or driver used by an application is sometimes hard-coded in the application. For example, a custom application written by an MIS department to transfer data from one data source to another would contain the names of those data sources — the application simply would not work with any other data sources. Another example is a vertical application, such as one used for order entry. Such an application always uses the same data source, which has a predefined schema known by the application.

Other applications choose the data source or driver at run time. Usually, these are generic applications that do ad hoc queries, such as a spreadsheet that uses ODBC to import data. Such applications usually list the available data sources or drivers and let users choose the ones they want to work with. Whether a generic application lists data sources, drivers, or both often depends on whether the application uses DBMS- or file-based drivers.

DBMS-based drivers usually require a fairly complex set of connection information, such as the network address, network protocol, database name, and so on. The purpose of a data source is to hide all of this information. Hence, the data source paradigm lends itself to use with DBMS-based drivers. An application can display a list of data sources to the user in one of two ways. It can call **SQLDriverConnect** with the **DSN** (Data Source Name) keyword and no associated value; the Driver Manager will display a list of data source names. If the application wants control over the appearance of the list, it calls **SQLDataSources** to retrieve a list of available data sources and constructs its own dialog box. This function is implemented by the Driver Manager and can be called before any drivers are loaded. The application then calls a connection function and passes it the name of the chosen data source.

If a data source is not specified, the default data source indicated by the system information is used. (For more information, see “Default Subkey” in Chapter 20, “Configuring Data Sources” contained in the Part III PDF file, “Installing and Configuring ODBC,” available on the Solid Web site.) If **SQLConnect** is called with a *ServerName* argument that cannot be found, is a null pointer, or is “DEFAULT”, the Driver Manager connects to the default data source. The default data source is also used if the connection string used in a call to **SQLDriverConnect** or **SQLBrowseConnect** contains the DSN keyword set to “DEFAULT”, or the data source is not found. Additionally, the default data source is used if the connection string used in a call to **SQLDriverConnect** does not contain the DSN keyword.

With file-based drivers, it is possible to use a file paradigm. For data stored on the local machine, users often know that their data is in a particular file, such as EMPLOYEE.DBF. Rather than choosing an unknown data source, it is easier for such users to choose the file they know. To implement this, the application first calls **SQLDrivers**. This function is implemented by the Driver Manager and can be called before any drivers are loaded. **SQLDrivers** returns a list of available drivers; it also returns values for the **FileUsage** and **FileExtns** keywords. The **FileUsage** keyword explains whether file-based drivers treat files as tables, such as Xbase, or databases, such as Microsoft Access. The **FileExtns** keyword lists the file extensions the driver recognizes, such as .dbf for an Xbase driver. Using this information, the application constructs a dialog box with which the user chooses a file. Based on the extension of the chosen file, the application then connects directly to the driver by calling **SQLDriverConnect** with the **DRIVER** keyword.

There is nothing to stop an application from using a data source with a file-based driver or calling **SQLDriverConnect** with the **DRIVER** keyword to connect directly to a DBMS-based driver. Here are several common uses of the **DRIVER** keyword for DBMS-based drivers:

**Not creating data sources.** For example, a custom application might use a particular driver and database. If the driver name and all information needed to connect to the database is hard-coded in the application, users do not need to create a data source on their computer to run the application — all they need to do is install the application and driver.

A disadvantage of this method is that the application must be recompiled and redistributed if the connection information changes. If a data source name is hard-coded in the application instead of complete connection information, then each user only needs to change the information in the data source.

**Accessing a particular DBMS a single time.** For example, a spreadsheet that retrieves data by calling ODBC functions might contain the **DRIVER** keyword to identify a particular driver. Because the driver name is meaningful to any users that have that driver, the spreadsheet could be passed among those users. If the spreadsheet contained a data source name, each user would have to create the same data source to use the spreadsheet.

**Browsing the system for all databases accessible to a particular driver.** For more information, see [“Connecting with SQLBrowseConnect,”](#) later in this chapter.

## Allocating a Connection Handle

Before the application can connect to a data source or driver, it must allocate a connection handle. To do this:

The application declares a variable of type `SQLHDBC`. It then calls **SQLAllocHandle** and passes the address of this variable, the handle of the environment in which to allocate the connection, and the `SQL_HANDLE_DBC` option. For example:

```
SQLHDBC hdbc1;  
SQLAllocHandle(SQL_HANDLE_DBC, henv1, &hdbc1);
```

The Driver Manager allocates a structure in which to store information about the statement and returns the connection handle in the variable.

The Driver Manager does not call **SQLAllocHandle** in the driver at this time because it does not know which driver to call. It delays calling **SQLAllocHandle** in the driver until the application calls a function to connect to a data source. For more information, see [“Driver Manager’s Role in the Connection Process,”](#) later in this chapter.

It is important to note that allocating a connection handle is not the same as loading a driver. The driver is not loaded until a connection function is called. Thus, after allocating a connection handle and before connecting to the driver or data source, the only functions the application can call with the connection handle are **SQLSetConnectAttr**, **SQLGetConnectAttr**, or **SQLGetInfo** with the `SQL_ODBC_VER` option. Calling other functions with the connection handle, such as **SQLEndTran**, returns `SQLSTATE 08003` (Connection not open). For complete details, see Appendix B, “ODBC State Transition Tables” contained on the Microsoft Web site (ODBC Programmer’s Reference).

For more information about connection handles, see [“Connection Handles”](#) in Chapter 4, “ODBC Fundamentals.”

## Connection Attributes

Connection attributes are characteristics of the connection. For example, because transactions occur at the connection level, the transaction isolation level is a connection attribute. Similarly, the login timeout, or number of seconds to wait while trying to connect before timing out, is a connection attribute.

Connection attributes are set with **SQLSetConnectAttr** and their current settings retrieved with **SQLGetConnectAttr**. If **SQLSetConnectAttr** is called before the driver is loaded, the Driver Manager stores the attributes in its connection structure and sets them in the driver as part of the connection process. There is no requirement that an application set any connection attributes; all connection attributes have defaults, some of which are driver specific.

A connection attribute can be set before or after connection, or either, depending on the attribute and the driver. The login timeout (`SQL_ATTR_LOGIN_TIMEOUT`) applies to the connection process and is effective only if set before connecting. The attributes that specify whether to use the ODBC cursor library (`SQL_ATTR_ODBC_CURSORS`) and the network packet size (`SQL_ATTR_PACKET_SIZE`) must be set

before connecting. The reason for this is that the ODBC cursor library resides between the Driver Manager and the driver, and therefore must be loaded before the driver.

The attributes to specify whether a data source is read-only or read-write (`SQL_ATTR_ACCESS_MODE`) and the current catalog (`SQL_ATTR_CURRENT_CATALOG`) can be set before or after connecting, depending on the driver. However, interoperable applications set them before connecting because some drivers do not support changing these after connecting.

Some connection attributes have a default before the connection is made, while others do not. Those that do are `SQL_ATTR_ACCESS_MODE`, `SQL_ATTR_AUTOCOMMIT`, `SQL_ATTR_LOGIN_TIMEOUT`, `SQL_ATTR_ODBC_CURSORS`, `SQL_ATTR_TRACE`, and `SQL_ATTR_TRACEFILE`.

The translation connection attributes (`SQL_ATTR_TRANSLATE_DLL` and `SQL_ATTR_TRANSLATE_OPTION`) must be set after connecting.

All other connection attributes can be set at any time. For more information, see the `SQLSetConnectAttr` function description contained in the Part II PDF file, “ODBC API Reference,” available on the Solid Web site. (Connection attributes cannot be set on the environment level by a call to `SQLSetEnvAttr`.)

## Establishing a Connection

After allocating environment and connection handles and setting any connection attributes, the application is ready to connect to the data source or driver. There are three different functions the application can use to do this: **SQLConnect** (Core interface conformance level), **SQLDriverConnect** (Core), and **SQLBrowseConnect** (Level 1). Each of the three is designed to be used in a different scenario. Before connecting, the application can determine which of these functions is supported with the **ConnectFunctions** keyword returned by **SQLDrivers**.

**Note** Some drivers limit the number of active connections they support. An application calls **SQLGetInfo** with the `SQL_MAX_DRIVER_CONNECTIONS` option to determine how many active connections a particular driver supports.

## Default Data Source

The driver may select a data source, called the default data source, in certain cases where the application does not explicitly specify one:

In a call to **SQLConnect** where the *ServerName* argument is a zero-length string, a null pointer, or `DEFAULT`.

In a call to **SQLDriverConnect** where *InConnectionString* either specifies `DSN=DEFAULT` or specifies with the `DSN` keyword a data source that is not contained in the system information.

It is driver-defined how the default data source is specified. This may involve administrative action and may depend on the user.

## Connecting with SQLConnect

**SQLConnect** is the simplest connection function. It requires a data source name and accepts an optional user ID and password. It works well for applications that hard-code a data source name and do not require a user ID or password. It also works well for applications that want to control their own “look and feel,” or

that have no user interface. Such applications can build a list of data sources using **SQLDataSources**; prompt the user for data source, user ID, and password; and then call **SQLConnect**.

## Connection Strings

A connection string contains information used for establishing a connection. A complete connection string contains all the information needed to establish a connection. The connection string is a series of keyword/value pairs separated by semicolons. (For the complete syntax of a connection string, see the **SQLDriverConnect** function description in the Part II PDF file, “ODBC API Reference,” available on the Solid Web site.) The connection string is used by:

**SQLDriverConnect**, which completes it by interaction with the user.

**SQLBrowseConnect**, which completes it iteratively with the data source.

**SQLConnect** does not use a connection string; using **SQLConnect** is analogous to connecting using a connection string with exactly three keyword/value pairs (for data source name, and optionally user ID and password).

## ODBC Connection Pooling

Connection pooling enables an application to use a connection from a pool of connections that do not need to be reestablished for each use. Once a connection has been created and placed in a pool, an application can reuse that connection without performing the complete connection process.

Using a pooled connection can result in significant performance gains, because applications can save the overhead involved in making a connection. This can be particularly significant for middle-tier applications that connect over a network or for applications that repeatedly connect and disconnect, such as Internet applications.

In addition to performance gains, the connection pooling architecture enables an environment and its associated connections to be used by multiple components in a single process. This means that stand-alone components in the same process can interact with each other without being aware of each other. A connection in a connection pool can be used repeatedly by multiple components.

**Note** Connection pooling can be used by an ODBC application exhibiting ODBC 2.x behavior, as long as the application can call *SQLSetEnvAttr*. When using connection pooling, the application must not execute SQL statements that change the database or the context of the database, such as changing the *<database name>*, which changes the catalog used by a data source.

An ODBC driver must be fully thread-safe, and connections must not have thread affinity to support connection pooling. This means the driver is able to handle a call on any thread at any time and is able to connect on one thread, to use the connection on another thread, and to disconnect on a third thread.

The connection pool is maintained by the Driver Manager. Connections are drawn from the pool when the application calls **SQLConnect** or **SQLDriverConnect** and are returned to the pool when the application calls **SQLDisconnect**. The size of the pool grows dynamically, based on the requested resource allocations. It shrinks based on the inactivity timeout: If a connection is inactive for a period of time (it has not been used in a connection), it is removed from the pool. The size of the pool is limited only by memory constraints and limits on the server.

The Driver Manager determines whether a specific connection in a pool should be used according to the arguments passed in **SQLConnect** or **SQLDriverConnect**, and according to the connection attributes set after the connection was allocated.

When the Driver Manager is pooling connections, it needs to be able to determine if a connection is still working before handing out the connection. Otherwise, the Driver Manager keeps on handing out the dead connection to the application whenever a transient network failure occurs. A new connection attribute has been defined in ODBC 3.x: **SQL\_ATTR\_CONNECTION\_DEAD**. This is a read-only connection attribute that returns either **SQL\_CD\_TRUE** or **SQL\_CD\_FALSE**. The value **SQL\_CD\_TRUE** means that the connection has been lost, while the value **SQL\_CD\_FALSE** means that the connection is still active. (Drivers conforming to earlier versions of ODBC can also support this attribute.)

A driver must implement this option efficiently or it will impair the connection pooling performance. Specifically, a call to get this connection attribute should not cause a round trip to the server. Instead, a driver should just return the last known state of the connection. The connection is dead if the last trip to the server failed, and not dead if the last trip succeeded.

In order to prevent unwanted repeated attempts by the Driver Manager to reestablish a connection when connection pooling is enabled, you can set **ODBCGetTryWaitValue**. **ODBCSetTryWaitValue** saves the information in the registry at the following location:

HKEY\_LOCAL\_MACHINE\Software\Odbc\Odbcinst.ini\ODBC Connection Pooling\Retry Wait

If there is a bad connection in the pool, the ODBC Driver Manager will attempt to connect before the connection can be reused for the first time. If the connection still fails, the ODBC Driver Manager returns the error and marks the connection with the time. From that point until the RetryWait value expires, the ODBC Driver Manager returns a failure without testing the connection.

To use a connection pool, an application performs the following steps:

- Enables connection pooling by calling **SQLSetEnvAttr** to set the **SQL\_ATTR\_CONNECTION\_POOLING** environment attribute to **SQL\_CP\_ONE\_PER\_DRIVER** or **SQL\_CP\_ONE\_PER\_HENV**. This call must be made before the application allocates the shared environment for which connection pooling is to be enabled. The environment handle in the call to **SQLSetEnvAttr** should be set to null, which makes **SQL\_ATTR\_CONNECTION\_POOLING** a process-level attribute. If the attribute is set to **SQL\_CP\_ONE\_PER\_DRIVER**, a single connection pool is supported for each driver. If an application works with many drivers and few environments, this might be more efficient because fewer comparisons may be required. If set to **SQL\_CP\_ONE\_PER\_HENV**, a single connection pool is supported for each environment. If an application works with many environments and few drivers, this might be more efficient because fewer comparisons may be required. Connection pooling is disabled by setting **SQL\_ATTR\_CONNECTION\_POOLING** to **SQL\_CP\_OFF**.
- Allocates an environment by calling **SQLAllocHandle** with the *HandleType* argument set to **SQL\_HANDLE\_ENV**. The environment allocated by this call will be an implicit shared environment because connection pooling has been enabled. The environment to be used is not determined, however, until **SQLAllocHandle** with a *HandleType* of **SQL\_HANDLE\_DBC** is called on this environment.



- Allocates a connection by calling **SQLAllocHandle** with *InputHandle* set to **SQL\_HANDLE\_DBC**, and the *InputHandle* set to the environment handle allocated for connection pooling. The Driver Manager attempts to find an existing environment that matches the environment attributes set by the application. If no such environment exists, one is created, with a reference count (maintained by the Driver Manager) of 1. If a matching shared environment is found, the environment is returned to the application and its reference count is incremented. (The actual connection to be used is not determined by the Driver Manager until **SQLConnect** or **SQLDriverConnect** is called.)
- Calls **SQLConnect** or **SQLDriverConnect** to make the connection. The Driver Manager uses the connection options in the call to **SQLConnect** (or the connection keywords in the call to **SQLDriverConnect**) and the connection attributes set after connection allocation to determine which connection in the pool should be used.

**Note** How a requested connection is matched to a pooled connection is determined by the **SQL\_ATTR\_CP\_MATCH** environment attribute. For more information, see **SQLSetEnvAttr** in the Part II PDF file, “ODBC API Reference,” available on the Solid Web site.

Calls **SQLDisconnect** when done with the connection. The connection is returned to the connection pool and becomes available for reuse.

**Note** For more information about connection pooling, see "Connection Pooling Considerations" in Chapter 17, "Programming Considerations."

## Disconnecting from a Data Source or Driver

When an application has finished using a data source, it calls **SQLDisconnect**. **SQLDisconnect** frees any statements that are allocated on the connection and disconnects the driver from the data source. It returns an error if a transaction is in process.

After disconnecting, the application can call **SQLFreeHandle** to free the connection. After freeing the connection, it is an application programming error to use the connection's handle in a call to an ODBC function; doing so has undefined but probably fatal consequences. When **SQLFreeHandle** is called, the driver releases the structure used to store information about the connection.

The application also can reuse the connection, either to connect to a different data source or reconnect to the same data source. The decision to remain connected, as opposed to disconnecting and reconnecting later, requires that the application writer consider the relative costs of each option; both connecting to a data source and remaining connected can be relatively costly depending on the connection medium. In making a correct tradeoff, the application must also make assumptions about the likelihood and timing of further operations on the same data source.

## Driver Manager's Role in the Connection Process

Remember that applications do not call driver functions directly. Instead, they call Driver Manager functions with the same name and the Driver Manager calls the driver functions. Usually, this happens almost immediately. For example, the application calls **SQLExecute** in the Driver Manager and after a few error checks, the Driver Manager calls **SQLExecute** in the driver.

The connection process is different. When the application calls **SQLAllocHandle** with the **SQL\_HANDLE\_ENV** and **SQL\_HANDLE\_DBC** options, the function allocates handles only in the Driver Manager. The Driver Manager does not call this function in the driver because it does not know which driver to call. Similarly, if the application passes the handle of an unconnected connection to **SQLSetConnectAttr** or **SQLGetConnectAttr**, only the Driver Manager executes the function. It stores or gets the attribute value from its connection handle and returns SQLSTATE 08003 (Connection not open) when getting a value for an attribute that has not been set and for which ODBC does not define a default value.

When the application calls **SQLConnect**, **SQLDriverConnect**, or **SQLBrowseConnect**, the Driver Manager first determines which driver to use. It then checks whether a driver is currently loaded on the connection:

If no driver is loaded on the connection, the Driver Manager checks whether the specified driver is loaded on another connection in the same environment. If not, the Driver Manager loads the driver on the connection and calls **SQLAllocHandle** in the driver with the **SQL\_HANDLE\_ENV** option.

The Driver Manager then calls **SQLAllocHandle** in the driver with the **SQL\_HANDLE\_DBC** option, regardless of whether it was just loaded. If the application set any connection attributes, the Driver Manager calls **SQLSetConnectAttr** in the driver; if an error occurs, the Driver Manager's connection function returns SQLSTATE IM006 (Driver's **SQLSetConnectAttr** failed). Finally, the Driver Manager calls the connection function in the driver.

If the specified driver is loaded on the connection, the Driver Manager calls only the connection function in the driver. In this case, the driver must make sure that all connection attributes on the connection maintain their current settings.

If a different driver is loaded on the connection, the Driver Manager calls **SQLFreeHandle** in the driver to free the connection. If there are no other connections that use the driver, the Driver Manager calls **SQLFreeHandle** in the driver to free the environment and unloads the driver. The Driver Manager then performs the same operations as when a driver is not loaded on the connection.

When the application calls **SQLDisconnect**, the Driver Manager calls **SQLDisconnect** in the driver. However, it leaves the driver loaded in case the application reconnects to the driver. When the application calls **SQLFreeHandle** with the **SQL\_HANDLE\_DBC** option, the Driver Manager calls **SQLFreeHandle** in the driver. If the driver is not used by any other connections, the Driver Manager then calls **SQLFreeHandle** in the driver with the **SQL\_HANDLE\_ENV** option and unloads the driver.

## Connecting with **SQLBrowseConnect**

**SQLBrowseConnect**, like **SQLDriverConnect**, uses a connection string. However, by using **SQLBrowseConnect**, an application can construct a complete connection string at run time. This allows the application to do two things:

Build its own dialog boxes to prompt for this information, thereby retaining control over its “look and feel.” Browse the system for data sources that can be used by a particular driver, possibly in several steps. For example, the user might first browse the network for servers and, after choosing a server, browse the server for databases accessible by the driver.

The application calls **SQLBrowseConnect** and passes a connection string, known as the *browse request connection string*, that specifies a driver or data source. The driver returns a connection string, known as

the *browse result connection string*, that contains keywords, possible values (if the keyword accepts a discrete set of values), and user-friendly names. The application builds a dialog box with the user-friendly names and prompts the user for values. It then builds a new browse request connection string from these values and returns this to the driver with another call to **SQLBrowseConnect**.

Because connection strings are passed back and forth, the driver can provide several levels of browsing by returning a new connection string when the application returns the old one. For example, the first time an application calls **SQLBrowseConnect**, the driver might return keywords to prompt the user for a server name. When the application returns the server name, the driver might return keywords to prompt the user for a database. The browsing process would be complete after the application returned the database name.

Each time **SQLBrowseConnect** returns a new browse result connection string, it returns **SQL\_NEED\_DATA** as its return code. This tells the application that the connection process is not complete. Until **SQLBrowseConnect** returns **SQL\_SUCCESS**, the connection is in a Need Data state and cannot be used for other purposes, such as to set a connection attribute. The application can terminate the connection browsing process by calling **SQLDisconnect**.

### ***SQL Server Browsing Example***

The following example shows how **SQLBrowseConnect** might be used to browse the connections available with a driver for SQL Server. First, the application requests a connection handle:

```
SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
```

Next, the application calls **SQLBrowseConnect** and specifies the SQL Server driver, using the driver description returned by **SQLDrivers**:

```
SQLBrowseConnect(hdbc, "DRIVER={SQL Server};", SQL_NTS, BrowseResult,
    sizeof(BrowseResult), &BrowseResultLen);
```

Because this is the first call to **SQLBrowseConnect**, the Driver Manager loads the SQL Server driver and calls the driver's **SQLBrowseConnect** function with the same arguments it received from the application.

The driver determines that this is the first call to **SQLBrowseConnect** and returns the second level of connection attributes: server, user name, password, application name, and workstation ID. For the server attribute, it returns a list of valid server names. The return code from **SQLBrowseConnect** is **SQL\_NEED\_DATA**. Here is the browse result string:

```
"SERVER:Server={red,blue,green,yellow};UID:Login ID=?;PWD:Password=?;
*APP:AppName=?;*WSID:WorkStation ID=?;"
```

Each keyword in the browse result string is followed by a colon and one or more words before the equal sign. These words are the user-friendly name that an application can use to build a dialog box. The **APP** and **WSID** keywords are prefixed by an asterisk, which means they are optional. The **SERVER**, **UID**, and **PWD** keywords are not prefixed by an asterisk; values must be supplied for them in the next browse request string. The value for the **SERVER** keyword may be one of the servers returned by **SQLBrowseConnect** or a user-supplied name.

The application calls **SQLBrowseConnect** again, specifying the green server and omitting the **APP** and **WSID** keywords and the user-friendly names after each keyword:

```
SQLBrowseConnect(hdbc, "SERVER=green;UID=Smith;PWD=Sesame;", SQL_NTS,
    BrowseResult, sizeof(BrowseResult), &BrowseResultLen);
```

The driver attempts to connect to the green server. If there are any nonfatal errors, such as a missing keyword-value pair, **SQLBrowseConnect** returns `SQL_NEED_DATA` and remains in the same state as it was prior to the error. The application can call **SQLGetDiagField** or **SQLGetDiagRec** to determine the error. If the connection is successful, the driver returns `SQL_NEED_DATA` and returns the browse result string:

```
"*DATABASE:Database={master,model,pubs,tempdb};  
*LANGUAGE:Language={us_english,Français};"
```

Because the attributes in this string are optional, the application can omit them. However, the application must call **SQLBrowseConnect** again. If the application chooses to omit the database name and language, it specifies an empty browse request string. In this example, the application chooses the pubs database and calls **SQLBrowseConnect** a final time, omitting the **LANGUAGE** keyword and the asterisk before the **DATABASE** keyword:

```
SQLBrowseConnect(hdbc, "DATABASE=pubs;", SQL_NTS, BrowseResult,  
    sizeof(BrowseResult), &BrowseResultLen);
```

Because the **DATABASE** attribute is the final connection attribute required by the driver, the browsing process is complete, the application is connected to the data source, and **SQLBrowseConnect** returns `SQL_SUCCESS`. **SQLBrowseConnect** also returns the complete connection string as the browse result string:

```
"DSN=MySQLServer;SERVER=green;UID=Smith;PWD=Sesame;DATABASE=pubs;"
```

The final connection string returned by the driver does not contain the user-friendly names after each keyword, nor does it contain optional keywords not specified by the application. The application can use this string with **SQLDriverConnect** to reconnect to the data source on the current connection handle (after disconnecting) or to connect to the data source on a different connection handle. For example:

```
SQLDriverConnect(hdbc, hwnd, BrowseResult, SQL_NTS, ConnStrOut,  
    sizeof(ConnStrOut), &ConnStrOutLen, SQL_DRIVER_NOPROMPT);
```

## Connecting with SQLDriverConnect

**SQLDriverConnect** is used to connect to a data source using a connection string. **SQLDriverConnect** is used instead of **SQLConnect** for the following reasons:

- To let the application use driver-specific connection information.
- To request that the driver prompt the user for connection information.
- To connect without specifying a data source.

## Driver-Specific Connection Information

**SQLConnect** assumes that a data source name, user ID, and password are sufficient to connect to a data source and that all other connection information can be stored on the system. This is often not the case. For example, a driver might need one user ID and password to log into a server and a different user ID and password to log into a DBMS. Because **SQLConnect** accepts a single user ID and password, this means that the other user ID and password must be stored with the data source information on the system if **SQLConnect** is to be used. This is a potential breach of security and should be avoided unless the password is encrypted.

**SQLDriverConnect** allows the driver to define an arbitrary amount of connection information in the keyword-value pairs of the connection string. For example, suppose a driver requires a data source name, a user ID and password for the server, and a user ID and password for the DBMS. A custom program that always uses the XYZ Corp data source might prompt the user for IDs and passwords and build the following set of keyword-value pairs, or *connection string*, to pass to **SQLDriverConnect**:

```
DSN=XYZ Corp;UID=Gomez;PWD=Sesame;UIDDBMS=JGomez;PWddbms=Shazam;
```

The **DSN** (Data Source Name) keyword names the data source, the **UID** and **PWD** keywords specify the user ID and password for the server, and the **UIDDBMS** and **PWddbms** keywords specify the user ID and password for the DBMS. Note that the final semicolon is optional. **SQLDriverConnect** parses this string; uses the XYZ Corp data source name to retrieve additional connection information from the system, such as the server address; and logs on to the server and DBMS using the specified user IDs and passwords.

Keyword-value pairs in **SQLDriverConnect** must follow certain syntax rules. The keywords and their values should not contain the []{}(),?\*=!@ characters. The value of the **DSN** keyword cannot consist only of blanks, and should not contain leading blanks. Because of the registry grammar, keywords and data source names cannot contain the backslash (\) character. Spaces are not allowed around the equal sign in the keyword-value pair.

The **FILEDSN** keyword can be used in a call to **SQLDriverConnect** to specify the name of a file containing data source information (see “Connecting Using File Data Sources” later in this section). The **SAVEFILE** keyword can be used to specify the name of a .dsn file in which the keyword-value pairs of a successful connection made by the call to **SQLDriverConnect** will be saved. For more information on file data sources, see the SQLDriverConnect function description in the Part II PDF file, “ODBC API Reference” available on the Solid Web site.

## Prompting the User for Connection Information

If the application uses **SQLConnect** and needs to prompt the user for any connection information, such as a user name and password, it must do so itself. While this allows the application to control its “look and feel,” it might force the application to contain driver-specific code. This occurs when the application needs to prompt the user for driver-specific connection information. This presents an impossible situation for generic applications, which are designed to work with any and all drivers, including drivers that do not exist when the application is written.

**SQLDriverConnect** can prompt the user for connection information. For example, the custom program mentioned earlier could pass the following connection string to **SQLDriverConnect**:

```
DSN=XYZ Corp;
```

The driver might then display a dialog box similar to the following, which prompts for user IDs and passwords.

|                  |                          |
|------------------|--------------------------|
| Server ID:       | <input type="text"/>     |
| Server Password: | <input type="password"/> |
| DBMS ID:         | <input type="text"/>     |
| DBMS Password:   | <input type="password"/> |

That the driver can prompt for connection information is particularly useful to generic and vertical applications. These applications should not contain driver-specific information, and having the driver prompt for the information it needs keeps that information out of the application. This is shown by the previous two examples. When the application passed only the data source name to the driver, the application did not contain any driver-specific information and was therefore not tied to a particular driver; when the application passed a complete connection string to the driver, it was tied to the driver that could interpret that string.

A generic application might take this one step further and not even specify a data source. When **SQLDriverConnect** receives an empty connection string, the Driver Manager displays the following dialog box.

**Select Data Source** ? X

File Data Source | Machine Data Source

Look in: Data Sources

DSN Name:  New...

Select the file data source that describes the driver that you wish to connect to. You can use any file data source that refers to an ODBC driver which is installed on your machine.

OK Cancel Help

After the user selects a data source, the Driver Manager constructs a connection string specifying that data source and passes it to the driver. The driver can then prompt the user for any additional information it needs.

The conditions under which the driver prompts the user are controlled by the *DriverCompletion* flag; there are options to always prompt, prompt if necessary, or never prompt. For a complete description of this flag, see the `SQLDriverConnect` function description in the Part II PDF file, “ODBC API Reference,” available on the Solid Web site.

## Connecting Using File Data Sources

*The connection information for a file data source* is stored in a .dsn file. As a result, the connection string can be used repeatedly by a single user or shared among several users if they have the appropriate driver installed. The file contains a driver name (or another data source name in the case of an unshareable file data source) and optionally, a connection string that can be used by **SQLDriverConnect**. The Driver Manager builds the connection string for the call to **SQLDriverConnect** from the keywords in the .dsn file.

A file data source allows an application to specify connection options without having to build a connection string for use with **SQLDriverConnect**. The file data source usually is created by specifying the **SAVEFILE** keyword, which causes the Driver Manager to save the output connection string created by a call to **SQLDriverConnect** to the .dsn file. That connection string can be used repeatedly by calling **SQLDriverConnect** with the **FILEDSN** keyword. This streamlines the connection process and provides a persistent source of the connection string.

File data sources also can be created by calling **SQLCreateDataSource** in the installer DLL. Information can be written into the .dsn file by calling **SQLWriteFileDSN**, and read from the .dsn file by calling **SQLReadFileDSN**; both of these functions are also in the installer DLL. For information about the installer DLL, see Chapter 20, “Configuring Data Sources” contained in the Part III PDF file, “Configuring and Installing ODBC Software,” available on the Solid Web site.

The keywords used for connection information are in the [ODBC] section of a .dsn file. The minimum information that a shareable .dsn file would have in the [ODBC] section is the **DRIVER** keyword:

**DRIVER** = SQL Server

The shareable .dsn file usually contains a connection string, as follows:

**DRIVER** = SQL Server

**UID** = Larry

**DATABASE** = MyDB

When the file data source is unshareable, the .dsn file contains only a **DSN** keyword. When the Driver Manager is sent the information in an unshareable file data source, it connects as necessary to the data source indicated by the **DSN** keyword. An unshareable .dsn file would contain the following keyword:

**DSN** = MyDataSource

The connection string used for a file data source is the union of the keywords specified in the .dsn file and the keywords specified in the connection string in the call to **SQLDriverConnect**. If any of the keywords in the .dsn file conflict with keywords in the connection string, the Driver Manager decides which keyword value should be used. For more information, see the `SQLDriverConnect` function description in the Part II PDF file, “ODBC API Reference,” available on the Solid Web site.

## Connecting Directly to Drivers

As was discussed in “[Choosing a Data Source or Driver](#),” earlier in this chapter, some applications do not want to use a data source at all. Instead, they want to connect directly to a driver. **SQLDriverConnect** provides a way for the application to connect directly to a driver without specifying a data source. Conceptually, a temporary data source is created at run time.

To connect directly to a driver, the application specifies the **DRIVER** keyword in the connection string instead of the **DSN** keyword. The value of the **DRIVER** keyword is the description of the driver as returned by **SQLDrivers**. For example, suppose a driver has the description Paradox Driver and requires the name of a directory containing the data files. To connect to this driver, the application might use either of the following connection strings:

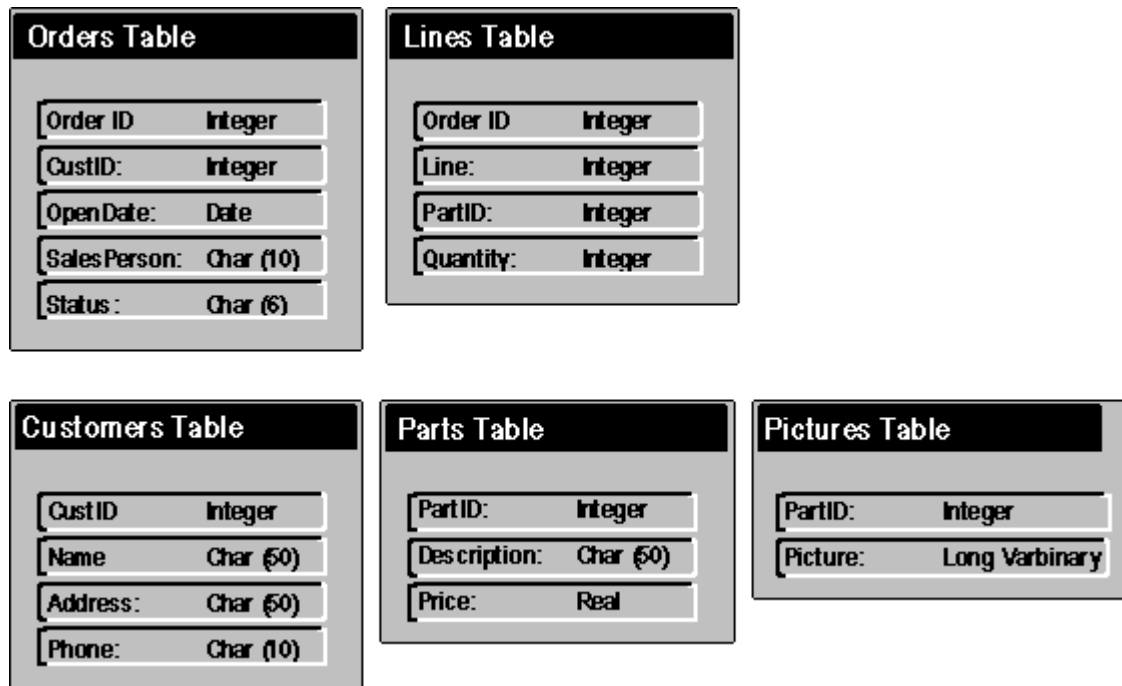
```
DRIVER={Paradox Driver};Directory=C:\PARADOX;  
DRIVER={Paradox Driver};
```

With the first string, the driver would not need any additional information. With the second string, the driver would need to prompt for the name of the directory containing the data files.



## Chapter 7: Overview of Catalog Functions

All databases have a structure that outlines how data will be stored in the database. For example, a simple sales order database might have the structure shown in the following illustration, in which the ID columns are used to link the tables.



This structure, along with other information such as privileges, is stored in a set of system tables called the database's *catalog*, which is also known as a *data dictionary*.

An application can discover this structure through calls to the *catalog functions*. The catalog functions return information in result sets and are usually implemented through **SELECT** statements against the tables in the catalog. For example, an application might request a result set containing information about all the tables on the system or all the columns in a particular table.

### Uses of Catalog Data

Applications use catalog data in a variety of ways. Here are some common uses:

- **Constructing SQL statements at run time.** Vertical applications, such as an order entry application, contain hard-coded SQL statements. The tables and columns that are used by the application are fixed ahead of time, as are the statements that access these tables. For example, an order entry application usually contains a single, parameterized **INSERT** statement for adding new orders to the system.

Generic applications, such as a spreadsheet program that uses ODBC to retrieve data, often construct SQL statements at run time based on input from the user. Such an application could require the user to type the names of the tables and columns to use. However, it would be easier for the user if the application displayed lists of tables and columns from which the user could make selections. To build these lists, the application would call the **SQLTables** and **SQLColumns** catalog functions.

- **Constructing SQL statements during development.** Application development environments typically allow the programmer to create database queries while developing a program. The queries are then hard-coded in the application being built.

Such environments could also use **SQLTables** and **SQLColumns** to create lists from which the programmer could make selections. They might also use **SQLPrimaryKeys** and **SQLForeignKeys** to automatically determine and show relationships between selected tables, and use **SQLStatistics** to determine and highlight indexed fields so the programmer can create efficient queries.

- **Constructing cursors.** An application, driver, or middleware that provides a scrollable cursor engine could use **SQLSpecialColumns** to determine which column or columns uniquely identify a row. The program could build a *keyset* containing the values of these columns for each row that has been fetched. When the application scrolls back to the row, it would then use these values to fetch the most recent data for the row. For more information about scrollable cursors and keysets, see “[Scrollable Cursors](#)” in Chapter 11, “Retrieving Results (Advanced).”

## Catalog Functions in ODBC

ODBC contains the following catalog functions:

| Function                   | Description  |
|----------------------------|--|
| <b>SQLTables</b>           | Returns a list of catalogs, schemas, tables, or table types in the data source.  |
| <b>SQLColumns</b>          | Returns a list of columns in one or more tables.   |
| <b>SQLStatistics</b>       | Returns a list of statistics about a single table. Also returns a list of indexes associated with that table.  |
| <b>SQLSpecialColumns</b>   | Returns a list of columns that uniquely identifies a row in a single table. Also returns a list of columns in that table that are automatically updated. |
| <b>SQLPrimaryKeys</b>      | Returns a list of columns that compose the primary key of a single table.  |
| <b>SQLForeignKeys</b>      | Returns a list of foreign keys in a single table or a list of foreign keys in other tables that refer to a single table.                                 |
| <b>SQLTablePrivileges</b>  | Returns a list of privileges associated with one or more tables.   |
| <b>SQLColumnPrivileges</b> | Returns a list of privileges associated with one or more columns in a single table.  |

|                            |  |
|----------------------------|--|
| <b>SQLProcedures</b>       | Returns a list of procedures in the data source.   |
| <b>SQLProcedureColumns</b> | Returns a list of input and output parameters, the return value, and the columns in the result set of a single procedure.  |
| <b>SQLGetTypeInfo</b>      | Returns a list of the SQL data types supported by the data source. These data types are generally used in <b>CREATE TABLE</b> and <b>ALTER TABLE</b> statements. |

Because **SQLTables**, **SQLColumns**, **SQLStatistics**, and **SQLSpecialColumns** conform to the X/Open CLI, and **SQLGetTypeInfo** conforms to the ISO 92 CLI, they are implemented by most drivers. The remaining catalog functions are in the ODBC conformance level.

## Data Returned by Catalog Functions

Each catalog function returns data as a result set. This result set is no different from any other result set. It is usually generated by a predefined, parameterized **SELECT** statement that is hard-coded in the driver or stored in a procedure in the data source. For information on how to retrieve data from a result set, see [“Was a Result Set Created?”](#) in Chapter 10, “Retrieving Results (Basic).”

The result set for each catalog function is described in the reference entry for that function. In addition to the listed columns, the result set can contain driver-specific columns after the last predefined column. These columns (if any) are described in the driver documentation.

Applications should bind driver-specific columns relative to the end of the result set. That is, they should calculate the number of a driver-specific column as the number of the last column—retrieved with **SQLNumResultCols**—less the number of columns that occur after the required column. This saves having to change the application when new columns are added to the result set in future versions of ODBC or the driver. For this scheme to work, drivers must add new driver-specific columns before old driver-specific columns so that column numbers do not change relative to the end of the result set.

Identifiers that are returned in the result set are not quoted, even if they contain special characters. For example, suppose the identifier quote character (which is driver-specific and returned through **SQLGetInfo**) is a double quotation mark (") and the Accounts Payable table contains a column named Customer Name. In the row returned by **SQLColumns** for this column, the value of the TABLE\_NAME column is Accounts Payable, not "Accounts Payable", and the value of the COLUMN\_NAME column is Customer Name, not "Customer Name". To retrieve the names of customers in the Accounts Payable table, the application would quote these names:

```
SELECT "Customer Name" FROM "Accounts Payable"
```

For more information, see [“Quoted Identifiers”](#) in Chapter 8, “SQL Statements.”

The catalog functions are based upon an SQL-like authorization model in which a connection is made based upon a username and password, and only data for which the user has a privilege is returned. Password protection of individual files, which does not fit into this model, is driver-defined.

The result sets returned by the catalog functions are almost never updatable, and applications should not expect to be able to change the structure of the database by changing the data in these result sets.

## Arguments in Catalog Functions

All catalog functions accept arguments with which an application can restrict the scope of the data returned. For example, the first and second calls to **SQLTables** in the following code return a result set containing information about all tables, while the third call returns information about the Orders table:

```
SQLTables(hstmt1, NULL, 0, NULL, 0, NULL, 0, NULL, 0);
SQLTables(hstmt2, NULL, 0, NULL, 0, "%", SQL_NTS, NULL, 0);
SQLTables(hstmt3, NULL, 0, NULL, 0, "Orders", SQL_NTS, NULL, 0);
```

Catalog function string arguments fall into four different types: ordinary arguments (OA), pattern value arguments (PV), identifier arguments (ID), and value list arguments (VL). Most string arguments can be of one of two different types, depending on the value of the SQL\_ATTR\_METADATA\_ID statement attribute. The following table lists the arguments for each catalog function and describes the type of the argument for a SQL\_TRUE or SQL\_FALSE value of SQL\_ATTR\_METADATA\_ID.

| Function                   | Argument             | Type when SQL_ATTR_METADATA_ID = SQL_FALSE | Type when SQL_ATTR_METADATA_ID = SQL_TRUE |
|----------------------------|----------------------|--|---|
| <b>SQLColumnPrivileges</b> | <i>CatalogName</i>   | OA   | ID  |
|                            | <i>SchemaName</i>    | OA   | ID  |
|                            | <i>TableName</i>     | OA   | ID  |
|                            | <i>ColumnName</i>    | PV   | ID  |
| <b>SQLColumns</b>          | <i>CatalogName</i>   | OA   | ID  |
|                            | <i>SchemaName</i>    | PV   | ID  |
|                            | <i>TableName</i>     | PV   | ID  |
|                            | <i>ColumnName</i>    | PV   | ID  |
| <b>SQLForeignKeys</b>      | <i>PKCatalogName</i> | OA   | ID  |
|                            | <i>PKSchemaName</i>  | OA   | ID  |
|                            | <i>PKTableName</i>   | OA   | ID  |
|                            | <i>FKCatalogName</i> | OA   | ID  |
|                            | <i>FKSchemaName</i>  | OA   | ID  |
|                            | <i>FKTableName</i>   | OA   | ID  |
| <b>SQLPrimaryKeys</b>      | <i>CatalogName</i>   | OA   | ID  |
|                            | <i>SchemaName</i>    | OA   | ID  |
|                            | <i>TableName</i>     | OA   | ID  |
| <b>SQLProcedureColumns</b> | <i>CatalogName</i>   | OA   | ID  |
|                            | <i>SchemaName</i>    | PV   | ID  |
|                            | <i>ProcName</i>      | PV   | ID  |
|                            | <i>ColumnName</i>    | PV   | ID  |
| <b>SQLProcedures</b>       | <i>CatalogName</i>   | OA   | ID  |
|                            | <i>SchemaName</i>    | PV   | ID  |
|                            | <i>ProcName</i>      | PV   | ID  |
| <b>SQLSpecialColumns</b>   | <i>CatalogName</i>   | OA   | ID  |
|                            | <i>SchemaName</i>    | OA   | ID  |
|                            | <i>TableName</i>     | OA   | ID  |
| <b>SQLStatistics</b>       | <i>CatalogName</i>   | OA   | ID  |

|                           |   |                      |                      |
|---------------------------|---|----------------------|----------------------|
|                           | <i>SchemaName</i><br><i>TableName</i>   | OA<br>OA             | ID<br>ID             |
| <b>SQLTablePrivileges</b> | <i>CatalogName</i><br><i>SchemaName</i><br><i>TableName</i>                     | OA<br>PV<br>PV       | ID<br>ID<br>ID       |
| <b>SQLTables</b>          | <i>CatalogName</i><br><i>SchemaName</i><br><i>TableName</i><br><i>TableType</i> | PV<br>PV<br>PV<br>VL | ID<br>ID<br>ID<br>VL |

### ***Ordinary Arguments***

When a catalog function string argument is an ordinary argument, it is treated as a literal string. An ordinary argument accepts neither a string search pattern, nor a list of values. The case of an ordinary argument is significant, and quote characters in the string are taken literally. These arguments are treated as ordinary arguments if the `SQL_ATTR_METADATA_ID` statement attribute is set to `SQL_FALSE`; they are treated as identifier arguments instead if this attribute is set to `SQL_TRUE`.

If an ordinary argument is set to a null pointer and the argument is a required argument, the function returns `SQL_ERROR` and `SQLSTATE HY009` (Invalid use of null pointer). If an ordinary argument is set to a null pointer and the argument is not a required argument, the argument's behavior is driver-dependent. The following arguments are required arguments:

| <b>Function</b>            | <b>Required arguments</b>       |
|----------------------------|---------------------------------|
| <b>SQLColumnPrivileges</b> | <i>TableName</i>                |
| <b>SQLForeignKeys</b>      | <i>PKTableName, FKTableName</i> |
| <b>SQLPrimaryKeys</b>      | <i>TableName</i>                |
| <b>SQLSpecialColumns</b>   | <i>TableName</i>                |
| <b>SQLStatistics</b>       | <i>TableName</i>                |

### ***Pattern Value Arguments***

Some arguments in the catalog functions, such as the *TableName* argument in **SQLTables**, accept search patterns. These arguments accept search patterns if the `SQL_ATTR_METADATA_ID` statement attribute is set to `SQL_FALSE`; they are identifier arguments that do not accept a search pattern if this attribute is set to `SQL_TRUE`.

The search pattern characters are:

- An underscore (`_`), which represents any single character.
- A percent sign (`%`), which represents any sequence of zero or more characters.
- An escape character, which is driver-specific and is used to include underscores, percent signs, and the escape character as literals.

The escape character is retrieved with the `SQL_SEARCH_PATTERN_ESCAPE` option in **SQLGetInfo**. It must precede any underscore, percent sign, or escape character in an argument that accepts search patterns to include that character as a literal. For example:

| Search pattern | Description   |
|----------------|---|
| %A%            | All identifiers containing the letter A   |
| ABC_           | All four character identifiers starting with ABC  |
| ABC\           | The identifier ABC_, assuming the escape character is a backslash (\)                       |
| \\%            | All identifiers starting with a backslash (\), assuming the escape character is a backslash |

Special care must be taken to escape search pattern characters in arguments that accept search patterns. This is particularly true for the underscore character, which is commonly used in identifiers. A common mistake in applications is to retrieve a value from one catalog function and pass that value to a search pattern argument in another catalog function. For example, suppose an application retrieves the table name `MY_TABLE` from the result set for **SQLTables** and passes this to **SQLColumns** to retrieve a list of columns in `MY_TABLE`. Instead of getting the columns for `MY_TABLE`, the application will get the columns for all the tables that match the search pattern `MY_TABLE`, such as `MY_TABLE`, `MY1TABLE`, `MY2TABLE`, and so on.

**Note** ODBC 2.x drivers do not support search patterns in the *CatalogName* argument in **SQLTables**. ODBC 3.x drivers accept search patterns in this argument if the `SQL_ATTR_ODBC_VERSION` environment attribute is set to `SQL_OV_ODBC3`; they do not accept search patterns in this argument if it is set to `SQL_OV_ODBC2`.

Passing a null pointer to a search pattern argument does not constrain the search for that argument; that is, a null pointer and the search pattern `%` (any characters) are equivalent. However, a zero-length search pattern—that is, a valid pointer to a string of length zero—matches only the empty string (`""`).

### Identifier Arguments

If a string in an identifier argument is quoted, the driver removes leading and trailing blanks, and treats literally the string within the quotation marks. If the string is not quoted, the driver removes trailing blanks and folds the string to uppercase. Setting an identifier argument to a null pointer returns `SQL_ERROR` and `SQLSTATE HY009` (Invalid use of null pointer), unless the argument is a catalog name and catalogs are not supported.

These arguments are treated as identifier arguments if the `SQL_ATTR_METADATA_ID` statement attribute is set to `SQL_TRUE`. In this case, the underscore (`_`) and the percent sign (`%`) will be treated as the actual character, not as a search pattern character. These arguments are treated as either an ordinary argument or a pattern argument, depending on the argument, if this attribute is set to `SQL_FALSE`.

Although identifiers containing special characters must be quoted in SQL statements, they must not be quoted when passed as catalog function arguments, because quote characters passed to catalog functions are interpreted literally. For example, suppose the identifier quote character (which is driver-specific and returned through **SQLGetInfo**) is a double quotation mark (`"`). The first call to **SQLTables** returns a result set containing information about the Accounts Payable table, while the second call returns information about the “Accounts Payable” table, which is probably not what was intended.

```
SQLTables(hstmt1, NULL, 0, NULL, 0, "Accounts Payable", SQL_NTS, NULL, 0);  
SQLTables(hstmt2, NULL, 0, NULL, 0, "\"Accounts Payable\"", SQL_NTS, NULL, 0);
```

Quoted identifiers are used to distinguish a true column name from a pseudo-column of the same name, such as ROWID in Oracle. If "ROWID" is passed in an argument of a catalog function, the function will work with the ROWID pseudo-column if it exists. If the pseudo-column does not exist, the function will work with the "ROWID" column. If ROWID is passed in an argument of a catalog function, the function will work with the ROWID column.

For more information about quoted identifiers, see "[Quoted Identifiers](#)" in Chapter 8, "SQL Statements."

### ***Value List Arguments***

A value list argument consists of a list of comma-separated values to be used for matching. There is only one value list argument in the ODBC catalog functions: the *TableType* argument in **SQLTables**. Setting *TableType* to a null pointer is the same as if it is set to SQL\_ALL\_TABLE\_TYPES, which enumerates all possible members of the value list. This argument is not affected by the SQL\_ATTR\_METADATA\_ID statement attribute. For more information, see the SQLTables function description in the Part II PDF file, "ODBC API Reference," available on the Solid Web site.

## **Schema Views**

An application can retrieve metadata information from the DBMS either by calling ODBC catalog functions or by using INFORMATION\_SCHEMA views. The views are defined by the ANSI SQL92 standard.

If supported by the DBMS and the driver, the INFORMATION\_SCHEMA views provide a more powerful and comprehensive means of retrieving metadata than the ODBC catalog functions provide. An application can execute its own custom **SELECT** statement against one of these views, can join views, or can perform a union on views. While offering greater utility and a wider range of metadata, INFORMATION\_SCHEMA views are not often supported by the DBMS. This may change as more DBMSs and drivers achieve compliance with SQL92.

To determine which views are supported, an application calls **SQLGetInfo** with the SQL\_INFO\_SCHEMA\_VIEWS option. To retrieve metadata from a supported view, the application executes a **SELECT** statement that specifies the schema information required.

## Chapter 8: Overview of SQL Statements

ODBC applications perform almost all database access by executing SQL statements. The form of these statements—hard-coded or constructed at run time, interoperable or data source-specific, and so on—depends on the needs of the application.

### Constructing SQL Statements

SQL statements can be constructed in one of three ways: hard-coded during development, constructed at run time, or entered directly by the user.

### Hard-Coded SQL Statements

Applications that perform a fixed task usually contain hard-coded SQL statements. For example, an order entry system might use the following call to list open sales orders:

```
SQLExecDirect(hstmt, "SELECT OrderID FROM Orders WHERE Status = 'OPEN'",
SQL_NTS);
```

There are several advantages to hard-coded SQL statements: they can be tested when the application is written, they are simpler to implement than statements constructed at run time, and they simplify the application.

Using statement parameters and preparing statements provide even better ways to use hard-coded SQL statements. For example, suppose the Parts table contains the PartID, Description, and Price columns. One way to insert a new row into this table would be to construct and execute an **INSERT** statement:

```
#define DESC_LEN 51
#define STATEMENT_LEN 51

SQLINTEGER PartID;
SQLCHAR Desc[DESC_LEN], Statement[STATEMENT_LEN];
SQLREAL Price;

// Set part ID, description, and price.
GetNewValues(&PartID, Desc, &Price);

// Build INSERT statement.
sprintf(Statement, "INSERT INTO Parts (PartID, Description, Price) "
"VALUES (%d, '%s', %f)", PartID, Desc, Price);

// Execute the statement.
SQLExecDirect(hstmt, Statement, SQL_NTS);
```

An even better way is to use a hard-coded, parameterized statement. This has two advantages over a statement with hard-coded data values. First, it is easier to construct a parameterized statement because the data values can be sent in their native types, such as integers and floating point numbers, rather than converting them to strings. Second, such a statement can be used easily more than once by just changing the parameter values and re-executing it; there is no need to rebuild it.



```

#define DESC_LEN 51

SQLCHAR * Statement = "INSERT INTO Parts (PartID, Description, Price) "
    "VALUES (?, ?, ?)";
SQLINTEGER PartID;
SQLCHAR Desc[DESC_LEN];
SQLREAL Price;
SQLINTEGER PartIDInd = 0, DescLenOrInd = SQL_NTS, PriceInd = 0;

// Bind the parameters.
SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_ULONG, SQL_INTEGER, 5, 0,
    &PartID, 0, &PartIDInd);
SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR, DESC_LEN - 1,
    0,
    Desc, sizeof(Desc), &DescLenOrInd);
SQLBindParameter(hstmt, 3, SQL_PARAM_INPUT, SQL_C_FLOAT, SQL_REAL, 7, 0,
    &Price, 0, &PriceInd);

// Set part ID, description, and price.
GetNewValues(&PartID, Desc, &Price);

// Execute the statement.
SQLExecDirect(hstmt, Statement, SQL_NTS);

```

Assuming this statement is to be executed more than once, it can be prepared for even greater efficiency:

```

#define DESC_LEN 51

SQLCHAR *Statement = "INSERT INTO Parts (PartID, Description, Price) "
    "VALUES (?, ?, ?)";
SQLINTEGER PartID;
SQLCHAR Desc[DESC_LEN];
SQLREAL Price;
SQLINTEGER PartIDInd = 0, DescLenOrInd = SQL_NTS, PriceInd = 0;

// Prepare the INSERT statement.
SQLPrepare(hstmt, Statement, SQL_NTS);

// Bind the parameters.
SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_ULONG, SQL_INTEGER, 5, 0,
    &PartID, 0, &PartIDInd);
SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR, DESC_LEN - 1,
    0,
    Desc, sizeof(Desc), &DescLenOrInd);
SQLBindParameter(hstmt, 3, SQL_PARAM_INPUT, SQL_C_FLOAT, SQL_REAL, 7, 0,
    &Price, 0, &PriceInd);

// Loop to continually get new values and insert them.
while (GetNewValues(&PartID, Desc, &Price))
    SQLExecute(hstmt);

```

Perhaps the most efficient way to use the statement is to construct a procedure containing the statement, as shown in the following code example. Because the procedure is constructed at development time and stored on the data source, it does not need to be prepared at run time. A drawback of this method is that the syntax

for creating procedures is DBMS-specific and procedures must be constructed separately for each DBMS on which the application is to run.

```
#define DESC_LEN 51

SQLINTEGER PartID;
SQLCHAR Desc[DESC_LEN];
SQLREAL Price;
SQLINTEGER PartIDInd = 0, DescLenOrInd = SQL_NTS, PriceInd = 0;

// Bind the parameters.
SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_ULONG, SQL_INTEGER, 5, 0,
    &PartID, 0, &PartIDInd);
SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR, DESC_LEN - 1,
    0,
    Desc, sizeof(Desc), &DescLenOrInd);
SQLBindParameter(hstmt, 3, SQL_PARAM_INPUT, SQL_C_FLOAT, SQL_REAL, 7, 0,
    &Price, 0, &PriceInd);

// Loop to continually get new values and insert them.
while (GetNewValues(&PartID, Desc, &Price))
    SQLExecDirect(hstmt, "{call InsertPart(?, ?, ?)}", SQL_NTS);
```

For more information about parameters, prepared statements, and procedures, see [“Executing a Statement”](#) in Chapter 9, “Executing Statements.”

## SQL Statements Constructed at Run Time

Applications that perform ad hoc analysis commonly build SQL statements at run time. For example, a spreadsheet might allow a user to select columns from which to retrieve data:

```
SQLCHAR *Statement, *TableName;
SQLCHAR **TableNamesArray, **ColumnNamesArray;
BOOL *ColumnSelectedArray;
BOOL CommaNeeded;
SQLSMALLINT i, NumColumns;

// Use SQLTables to build a list of tables (TableNamesArray[]). Let the user
select a
// table and store the selected table in TableName.

// Use SQLColumns to build a list of the columns in the selected table
// (ColumnNamesArray). Set NumColumns to the number of columns in the table. Let
the
// user select one or more columns and flag these columns in
ColumnSelectedArray[].

// Build a SELECT statement from the selected columns.
CommaNeeded = FALSE;
strcpy(Statement, "SELECT ");
for (i = 0; i = NumColumns; i++) {
```

```

    if (ColumnSelectedArray[i]) {
        if (CommaNeeded)
            strcat(Statement, ",");
        else
            CommaNeeded = TRUE;
        strcat(Statement, ColumnNamesArray[i]);
    }
}

strcat(Statement, " FROM ");
strcat(Statement, TableName);

// Execute the statement directly. Because it will be executed only once, do not
// prepare it.
SQLExecDirect(hstmt, Statement, SQL_NTS);

```

Another class of applications that commonly constructs SQL statements at run time are application development environments. However, the statements they construct are hard-coded in the application they are building, where they can usually be optimized and tested.

Applications that construct SQL statements at run time can provide tremendous flexibility to the user. As can be seen from the preceding example, which did not even support such common operations as **WHERE** clauses, **ORDER BY** clauses, or joins, constructing SQL statements at run time is vastly more complex than hard-coding statements. Furthermore, testing such applications is problematic, because they can construct an arbitrary number of SQL statements.

A potential disadvantage of constructing SQL statements at run time is that it takes far more time to construct a statement than use a hard-coded statement. Fortunately, this is rarely a concern. Such applications tend to be user-interface intensive, and the time the application spends constructing SQL statements is generally small compared to the time the user spends entering criteria.

## SQL Statements Entered by the User

Applications that perform ad hoc analysis also commonly allow the user to enter SQL statements directly. For example:

```

SQLCHAR *Statement, SqlState[6], Msg[SQL_MAX_MESSAGE_LENGTH];
SQLSMALLINT i, MsgLen;
SQLINTEGER NativeError;
SQLRETURN rc1, rc2;

// Prompt user for SQL statement.
GetSQLStatement(Statement);

// Execute the statement directly. Because it will be executed only once, do not
// prepare it.
rc1 = SQLExecDirect(hstmt, Statement, SQL_NTS);

// Process any errors or returned information.
if ((rc1 == SQL_ERROR) || rc1 == SQL_SUCCESS_WITH_INFO) {

```

```

i = 1;
while ((rc2 = SQLGetDiagRec(SQL_HANDLE_STMT, hstmt, i, SqlState, &NativeError,
    Msg, sizeof(Msg), &MsgLen)) != SQL_NO_DATA) {
    DisplayError(SqlState, NativeError, Msg, MsgLen);
    i++;
}
}

```

This approach simplifies application coding; the application relies on the user to build the SQL statement and on the data source to check the statement's validity. Because it's difficult to write a graphical user interface that adequately exposes the intricacies of SQL, simply asking the user to enter the SQL statement text may be a preferable alternative. However, this requires the user to know not only SQL but also the schema of the data source being queried. Some applications provide a graphical user interface by which the user can create a basic SQL statement, and a text interface with which the user can modify it.

## Interoperability of SQL Statements

Like the rest of an application, SQL statements can be interoperable or DBMS-specific. And like the rest of the application, the choice of how interoperable SQL statements need to be depends on the type of application. Custom applications are less likely to use interoperable SQL statements because they are usually designed to exploit the capabilities of one or possibly two DBMSs. Generic applications use interoperable SQL statements because they are designed to work with a variety of DBMSs. And vertical applications usually fall somewhere in between, demanding a certain level of functionality but otherwise using interoperable SQL statements.

## Choosing an SQL Grammar

The first decision to make when constructing SQL statements is which grammar to use. In addition to the grammars available from the various standards bodies, such as X/Open, ANSI, and ISO, virtually every DBMS vendor defines its own grammar, each of which varies slightly from the standard.

Appendix C, "SQL Minimum Grammar," in the **SOLID Programmer Guide** describes the minimum SQL grammar that all ODBC drivers must support. This grammar is a subset of the Entry level of SQL92. Drivers may support additional grammar to conform to the Intermediate, Full, or FIPS 127-2 Transitional levels defined by SQL92. For more information, see Appendix C, "SQL Minimum Grammar," in the **SOLID Programmer Guide** and SQL92.

Appendix C also defines *escape sequences* containing standard grammar for commonly available language features, such as outer joins, that are not covered by the SQL92 grammar. For more information, see Appendix C, "SQL Minimum Grammar" in the **SOLID Programmer Guide** and "[Escape Sequences](#)," later in this chapter.

The grammar that is chosen affects how the driver processes the statement. Drivers must modify SQL92 SQL and the ODBC-defined escape sequences to DBMS-specific SQL. Because most SQL grammars are based on one or more of the various standards, most drivers do little or no work to meet this requirement. It often consists only of searching for the escape sequences defined by ODBC and replacing them with DBMS-specific grammar. When a driver encounters grammar it does not recognize, it assumes the grammar is DBMS-specific and passes the SQL statement without modification to the data source for execution.

Thus, there are really two choices of grammar to use: the SQL92 grammar (and the ODBC escape sequences) and a DBMS-specific grammar. Of the two, only the SQL92 grammar is interoperable, so all interoperable applications should use it. Applications that are not interoperable can use the SQL92 grammar or a DBMS-specific grammar. DBMS-specific grammars have two advantages: they can exploit any features not covered by SQL92 and they are marginally faster because the driver does not have to modify them. The latter feature can be partially enforced by setting the SQL\_ATTR\_NOSCAN statement attribute, which stops the driver from searching for and replacing escape sequences.

If the SQL92 grammar is used, the application can discover how it is modified by the driver by calling **SQLNativeSql**. This is often useful when debugging applications. **SQLNativeSql** accepts an SQL statement and returns it after the driver has modified it. Because this function is in the Core interface conformance level, it is supported by all drivers.

## Constructing Interoperable SQL Statements

As mentioned in the previous sections, interoperable applications should use the ODBC SQL grammar. Beyond using this grammar, however, there are a number of additional problems faced by interoperable applications. For example, what does an application do if it wants to use a feature, such as outer joins, that is not supported by all data sources?

At this point, the application writer must make some decisions about which language features are required and which are optional. Generally, if a particular driver does not support a feature required by the application, the application simply refuses to run with that driver. However, if the feature is optional, the application can work around the feature. For example, it might disable those parts of the interface that allow the user to use the feature.

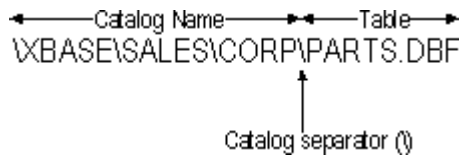
To determine which features are supported, applications start by calling **SQLGetInfo** with the SQL\_SQL\_CONFORMANCE option. The SQL conformance level gives the application a broad view of which SQL is supported. To refine this view, the application calls **SQLGetInfo** with any of a number of other options. For a complete list of these options, see SQLGetInfo function description in the Part II PDF file, “ODBC API Reference” available on the Solid Web site. Finally, **SQLGetTypeInfo** returns information about the data types supported by the data source. The following sections list a number of things that applications should watch for when constructing interoperable SQL statements.

## Catalog and Schema Usage

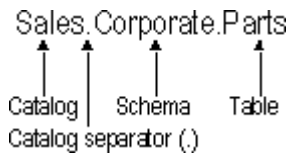
Data sources do not necessarily support catalog and schema names as object name identifiers in all SQL statements. Data sources might support catalog and schema names in one or more of the following classes of SQL statements: Data Manipulation Language (DML) statements, procedure calls, table definition statements, index definition statements, and privilege definition statements. To determine the classes of SQL statements in which catalog and schema names can be used, an application calls SQLGetInfo with the SQL\_CATALOG\_USAGE and SQL\_SCHEMA\_USAGE options.

## Catalog Position

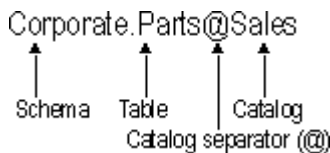
The position of a catalog name in an identifier and how it is separated from the rest of the identifier varies from data source to data source. For example, in an Xbase data source, the catalog name is a directory and, in Windows, is separated from the table name (which is a file name) by a backslash (\). The following figure illustrates this condition.



In a SQL Server data source, the catalog is a database and is separated from the schema and table names by a period (.).



In an Oracle data source, the catalog is also the database, but follows the table name and is separated from the schema and table names by an at sign (@).



To determine the catalog separator and the location of the catalog name, an application calls `SQLGetInfo` with the `SQL_CATALOG_NAME_SEPARATOR` and `SQL_CATALOG_LOCATION` options. Interoperable applications should construct identifiers according to these values.

When quoting identifiers that contain more than one part, applications must be careful to quote each part separately and not quote the character that separates the identifiers. For example, the following statement to select all of the rows and columns of an Xbase table quotes the catalog (`\XBASE\SALES\CORP`) and table (`PARTS.DBF`) names, but not the catalog separator (`\`):

```
SELECT * FROM "\XBASE\SALES\CORP\" "PARTS.DBF"
```

The following statement to select all of the rows and columns of an Oracle table quotes the catalog (`Sales`), schema (`Corporate`), and table (`Parts`) names, but not the catalog (`@`) or schema (`.`) separators:

```
SELECT * FROM "Corporate"."Parts"@Sales"
```

For information about quoting identifiers, see the next section, "Quoted Identifiers."

## Quoted Identifiers

In an SQL statement, identifiers containing special characters or match keywords must be enclosed in identifier quote characters; identifiers enclosed in such characters are known as quoted identifiers (also known as delimited identifiers in SQL92). For example, the Accounts Payable identifier is quoted in the following `SELECT` statement:

```
SELECT * FROM "Accounts Payable"
```

The reason for quoting identifiers is to make the statement parseable. For example, if `Accounts Payable` was not quoted in the previous statement, the parser would assume there were two tables, `Accounts` and `Payable`, and return a syntax error that they were not separated by a comma. The identifier quote character

is driver-specific and is retrieved with the `SQL_IDENTIFIER_QUOTE_CHAR` option in `SQLGetInfo`. The lists of special characters and of keywords are retrieved with the `SQL_SPECIAL_CHARACTERS` and `SQL_KEYWORDS` options in `SQLGetInfo`.

To be safe, interoperable applications often quote all identifiers except those for pseudo-columns, such as the `ROWID` column in Oracle. `SQLSpecialColumns` returns a list of pseudo-columns. Also, if there are application-specific restrictions on where special characters can appear in an object name, it is best for interoperable applications not to use special characters in those positions.

## Identifier Case

In SQL statements and catalog function arguments, identifiers and quoted identifiers can be either case sensitive or not. An application determines which they are by calling `SQLGetInfo` with the `SQL_IDENTIFIER_CASE` and `SQL_QUOTED_IDENTIFIER_CASE` options.

Each of these options has four possible return values: one stating that the identifier or quoted identifier case is sensitive and three stating that it is not sensitive. The three values that are not case sensitive further describe the case in which identifiers are stored in the system catalog. How identifiers are stored in the system catalog is relevant only for display purposes, such as when an application displays the results of a catalog function; it does not change the case sensitivity of identifiers.

## Escape Sequences

ODBC defines escape sequences containing standard grammar for date, time, timestamp, and datetime interval literals, scalar function calls, `LIKE` predicate escape characters, outer joins, and procedure calls. Interoperable applications should use these sequences whenever possible.

To determine if a driver supports the escape sequences for date, time, timestamp, or datetime interval literals, an application calls `SQLGetTypeInfo`. If the data source supports a date, time, timestamp, or datetime interval data type, it must also support the corresponding escape sequence. To determine if the other escape sequences are supported, an application calls `SQLGetInfo`.

For more information, see [“Escape Sequences in ODBC,”](#) later in this chapter.

## Literal Prefixes and Suffixes

In an SQL statement, a literal is a character representation of an actual data value. For example, in the following statement, `ABC`, `FFFF`, and `10` are literals:

```
SELECT CharCol, BinaryCol, IntegerCol FROM MyTable
WHERE CharCol = 'ABC' AND BinaryCol = 0xFFFF AND IntegerCol = 10
```

Literals for some data types require special prefixes and suffixes. In the preceding example, the character literal (`ABC`) requires a single quotation mark (`'`) as both a prefix and a suffix, the binary literal (`FFFF`) requires the characters `0x` as a prefix, and the integer literal (`10`) does not require a prefix or suffix.

For all data types except date, time, and timestamps, interoperable applications should use the values returned in the `LITERAL_PREFIX` and `LITERAL_SUFFIX` columns in the result set created by `SQLGetTypeInfo`. For date, time, timestamp, and datetime interval literals, interoperable applications should use the escape sequences discussed in the previous section.

## Parameter Markers in Procedure Calls

When calling procedures that accept parameters, interoperable applications should use parameter markers instead of literal parameter values. Some data sources do not support the use of literal parameter values in procedure calls. For more information about parameters, see “Statement Parameters” in Chapter 9, “Executing Statements.” For more information about calling procedures, see [“Procedure Calls,”](#) later in this chapter.

## DDL Statements

Data Definition Language (DDL) statements vary tremendously among DBMSs. ODBC SQL defines statements for the most common data definition operations: creating and dropping tables, indexes, and views; altering tables; and granting and revoking privileges. All other DDL statements are data source-specific. Thus, interoperable applications cannot perform some data definition operations. In general, this is not a problem, because such operations tend to be highly DBMS-specific and are best left to the proprietary database administration software shipped with most DBMSs or the setup program shipped with the driver.

Another problem in data definition is that data type names also vary tremendously among DBMSs. Rather than defining standard data type names and forcing drivers to convert them to DBMS-specific names, `SQLGetTypeInfo` provides a way for applications to discover DBMS-specific data type names. Interoperable applications should use these names in SQL statements to create and alter tables; the names listed in Appendix C, “SQL Minimum Grammar” and Appendix D, “Data Types” (both contained in the **SOLID Programmer Guide**) are examples only.



## Escape Sequences in ODBC

A number of language features, such as outer joins and scalar function calls, are commonly implemented by DBMSs. However, the syntaxes for these features tend to be DBMS-specific, even when standard syntaxes are defined by the various standards bodies. Because of this, ODBC defines escape sequences that contain standard syntaxes for the following language features:

- Date, time, timestamp, and datetime interval literals
- Scalar functions such as numeric, string, and data type conversion functions
- LIKE predicate escape character
- Outer joins
- Procedure calls

The escape sequence used by ODBC is as follows:

{extension}

The escape sequence is recognized and parsed by drivers, which replace the escape sequences with DBMS-specific grammar. For more information about escape sequence syntax, see Appendix C, “SQL Minimum Grammar” contained on the Microsoft Web site (ODBC Programmer’s Reference).

**Note** In ODBC 2.x, this was the standard syntax of the escape sequence:

--(\*vendor(vendor-name), product(product-name) extension \*)--

In addition to this syntax, a shorthand syntax was defined of the form: {extension}. In ODBC 3.x, the long form of the escape sequence has been deprecated, and the shorthand form is used exclusively.

Because the escape sequences are mapped by the driver to DBMS-specific syntaxes, an application can use either the escape sequence or DBMS-specific syntax. However, applications that use the DBMS-specific syntax will not be interoperable. When using the escape sequence, applications should make sure that the SQL\_ATTR\_NOSCAN statement attribute is turned off, which it is by default. Otherwise, the escape sequence will be sent directly to the data source, where it will generally cause a syntax error.

Drivers support only those escape sequences that they can map to underlying language features. For example, if the data source does not support outer joins, neither will the driver. To determine which escape sequences are supported, an application calls SQLGetTypeInfo and SQLGetInfo. For more information, see the next section, “Date, Time, and Timestamp Literals.”

## Date, Time, and Timestamp Literals

The escape sequence for date, time, and timestamp literals is:

{literal-type 'value'}

where literal-type is one of the following:

| literal-type | Meaning   | Format of value                |
|--------------|-----------|--------------------------------|
| d            | Date      | yyyy-mm-dd                     |
| t            | Time      | hh:mm:ss [1]                   |
| ts           | Timestamp | yyyy-mm-dd hh:mm:ss[.f...] [1] |

[1] The number of digits to the right of the decimal point in a time or timestamp interval literal containing a seconds component is dependent upon the seconds precision, as contained in the SQL\_DESC\_PRECISION descriptor field (for more information, see SQLSetDescField).

For more information about the date, time, and timestamp escape sequences, see “Date, Time, and Timestamp Escape Sequences” in Appendix C, “SQL Minimum Grammar” contained on the Microsoft Web site (ODBC Programmer’s Reference).

For example, both of the following SQL statements update the open date of sales order 1023 in the Orders table. The first statement uses the escape sequence syntax. The second statement uses the native syntax for a DATE column in the Rdb from Digital Equipment Corporation; this statement is not interoperable.

```
UPDATE Orders SET OpenDate={d '1995-01-15'} WHERE OrderID=1023
```

```
UPDATE Orders SET OpenDate='15-Jan-1995' WHERE OrderID=1023
```

The escape sequence for a date, time, or timestamp literal also can be placed in a character variable bound to a date, time, or timestamp parameter. For example, the following code uses a date parameter bound to a character variable to update the open date of sales order 1023 in the Orders table:

```
SQLCHAR OpenDate[56]; // The size of a date literal is 55.
SQLINTEGER OpenDateLenOrInd = SQL_NTS;

// Bind the parameter.
SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_TYPE_DATE, 0, 0,
    OpenDate, sizeof(OpenDate), &OpenDateLenOrInd);

// Place the date in the OpenDate variable. In addition to the escape sequence
shown,
// it would also be possible to use either of the strings "{d '1995-01-15'}" and
// "15-Jan-1995", although the latter is data source-specific.
strcpy(OpenDate, "{d '1995-01-15'}");

// Execute the statement.
SQLExecDirect(hstmt, "UPDATE Orders SET OpenDate=? WHERE OrderID = 1023",
    SQL_NTS);
```

However, it is usually more efficient to bind the parameter directly to a date structure:

```
SQL_DATE_STRUCT OpenDate;
SQLINTEGER OpenDateInd = 0;

// Bind the parameter.
```

```

SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_TYPE_DATE, SQL_TYPE_DATE, 0,
0,
    &OpenDate, 0, &OpenDateLen);

// Place the date in the dsOpenDate structure.
OpenDate.year = 1995;
OpenDate.month = 1;
OpenDate.day = 15;

// Execute the statement.
SQLExecDirect(hstmt, "UPDATE Employee SET OpenDate=? WHERE OrderID = 1023",
SQL_NTS);

```

To determine if a driver supports the ODBC escape sequences for date, time, or timestamp literals, an application calls `SQLGetTypeInfo`. If the data source supports a date, time, or timestamp data type, it must also support the corresponding escape sequence.

Data sources can also support the datetime literals defined in the ANSI SQL92 specification, which are different from the ODBC escape sequences for date, time, or timestamp literals. To determine if a data source supports the ANSI literals, an application calls `SQLGetInfo` with the `SQL_ANSI_SQL_DATETIME_LITERALS` option.

To determine if a driver supports the ODBC escape sequences for interval literals, an application calls `SQLGetTypeInfo`. If the data source supports a datetime interval data type, it must also support the corresponding escape sequence.

Data sources can also support the datetime literals defined in the ANSI SQL92 specification, which are different from the ODBC escape sequences for datetime interval literals. To determine if a data source supports the ANSI literals, an application calls `SQLGetInfo` with the `SQL_ANSI_SQL_DATETIME_LITERALS` option.

## Scalar Function Calls

Scalar functions return a value for each row. For example, the absolute value scalar function takes a numeric column as an argument and returns the absolute value of each value in the column. The escape sequence for calling a scalar function is:

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

where `scalar-function` is one of the functions listed in Appendix E, “Scalar Functions” in the **SOLID Programmer Guide**. For more information on the scalar function escape sequence, see “Scalar Functions Escape Sequence” in Appendix C, “SQL Grammar,” contained on the Microsoft Web site (ODBC Programmer’s Guide).

For example, the following SQL statements create the same result set of uppercase customer names. The first statement uses the escape-sequence syntax. The second statement uses the native syntax for Ingres for OS/2 and is not interoperable.

```

SELECT {fn UCASE(Name)} FROM Customers

SELECT uppercase(Name) FROM Customers

```

An application can mix calls to scalar functions that use native syntax and calls to scalar functions that use ODBC syntax. For example, assume that names in the Employee table are stored as a last name, a comma, and a first name. The following SQL statement creates a result set of last names of employees in the Employee table. The statement uses the ODBC scalar function SUBSTRING and the SQL Server scalar function CHARINDEX, and will execute correctly only on SQL Server.

```
SELECT {fn SUBSTRING(Name, 1, CHARINDEX(',', Name) - 1)} FROM Customers
```

For maximum interoperability, applications should use the CONVERT scalar function to make sure the output of a scalar function is the required type. The CONVERT function converts data from one SQL data type to the specified SQL data type. The syntax of the CONVERT function is:

**CONVERT**(value\_exp, data\_type)

where value\_exp is a column name, the result of another scalar function, or a literal value, and data\_type is a keyword that matches the #define name used by an SQL data type identifier as defined in Appendix D, “Data Types.” For example, the following SQL statement uses the CONVERT function to make sure that the output of the CURDATE function is a date, rather than a timestamp or character data:

```
INSERT INTO Orders (OrderID, CustID, OpenDate, SalesPerson, Status)
VALUES (?, ?, {fn CONVERT({fn CURDATE()}, SQL_DATE)}, ?, ?)
```

To determine which scalar functions are supported by a data source, an application calls SQLGetInfo with the SQL\_CONVERT\_FUNCTIONS, SQL\_NUMERIC\_FUNCTIONS, SQL\_STRING\_FUNCTIONS, SQL\_SYSTEM\_FUNCTIONS, and SQL\_TIMEDATE\_FUNCTIONS options. To determine which conversion operations are supported by the CONVERT function, an application calls SQLGetInfo with any of the options that start with SQL\_CONVERT.

## LIKE Predicate Escape Character

In a LIKE predicate, the percent sign (%) matches zero or more of any character and the underscore(\_) matches any one character. To match an actual percent sign or underscore in a LIKE predicate, an escape character must precede the percent sign or underscore. The escape sequence that defines the LIKE predicate escape character is:

```
{escape 'escape-character'}
```

where escape-character is any character supported by the data source.

For more information on the LIKE escape sequence, see “LIKE Escape Sequence” in Appendix C, “SQL Minimum Grammar” contained on the Microsoft Web site (ODBC Programmer’s Guide)...

For example, the following SQL statements create the same result set of customer names that start with the characters “%AAA”. The first statement uses the escape-sequence syntax. The second statement uses the native syntax for Microsoft Access and is not interoperable. Note that the second percent character in each LIKE predicate is a wild card that matches zero or more of any character.

```
SELECT Name FROM Customers WHERE Name LIKE '\%AAA%' {escape '\'}
```

```
SELECT Name FROM Customers WHERE Name LIKE '[%]AAA%'
```

To determine whether the LIKE predicate escape character is supported by a data source, an application calls SQLGetInfo with the SQL\_LIKE\_ESCAPE\_CLAUSE option.

## Outer Joins

ODBC supports the SQL92 left, right, and full outer join syntax. The escape sequence for outer joins is:

**{oj outer-join}**

where outer-join is:

table-reference {LEFT | RIGHT | FULL} OUTER JOIN  
{table-reference | outer-join} ON search-condition

table-reference specifies a table name, and search-condition specifies the join condition between the table-references.

An outer join request must appear after the FROM keyword and before the WHERE clause (if one exists). For complete syntax information, see “Outer Join Escape Sequence” in Appendix C, “SQL Minimum Grammar,” contained on the Microsoft Web site (ODBC Programmer’s Guide).

For example, the following SQL statements create the same result set that lists all customers and shows which has open orders. The first statement uses the escape-sequence syntax. The second statement uses the native syntax for Oracle and is not interoperable.

```
SELECT Customers.CustID, Customers.Name, Orders.OrderID, Orders.Status
FROM {oj Customers LEFT OUTER JOIN Orders ON Customers.CustID=Orders.CustID}
WHERE Orders.Status='OPEN'
```

```
SELECT Customers.CustID, Customers.Name, Orders.OrderID, Orders.Status
FROM Customers, Orders
WHERE (Orders.Status='OPEN') AND (Customers.CustID= Orders.CustID(+))
```

To determine the types of outer joins that a data source and driver support, an application calls SQLGetInfo with the SQL\_OJ\_CAPABILITIES flag. The types of outer joins that might be supported are left, right, full, or nested outer joins; outer joins in which the column names in the ON clause do not have the same order as their respective table names in the OUTER JOIN clause; inner joins in conjunction with outer joins; and outer joins using any ODBC comparison operator. If the SQL\_OJ\_CAPABILITIES information type returns 0, no outer join clause is supported.

## Procedure Calls

A procedure is an executable object stored on the data source. Generally, it is one or more SQL statements that have been precompiled. The escape sequence for calling a procedure is:

**{[?]=call procedure-name[(parameter)[parameter]...]}**

where procedure-name specifies the name of a procedure and parameter specifies a procedure parameter.

For more information on the procedure call escape sequence, see “Procedure Call Escape Sequence” in Appendix C, “SQL Minimum Grammar,” contained on the Microsoft Web site (ODBC Programmer’s Guide).

A procedure can have zero or more parameters. It can also return a value, as indicated by the optional parameter marker ?= at the start of the syntax. If parameter is an input or an input/output parameter, it can

be a literal or a parameter marker. However, interoperable applications should always use parameter markers, because some data sources do not accept literal parameter values. If parameter is an output parameter, it must be a parameter marker. Parameter markers must be bound with `SQLBindParameter` before the procedure call statement is executed.

Input and input/output parameters can be omitted from procedure calls. If a procedure is called with parentheses but without any parameters, such as `{call procedure-name()}`, the driver instructs the data source to use the default value for the first parameter. If the procedure does not have any parameters, this may cause the procedure to fail. If a procedure is called without parentheses, such as `{call procedure-name}`, the driver does not send any parameter values.

Literals can be specified for input and input/output parameters in procedure calls. For example, suppose the procedure `InsertOrder` has five input parameters. The following call to `InsertOrder` omits the first parameter, provides a literal for the second parameter, and uses a parameter marker for the third, fourth, and fifth parameters:

```
{call InsertOrder(, 10, ?, ?, ?)} // Not interoperable!
```

Note that if a parameter is omitted, the comma delimiting it from other parameters must still appear. If an input or input/output parameter is omitted, the procedure uses the default value of the parameter. Another way to specify the default value of an input or input/output parameter is to set the value of the length/indicator buffer bound to the parameter to `SQL_DEFAULT_PARAM`.

If an input/output parameter is omitted or if a literal is supplied for the parameter, the driver discards the output value. Similarly, if the parameter marker for the return value of a procedure is omitted, the driver discards the return value. Finally, if an application specifies a return value parameter for a procedure that does not return a value, the driver sets the value of the length/indicator buffer bound to the parameter to `SQL_NULL_DATA`.

Suppose the procedure `PARTS_IN_ORDERS` creates a result set containing a list of orders which contain a particular part number. The following code calls this procedure for part number 544:

```
SQLINTEGER PartID;
SQLINTEGER PartIDInd = 0;

// Bind the parameter.
SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_SLONG, SQL_INTEGER, 0, 0,
    &PartID, 0, PartIDInd);

// Place the department number in PartID.
PartID = 544;

// Execute the statement.
SQLExecDirect(hstmt, "{call PARTS_IN_ORDERS(?)}", SQL_NTS);
```

To determine if a data source supports procedures, an application calls `SQLGetInfo` with the `SQL_PROCEDURES` option.

For more information about procedures, see [“Procedures”](#) in Chapter 9, “Executing Statements.”

## Chapter 9: Overview of Executing Statements

ODBC applications perform almost all database access by executing SQL statements. The general sequence of events is to allocate a statement handle, set any statement attributes, execute the statement, retrieve any results, and free the statement handle.

### Allocating a Statement Handle

Before the application can execute a statement, it must allocate a statement handle. To do this:

1. The application declares a variable of type HSTMT. It then calls `SQLAllocHandle` and passes the address of this variable, the handle of the connection in which to allocate the statement, and the `SQL_HANDLE_STMT` option. For example:

```
SQLHSTMT hstmt1;  
SQLAllocHandle(SQL_HANDLE_STMT, hdbc1, &hstmt1);
```

2. The Driver Manager allocates a structure in which to store information about the statement and calls `SQLAllocHandle` in the driver with the `SQL_HANDLE_STMT` option.
3. The driver allocates its own structure in which to store information about the statement and returns the driver statement handle to the Driver Manager.
4. The Driver Manager returns the Driver Manager statement handle to the application in the application variable.

The statement handle identifies which statement to use when calling ODBC functions. For more information about statement handles, see [“Statement Handles”](#) in Chapter 4, “ODBC Fundamentals.”

### Statement Attributes

Statement attributes are characteristics of the statement. For example, whether to use bookmarks and what kind of cursor to use with the statement’s result set are statement attributes.

Statement attributes are set with `SQLSetStmtAttr` and their current settings retrieved with `SQLGetStmtAttr`. There is no requirement that an application set any statement attributes; all statement attributes have defaults, some of which are driver-specific.

When a statement attribute can be set depends on the attribute itself. The `SQL_ATTR_CONCURRENCY`, `SQL_ATTR_CURSOR_TYPE`, `SQL_ATTR_SIMULATE_CURSOR`, and `SQL_ATTR_USE_BOOKMARKS` statement attributes must be set before the statement is executed. The `SQL_ATTR_ASYNC_ENABLE` and `SQL_ATTR_NOSCAN` statement attributes can be set at any time, but are not applied until the statement is used again. `SQL_ATTR_MAX_LENGTH`, `SQL_ATTR_MAX_ROWS`, and `SQL_ATTR_QUERY_TIMEOUT` statement attributes can be set at any time, but it is driver-specific whether they are applied before the statement is used again. The remaining statement attributes can be set at any time.

**Note** The ability to set statement attributes at the connection level by calling `SQLSetConnectAttr` has been deprecated in ODBC 3.x. ODBC 3.x applications should never set statement attributes at the connection level. ODBC 3.x drivers need only support this functionality if they should work with ODBC 2.x applications. For more information, see “`SQLSetConnectOption` Mapping” in Appendix G, “Driver Guidelines for Backward Compatibility” contained on the Microsoft Web site (ODBC Programmer’s Reference).

An exception to this is the `SQL_ATTR_METADATA_ID` and `SQL_ATTR_ASYNC_ENABLE` attributes, which are both connection attributes and statement attributes, and can be set either at the connection level or the statement level.

None of the statement attributes introduced in ODBC 3.x (except for `SQL_ATTR_METADATA_ID`) can be set at the connection level.

For more information, see the `SQLSetStmtAttr` function description in the Part II PDF file, “ODBC API Reference,” available on the Solid Web site.

## Executing a Statement

There are four ways to execute a statement, depending on when they are compiled (prepared) by the database engine and who defines them:

- **Direct execution.** The application defines the SQL statement. It is prepared and executed at run time in a single step.
- **Prepared execution.** The application defines the SQL statement. It is prepared and executed at run time in separate steps. The statement can be prepared once and executed multiple times.
- **Procedures.** The application can define and compile one or more SQL statements at development time and store these statements on the data source as a procedure. The procedure is executed one or more times at run time. The application can enumerate available stored procedures using catalog functions.
- **Catalog functions.** The driver writer creates a function that returns a predefined result set. Usually, this function submits a predefined SQL statement or calls a procedure created for this purpose. The function is executed one or more times at run time.

A particular statement (as identified by its statement handle) can be executed any number of times. The statement can be executed with a variety of different SQL statements, or it can be executed repeatedly with the same SQL statement. For example, the following code uses the same statement handle (*hstmt1*) to retrieve and display the tables in the Sales database. It then reuses this handle to retrieve the columns in a table selected by the user.

```
SQLHSTMT    hstmt1;  
SQLCHAR *   Table;
```

```
// Create a result set of all tables in the Sales database.  
SQLTables(hstmt1, "Sales", SQL_NTS, "sysadmin", SQL_NTS, NULL, 0, NULL, 0);
```



```
// Fetch and display the table names; then close the cursor.
// Code not shown.

// Have the user select a particular table.
SelectTable(Table);

// Reuse hstmt1 to create a result set of all columns in Table.
SQLColumns(hstmt1, "Sales", SQL_NTS, "sysadmin", SQL_NTS, Table, SQL_NTS, NULL,
0);

// Fetch and display the column names in Table; then close the cursor.
// Code not shown.
```

And the following code shows how a single handle is used to repeatedly execute the same statement to delete rows from a table.

```
SQLHSTMT      hstmt1;
SQLINTEGER    OrderID;
SQLINTEGER    OrderIDInd = 0;

// Prepare a statement to delete orders from the Orders table.
SQLPrepare(hstmt1, "DELETE FROM Orders WHERE OrderID = ?", SQL_NTS);

// Bind OrderID to the parameter for the OrderID column.
SQLBindParameter(hstmt1, 1, SQL_PARAM_INPUT, SQL_C_ULONG, SQL_INTEGER, 5, 0,
                  &OrderID, 0, &OrderIDInd);

// Repeatedly execute hstmt1 with different values of OrderID.
while ((OrderID = GetOrderID()) != 0) {
    SQLExecute(hstmt1);
}
```

For many drivers, allocating statements is an expensive task, so reusing the same statement in this manner is usually more efficient than freeing existing statements and allocating new ones. Applications that create result sets on a statement must be careful to close the cursor over the result set before reexecuting the statement; for more information, see "[Closing the Cursor](#)" in Chapter 10, "Retrieving Results (Basic)."

Reusing statements also forces the application to avoid a limitation in some drivers of the number of statements that can be active at one time. The exact definition of "active" is driver-specific, but it often refers to any statement that has been prepared or executed and still has results available. For example, after an **INSERT** statement has been prepared, it is generally considered to be active; after a **SELECT** statement has been executed and the cursor is still open, it is generally considered to be active; after a **CREATE TABLE** statement has been executed, it is not generally considered to be active.

An application determines how many statements can be active on a single connection at one time by calling **SQLGetInfo** with the **SQL\_MAX\_CONCURRENT\_ACTIVITIES** option. An application can use more active statements than this limit by opening multiple connections to the data source; because connections can be expensive, however, the effect on performance should be considered.

Applications can limit the amount of time allotted for a statement to execute with the **SQL\_ATTR\_QUERY\_TIMEOUT** statement attribute. If the timeout period expires before the data source

returns the result set, the function executing the SQL statement returns SQLSTATE HYT00 (Timeout expired). By default, there is no timeout.

## Direct Execution

Direct execution is the simplest way to execute a statement. When the statement is submitted for execution, the data source compiles it into an access plan and then executes that access plan.

Direct execution is commonly used by generic applications that build and execute statements at run time. For example, the following code builds an SQL statement and executes it a single time:

```
SQLCHAR *SQLStatement;  
  
// Build an SQL statement.  
BuildStatement(SQLStatement);  
  
// Execute the statement.  
SQLExecDirect(hstmt, SQLStatement, SQL_NTS);
```

Direct execution works best for statements that will be executed a single time. Its major drawback is that the SQL statement is parsed every time it is executed. In addition, the application cannot retrieve information about the result set created by the statement (if any) until after the statement is executed; this is possible if the statement is prepared and executed in two separate steps.

To execute a statement directly, the application:

1. Sets the values of any parameters. For more information, see “[Statement Parameters](#),” later in this chapter.
2. Calls SQLExecDirect and passes it a string containing the SQL statement.
3. When SQLExecDirect is called, the driver:
  - Modifies the SQL statement to use the data source’s SQL grammar without parsing the statement; this includes replacing the escape sequences discussed in “[Escape Sequences in ODBC](#)” in Chapter 8, “SQL Statements.” The application can retrieve the modified form of an SQL statement by calling SQLNativeSql. Note that escape sequences are not replaced if the SQL\_ATTR\_NOSCAN statement attribute is set.
  - Retrieves the current parameter values and converts them as necessary. For more information, see “[Statement Parameters](#),” later in this chapter.
  - Sends the statement and converted parameter values to the data source for execution.
  - Returns any errors. These include sequencing or state diagnostics such as SQLSTATE 24000 (Invalid cursor state), syntactic errors such as SQLSTATE 42000 (Syntax error or access violation), and semantic errors such as SQLSTATE 42S02 (Base table or view not found).

## Prepared Execution

Prepared execution is an efficient way to execute a statement more than once. The statement is first compiled, or prepared, into an access plan. The access plan is then executed one or more times at a later time. For more information about access plans, see “[Processing an SQL Statement](#)” in Chapter 2, “An Introduction to SQL and ODBC.”

Prepared execution is commonly used by vertical and custom applications to repeatedly execute the same, parameterized SQL statement. For example, the following code prepares a statement to update the prices of different parts. It then executes the statement multiple times with different parameter values each time.

```
SQLREAL    Price;
SQLINTEGER PartID;
SQLINTEGER PartIDInd = 0, PriceInd = 0;

// Prepare a statement to update salaries in the Employees table.
SQLPrepare(hstmt, "UPDATE Parts SET Price = ? WHERE PartID = ?", SQL_NTS);

// Bind Price to the parameter for the Price column and PartID to
// the parameter for the PartID column.
SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_FLOAT, SQL_REAL, 7, 0,
    &Price, 0, &PriceInd);
SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT, SQL_C_ULONG, SQL_INTEGER, 10, 0,
    &PartID, 0, &PartIDInd);

// Repeatedly execute the statement.
while (GetPrice(&PartID, &Price)) {
    SQLExecute(hstmt);
}
```

Prepared execution is faster than direct execution for statements executed more than once, primarily because the statement is compiled only once; statements executed directly are compiled each time they are executed. Prepared execution also can provide a reduction in network traffic because the driver can send an access plan identifier to the data source each time the statement is executed, rather than an entire SQL statement, if the data source supports access plan identifiers.

The application can retrieve the metadata for the result set after the statement is prepared and before it is executed. However, returning metadata for prepared, unexecuted statements is expensive for some drivers and should be avoided by interoperable applications if possible. For more information, see “[Result Set Metadata](#)” in Chapter 10, “Retrieving Results (Basic).”

Prepared execution should not be used for statements executed a single time. For such statements, it is slightly slower than direct execution because it requires an additional ODBC function call.

**Important** Committing or rolling back a transaction, either by explicitly calling `SQLEndTran` or by working in autocommit mode, causes some data sources to delete the access plans for all statements on a connection. For more information, see the `SQL_CURSOR_COMMIT_BEHAVIOR` and `SQL_CURSOR_ROLLBACK_BEHAVIOR` options in the `SQLGetInfo` function description.

To prepare and execute a statement, the application:

1. Calls `SQLPrepare` and passes it a string containing the SQL statement.

2. Sets the values of any parameters. Parameters can actually be set before or after preparing the statement. For more information, see [“Statement Parameters,”](#) later in this chapter.
3. Calls `SQLExecute` and does any additional processing that is necessary, such as fetching data.
4. Repeats steps 2 and 3 as necessary.
5. When `SQLPrepare` is called, the driver:
  - Modifies the SQL statement to use the data source’s SQL grammar without parsing the statement. This includes replacing the escape sequences discussed in [“Escape Sequences in ODBC”](#) in Chapter 8, “SQL Statements.” The application can retrieve the modified form of an SQL statement by calling `SQLNativeSql`. Note that escape sequences are not replaced if the `SQL_ATTR_NOSCAN` statement attribute is set.
  - Sends the statement to the data source for preparation.
  - Stores the returned access plan identifier for later execution (if the preparation succeeded) or returns any errors (if the preparation failed). Errors include syntactic errors such as `SQLSTATE 42000` (Syntax error or access violation) and semantic errors such as `SQLSTATE 42S02` (Base table or view not found).

Note Some drivers do not return errors at this point, but instead return them when the statement is executed or when catalog functions are called. Thus, `SQLPrepare` might appear to have succeeded when in fact it has failed.

6. When `SQLExecute` is called, the driver:
  - Retrieves the current parameter values and converts them as necessary. For more information, see [“Statement Parameters,”](#) later in this chapter.
  - Sends the access plan identifier and converted parameter values to the data source.
  - Returns any errors. These are generally run-time errors such as `SQLSTATE 24000` (Invalid cursor state). However, some drivers return syntactic and semantic errors at this point.

If the data source does not support statement preparation, the driver must emulate it to the extent possible. For example, the driver might do nothing when `SQLPrepare` is called, then perform direct execution of the statement when `SQLExecute` is called.

If the data source supports syntax checking without execution, the driver might submit the statement for checking when `SQLPrepare` is called and submit the statement for execution when `SQLExecute` is called.

If the driver cannot emulate statement preparation, it stores the statement when SQLPrepare is called and submits it for execution when SQLExecute is called.

Because emulated statement preparation is not perfect, SQLExecute can return any errors normally returned by SQLPrepare.

## Procedures

A procedure is an executable object stored on the data source. Generally, it is one or more SQL statements that have been precompiled.

## When to Use Procedures

There are a number of advantages to using procedures, all based on the fact that using procedures moves SQL statements from the application to the data source. What is left in the application is an interoperable procedure call. These advantages include:

- **Performance.** Procedures are usually the fastest way to execute SQL statements. Like prepared execution, the statement is compiled and executed in two separate steps. Unlike prepared execution, procedures are executed only at run time. They are compiled at a different time.
- **Business rules.** A business rule is a rule about the way in which a company does business. For example, only someone with the title Sales Person might be allowed to add new sales orders. Placing these rules in procedures allows individual companies to customize vertical applications by rewriting the procedures called by the application without having to modify the application code. For example, an order entry application might call the procedure InsertOrder with a fixed number of parameters; exactly how InsertOrder is implemented can vary from company to company.
- **Replaceability.** Closely related to placing business rules in procedures is the fact that procedures can be replaced without recompiling the application. If a business rule changes after a company has bought and installed an application, the company can change the procedure containing that rule. From the application's standpoint, nothing has changed; it still calls a particular procedure to accomplish a particular task.
- **DBMS-specific SQL.** Procedures provide a way for applications to exploit DBMS-specific SQL and still remain interoperable. For example, a procedure on a DBMS that supports control-of-flow statements in SQL might trap and recover from errors, while a procedure on a DBMS that does not support control-of-flow statements might simply return an error.
- **Procedures survive transactions.** On some data sources, the access plans for all prepared statements on a connection are deleted when a transaction is committed or rolled back. By placing SQL statements in procedures, which are permanently stored in the data source, the statements survive the transaction. Whether the procedures survive in a prepared, partially prepared, or unprepared state is DBMS-specific.

- **Separate development.** Procedures can be developed separately from the rest of the application. In large corporations, this might provide a way to further exploit the skills of highly specialized programmers. In other words, application programmers can write user-interface code and database programmers can write procedures.

Procedures are generally used by vertical and custom applications. These applications tend to perform fixed tasks, and it is possible to hard-code procedure calls in them. For example, an order entry application might call the procedures `InsertOrder`, `DeleteOrder`, `UpdateOrder`, and `GetOrders`.

There is little reason to call procedures from generic applications. Procedures are generally written to perform a task in the context of a particular application and so have no use to generic applications. For example, a spreadsheet has no reason to call the `InsertOrder` procedure just mentioned. Furthermore, generic applications should not construct procedures at run time in hopes of providing faster statement execution; not only is this likely to be slower than prepared or direct execution, it also requires DBMS-specific SQL statements.

An exception to this is application-development environments, which often provide a way for programmers to build SQL statements that execute procedures and may provide a way for programmers to test procedures. Such environments call `SQLProcedures` to list available procedures and `SQLProcedureColumns` to list the input, input/output, and output parameters, the procedure return value, and the columns of any result sets created by a procedure. However, such procedures must be developed beforehand on each data source; doing so requires DBMS-specific SQL statements.

There are three major disadvantages to using procedures. The first is that procedures must be written and compiled for each DBMS with which the application is to run. While this is not a problem for custom applications, it can significantly increase development and maintenance time for vertical applications designed to run with a number of DBMSs.

The second disadvantage is that many DBMSs do not support procedures. Again, this is most likely to be a problem for vertical applications designed to run with a number of DBMSs. To determine whether procedures are supported, an application calls `SQLGetInfo` with the `SQL_PROCEDURES` option.

The third disadvantage, which is particularly applicable to application development environments, is that ODBC does not define a standard grammar for creating procedures. Thus, although applications can call procedures interoperably, they cannot create them interoperably.

## Executing Procedures

ODBC defines a standard escape sequence for executing procedures. For the syntax of this sequence and a code example that uses it, see “[Procedure Calls](#)” in Chapter 8, “SQL Statements.”

To execute a procedure, an application:

1. Sets the values of any parameters. For more information, see “[Statement Parameters](#),” later in this chapter.
2. Calls `SQLExecDirect` and passes it a string containing the SQL statement that executes the procedure. This statement can use the escape sequence defined by ODBC or DBMS-specific syntax; statements that use DBMS-specific syntax are not interoperable.

3. When `SQLExecDirect` is called, the driver:

- Retrieves the current parameter values and converts them as necessary. For more information, see “Statement Parameters,” later in this chapter.
- Calls the procedure in the data source and sends it the converted parameter values. How the driver calls the procedure is driver-specific. For example, it might modify the SQL statement to use the data source’s SQL grammar and submit this statement for execution, or it might call the procedure directly using a Remote Procedure Call (RPC) mechanism that is defined in the data stream protocol of the DBMS.
- Returns the values of any input/output or output parameters or the procedure return value, assuming the procedure succeeds. Note that these values might not be available until after all other results (row counts and result sets) generated by the procedure have been processed. If the procedure fails, the driver returns any errors.

## Batches of SQL Statements

A batch of SQL statements is a group of two or more SQL statements or a single SQL statement that has the same effect as a group of two or more SQL statements. In some implementations, the entire batch statement is executed before any results are available. This is often more efficient than submitting statements separately, because network traffic can often be reduced and the data source can sometimes optimize execution of a batch of SQL statements. In other implementations, calling `SQLMoreResults` triggers the execution of the next statement in the batch. ODBC supports the following types of batches:

- Explicit batches. An explicit batch is two or more SQL statements separated by semicolons (;). For example, the following batch of SQL statements opens a new sales order. This requires inserting rows into both the `Orders` and `Lines` tables. Note that there is no semicolon after the last statement.

```
INSERT INTO Orders (OrderID, CustID, OpenDate, SalesPerson, Status)
VALUES (2002, 1001, {fn CURDATE()}, 'Garcia', 'OPEN');

INSERT INTO Lines (OrderID, Line, PartID, Quantity)
VALUES (2002, 1, 1234, 10);

INSERT INTO Lines (OrderID, Line, PartID, Quantity)
VALUES (2002, 2, 987, 8);

INSERT INTO Lines (OrderID, Line, PartID, Quantity)
VALUES (2002, 3, 566, 17);

INSERT INTO Lines (OrderID, Line, PartID, Quantity)
VALUES (2002, 4, 412, 500)
```

- If a procedure contains more than one SQL statement, it is considered to be a batch of SQL statements. For example, the following SQL Server – specific statement creates a procedure that returns a result set containing information about a customer and a result set listing all the open sales orders for that customer:

```
CREATE PROCEDURE GetCustInfo (@CustomerID INT) AS
SELECT * FROM Customers WHERE CustID = @CustomerID
SELECT OrderID FROM Orders
WHERE CustID = @CustomerID AND Status = 'OPEN'
```

The **CREATE PROCEDURE** statement itself is not a batch of SQL statements. However, the procedure it creates is a batch of SQL statements. Note that no semicolons separate the two **SELECT** statements because the **CREATE PROCEDURE** statement is specific to SQL Server, and SQL Server does not require semicolons to separate multiple statements in a **CREATE PROCEDURE** statement.

## Result-Generating and Result-Free Statements

SQL statements can be loosely divided into the following five categories:

- **Result set–generating statements.** These are SQL statements that generate a result set. For example, a **SELECT** statement.
- **Row count–generating statements.** These are SQL statements that generate a count of affected rows. For example, an **UPDATE** or **DELETE** statement.
- **Data Definition Language (DDL) statements.** These are SQL statements that modify the structure of the database. For example, **CREATE TABLE** or **DROP INDEX**.
- **Context-changing statements.** These are SQL statements that change the context of a database. For example, the **USE** and **SET** statements in SQL Server.
- **Administrative statements.** These are SQL statements used for administrative purposes in a database. For example, **GRANT** and **REVOKE**.

SQL statements in the first two categories are collectively known as *result-generating statements*. SQL statements in the latter three categories are collectively known as *result-free statements*. ODBC defines the semantics of batches that include only result-generating statements. These semantics vary widely and are therefore data source–specific. For example, the SQL Server driver does not support dropping an object and then referring to or re-creating the same object in the same batch. Therefore, the term *batch* as used in this manual refers only to batches of result-generating statements.

## Executing Batches

Before an application executes a batch of statements, it should first check whether they are supported. To do this, the application calls `SQLGetInfo` with the `SQL_BATCH_SUPPORT`, `SQL_PARAM_ARRAY_ROW_COUNTS`, and `SQL_PARAM_ARRAY_SELECTS` options. The first option returns whether row count–generating and result set–generating statements are supported in explicit



batches and procedures, while the latter two options return information about the availability of row counts and result sets in parameterized execution.

Batches of statements are executed through `SQLExecute` or `SQLExecDirect`. For example, the following call executes an explicit batch of statements to open a new sales order.

```
SQLCHAR *BatchStmt =
    "INSERT INTO Orders (OrderID, CustID, OpenDate, SalesPerson, Status)"
    "VALUES (2002, 1001, {fn CURDATE()}, 'Garcia', 'OPEN');"
    "INSERT INTO Lines (OrderID, Line, PartID, Quantity) VALUES (2002, 1, 1234,"
    "10);"
    "INSERT INTO Lines (OrderID, Line, PartID, Quantity) VALUES (2002, 2, 987, 8);"
    "INSERT INTO Lines (OrderID, Line, PartID, Quantity) VALUES (2002, 3, 566, 17);"
    "INSERT INTO Lines (OrderID, Line, PartID, Quantity) VALUES (2002, 4, 412,"
    "500)";
```

```
SQLExecDirect(hstmt, BatchStmt, SQL_NTS);
```

When a batch of result-generating statements is executed, it returns one or more row counts or result sets. For information about how to retrieve these, see [“Multiple Results”](#) in Chapter 11, “Retrieving Results (Advanced).”

If a batch of statements includes parameter markers, these are numbered in increasing parameter order as they are in any other statement. For example, the following batch of statements has parameters numbered from 1 to 21; those in the first INSERT statement are numbered 1 to 5 and those in the last INSERT statement are numbered 18 to 21.

```
INSERT INTO Orders (OrderID, CustID, OpenDate, SalesPerson, Status)
    VALUES (?, ?, ?, ?, ?);
INSERT INTO Lines (OrderID, Line, PartID, Quantity) VALUES (?, ?, ?, ?);
INSERT INTO Lines (OrderID, Line, PartID, Quantity) VALUES (?, ?, ?, ?);
INSERT INTO Lines (OrderID, Line, PartID, Quantity) VALUES (?, ?, ?, ?);
INSERT INTO Lines (OrderID, Line, PartID, Quantity) VALUES (?, ?, ?, ?);
```

For more information about parameters, see “Statement Parameters,” later in this chapter.

## Errors and Batches

When an error occurs while executing a batch of SQL statements, one of four things can happen; which one happens is data source – specific and may even depend on the statements included in the batch.

- No statements in the batch are executed.
- No statements in the batch are executed and the transaction is rolled back.
- All of the statements before the error statement are executed.
- All of the statements except the error statement are executed.

In the first two cases, `SQLExecute` and `SQLExecDirect` return `SQL_ERROR`. In the latter two cases, they may return `SQL_SUCCESS_WITH_INFO` or `SQL_SUCCESS`, depending on the implementation. In all

cases, further error information can be retrieved with `SQLGetDiagField`, `SQLGetDiagRec`, or `SQLError`. However, the nature and depth of this information is data source – specific. Furthermore, this information is unlikely to exactly identify the statement in error.

## Executing Catalog Functions

Because a catalog function creates a result set, it is equivalent to executing any result set–generating SQL statement. In fact, catalog functions are often implemented by executing predefined SQL statements or calling predefined procedures that are shipped with the driver or DBMS. Almost anything that applies to SQL statements that create result sets also applies to catalog functions. For example, the `SQL_ATTR_MAX_ROWS` statement attribute limits the number of rows returned by the catalog function, just as it limits the number of rows returned by a `SELECT` statement.

To execute a catalog function, an application just calls the function.

For more information about catalog functions, see “[Chapter 7: Overview of Catalog Functions](#).”

## Statement Parameters

A parameter is a variable in an SQL statement. For example, suppose a `Parts` table has columns named `PartID`, `Description`, and `Price`. To add a part without parameters would require constructing an SQL statement such as:

```
INSERT INTO Parts (PartID, Description, Price) VALUES (2100, 'Drive shaft',
50.00)
```

Although this statement inserts a new order, it is not a good solution for an order entry application because the values to insert cannot be hard-coded in the application. An alternative is to construct the SQL statement at run time, using the values to be inserted. This also is not a good solution, due to the complexity of constructing statements at run time. The best solution is to replace the elements of the `VALUES` clause with question marks (`?`), or parameter markers:

```
INSERT INTO Parts (PartID, Description, Price) VALUES (?, ?, ?)
```

The parameter markers are then bound to application variables. To add a new row, the application has only to set the values of the variables and execute the statement. The driver then retrieves the current values of the variables and sends them to the data source. If the statement will be executed multiple times, the application can make the process even more efficient by preparing the statement.

The statement just shown might be hard-coded in an order entry application to insert a new row. However, parameter markers are not limited to vertical applications. For any application, they ease the difficulty of constructing SQL statements at run time by avoiding conversions to and from text. For example, the part ID just shown is most likely stored in the application as an integer. If the SQL statement is constructed without parameter markers, the application must convert the part ID to text and the data source must convert it back to an integer. By using a parameter marker, the application can send the part ID to the driver as an integer, which usually can send it to the data source as an integer, thereby saving two conversions. For long data values this is critical, because the text forms of such values often exceed the allowable length of an SQL statement.

Parameters are legal only in certain places in SQL statements. For example, they are not allowed in the select list (the list of columns to be returned by a `SELECT` statement), nor are they allowed as both

operands of a binary operator such as the equals sign (=), as it would be impossible to determine the parameter type. In general, parameters are legal only in Data Manipulation Language (DML) statements, and not in Data Definition Language (DDL) statements. For details, see “Parameter Markers” in Appendix C, “SQL Minimum Grammar” of the **SOLID Programmer Guide**.

When the SQL statement invokes a procedure, named parameters can be used. Named parameters are identified by their names, not by their position in the SQL statement. They can be bound by a call to `SQLBindParameter`, but the parameter is identified by the `SQL_DESC_NAME` field of the IPD (implementation parameter descriptor), not by the `ParameterNumber` argument of `SQLBindParameter`. They can also be bound by calling `SQLSetDescField` or `SQLSetDescRec`. For more information on named parameters, see “Binding Parameters by Name (Named Parameters)” later in this chapter. For more information on descriptors, see [“Chapter 13: Overview of Descriptors.”](#)

## Binding Parameters

Each parameter in an SQL statement must be associated, or bound, to a variable in the application before the statement is executed. When the application binds a variable to a parameter, it describes that variable — address, C data type, and so on — to the driver. It also describes the parameter itself — SQL data type, precision, and so on. The driver stores this information in the structure it maintains for that statement and uses the information to retrieve the value from the variable when the statement is executed.

Parameters can be bound or rebound at any time before a statement is executed. If a parameter is rebound after a statement is executed, the binding does not apply until the statement is executed again. To bind a parameter to a different variable, an application simply rebinds the parameter with the new variable; the previous binding is automatically released.

A variable remains bound to a parameter until a different variable is bound to the parameter, all parameters are unbound by calling `SQLFreeStmt` with the `SQL_RESET_PARAMS` option, or the statement is released. For this reason, the application must be sure that variables are not freed until after they are unbound. For more information, see [“Allocating and Freeing Buffers”](#) in Chapter 4, “ODBC Fundamentals.”

Because parameter bindings are just information stored in the structure maintained by the driver for the statement, they can be set in any order. They are also independent of the SQL statement that is executed. For example, suppose an application binds three parameters and then executes the following SQL statement:

```
INSERT INTO Parts (PartID, Description, Price) VALUES (?, ?, ?)
```

If the application then immediately executes this SQL statement:

```
SELECT * FROM Orders WHERE OrderID = ?, OpenDate = ?, Status = ?
```

on the same statement handle, the parameter bindings for the INSERT statement are used because those are the bindings stored in the statement structure. In most cases, this is a poor programming practice and should be avoided. Instead, the application should call `SQLFreeStmt` with the `SQL_RESET_PARAMS` option to unbind all the old parameters and then bind new ones.

## Binding Parameter Markers

The application binds parameters by calling `SQLBindParameter`. `SQLBindParameter` binds one parameter at a time. With it, the application specifies:

- The parameter number. Parameters are numbered in increasing parameter order in the SQL statement, starting with the number 1. While it is legal to specify a parameter number that is higher than there are parameters in the SQL statement, the parameter value will be ignored when the statement is executed.
- The parameter type (input, input/output, or output). Except for parameters in procedure calls, all parameters are input parameters. For more information, see [“Procedure Parameters,”](#) later in this chapter.
- The C data type, address, and byte length of the variable bound to the parameter. The driver must be able to convert the data from the C data type to the SQL data type or an error is returned. For a list of supported conversions, see Appendix D, “Data Types” in the **SOLID Programmer Guide**.
- The SQL data type, precision, and scale of the parameter itself.
- The address of a length/indicator buffer. It provides the byte length of binary or character data, specifies that the data is NULL, or specifies that the data will be sent with SQLPutData. For more information, see [“Using Length/Indicator Values”](#) in Chapter 4, “ODBC Fundamentals.”

For example, the following code binds SalesPerson and CustID to parameters for the SalesPerson and CustID columns. Because SalesPerson contains character data, which is variable length, the code specifies the byte length of SalesPerson (11) and binds SalesPersonLenOrInd to contain the byte length of the data in SalesPerson. This information is not necessary for CustID because it contains integer data, which is of fixed length.

```
SQLCHAR    SalesPerson[11];
SQLINTEGER SalesPersonLenOrInd, CustIDInd;
SQLINTEGER CustID;

// Bind SalesPerson to the parameter for the SalesPerson column and
// CustID to the parameter for the CustID column.
SQLBindParameter(hstmt1, 1, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR, 10, 0,
    SalesPerson, sizeof(SalesPerson), &SalesPersonLenOrInd);
SQLBindParameter(hstmt1, 2, SQL_PARAM_INPUT, SQL_C_ULONG, SQL_INTEGER, 10, 0,
    &CustID, 0, &CustIDInd);

// Set the values of the salesperson and customer ID variables and
length/indicators.
strcpy(SalesPerson, "Garcia");
SalesPersonLenOrInd = SQL_NTS;
CustID = 1331;
CustIDInd = 0;

// Execute a statement to get data for all orders made to the specified
// customer by the specified salesperson.
SQLExecDirect(hstmt1, "SELECT * FROM Orders WHERE SalesPerson=? AND
CustID=?", SQL_NTS);
```

When `SQLBindParameter` is called, the driver stores this information in the structure for the statement. When the statement is executed, it uses the information to retrieve the parameter data and send it to the data source.

Note In ODBC 1.0, parameters were bound with `SQLSetParam`. The Driver Manager maps calls between `SQLSetParam` and `SQLBindParameter`, depending on the versions of ODBC used by the application and driver.

### ***Binding Parameters by Name (Named Parameters)***

Certain DBMSs allow an application to specify the parameters to a stored procedure by name, instead of by position in the procedure call. Such parameters are called named parameters. ODBC supports the use of named parameters. In ODBC, named parameters are used only in calls to stored procedures, and cannot be used in other SQL statements.

The driver checks the value of the `SQL_DESC_UNNAMED` field of the IPD to determine whether named parameters are used. If `SQL_DESC_UNNAMED` is not set to `SQL_UNNAMED`, the driver uses the name in the `SQL_DESC_NAME` field of the IPD to identify the parameter. To bind the parameter, an application can call `SQLBindParameter` to specify the parameter information, and then call `SQLSetDescField` to set the `SQL_DESC_NAME` field of the IPD. When named parameters are used, the order of the parameter in the procedure call is not important, and the parameter's record number is ignored.

The difference between unnamed parameters and named parameters is in the relationship between the record number of the descriptor and the parameter number in the procedure. When unnamed parameters are used, the first parameter marker is related to the first record in the parameter descriptor, which in turn is related to the first parameter (in creation order) in the procedure call. When named parameters are used, the first parameter marker is still related to the first record of the parameter descriptor, but the relationship between the record number of the descriptor and the parameter number in the procedure does not exist anymore. Named parameters do not use the mapping of the descriptor record number to the procedure parameter position; it is replaced by the mapping of the descriptor record name to the procedure parameter name.

Note If automatic population of the IPD is enabled, the driver will populate the descriptor such that the order of the descriptor records will match the order of the parameters in the procedure definition, even if named parameters are used.

If a named parameter is used, all parameters must be named parameters. If any parameter is not a named parameter, then all parameters must not be named parameters. If there were a mixture of named parameters and unnamed parameters, the behavior would be driver-dependent.

As an example of named parameters, suppose a SQL Server stored procedure has been defined as follows:

```
CREATE PROCEDURE test @title_id int = 1, @quote char(30) AS <blah>
```

In this procedure, the first parameter, `@title_id`, has a default value of 1. An application can use the following code to invoke this procedure such that it specifies only one dynamic parameter. This parameter is a named parameter with the name "`@quote`".

```
// prepare the procedure invocation statement.
SQLPrepare(hstmt, "{call test(?)}", SQL_NTS);

// populate record 1 of ipd.
```

```

SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR,
                 30, 0, szQuote, 0, &cbValue);

// get ipd handle and set the SQL_DESC_NAMED and SQL_DESC_UNNAMED fields for
// record #1.
SQLGetStmtAttr(hstmt, SQL_ATTR_IMP_PARAM_DESC, &hIpd, 0, 0);
SQLSetDescField(hIpd, 1, SQL_DESC_NAME, "@quote", SQL_NTS);
SQLSetDescField(hIpd, 1, SQL_DESC_UNNAMED, SQL_NAMED, 0);

// assuming that szQuote has been appropriately initialized,
// execute
SQLExecute(hstmt);

```

### ***Parameter Binding Offsets***

An application can specify that an offset is added to bound parameter buffer addresses and the corresponding length/indicator buffer addresses when `SQLExecDirect` or `SQLExecute` is called. The result of these additions determines the addresses used in these operations.

Bind offsets allow an application to change bindings without calling `SQLBindParameter` for previously bound parameters. A call to `SQLBindParameter` to rebind a parameter changes the buffer address and the length/indicator pointer. Rebinding with an offset, on the other hand, simply adds an offset to the existing bound parameter buffer address and length/indicator buffer address. When offsets are used, the bindings are a “template” of how the application buffers are laid out and the application can move this “template” to different areas of memory by changing the offset. A new offset can be specified at any time, and is always added to the originally bound values.

To specify a bind offset, the application sets the `SQL_ATTR_PARAM_BIND_OFFSET_PTR` statement attribute to the address of an `SQLINTEGER` buffer. Before the application calls a function that uses the bindings, it places an offset in bytes in this buffer, as long as neither the parameter buffer address nor the length/indicator buffer address is 0, and the bound parameter is in the SQL statement. The sum of the address and the offset must be a valid address. (This means that either or both of the offset and the address to which the offset is added, can be invalid, as long as the sum of them is a valid address.)

Note Binding offsets are not supported by ODBC 2.x drivers.

### ***Describing Parameters***

`SQLBindParameter` has arguments that describe the parameter: its SQL type, precision, and scale. The driver uses this information, or metadata, to convert the parameter value to the type needed by the data source. At first glance, it might seem that the driver is in a better position to know the parameter metadata than the application; after all, the driver can easily discover the metadata for a result set column. As it turns out, this is not the case. First, most data sources do not provide a way for the driver to discover parameter metadata. Second, most applications already know the metadata.

If an SQL statement is hard-coded in the application, then the application writer already knows the type of each parameter. If an SQL statement is constructed by the application at run time, the application can determine the metadata as it builds the statement. For example, when the application constructs the clause

```
WHERE OrderID = ?
```

it can call **`SQLColumns`** for the `OrderID` column.

The only situation in which the application cannot easily determine the parameter metadata is when the user enters a parameterized statement. In this case, the application calls `SQLPrepare` to prepare the statement, `SQLNumParams` to determine the number of parameters, and `SQLDescribeParam` to describe each parameter. However, as was noted earlier, most data sources do not provide a way for the driver to discover parameter metadata, so `SQLDescribeParam` is not widely supported.

## Setting Parameter Values

To set the value of a parameter, the application simply sets the value of the variable bound to the parameter. It is not important when this value is set, as long as it is set before the statement is executed. The application can set the value before or after binding the variable and it can change the value as many times as it wants. When the statement is executed, the driver simply retrieves the current value of the variable. This is particularly useful when a prepared statement is executed more than once; the application sets new values for some or all of the variables each time the statement is executed. For an example of this, see [“Prepared Execution,”](#) earlier in this chapter.

If a length/indicator buffer was bound in the call to `SQLBindParameter`, it must be set to one of the following values before the statement is executed:

- The byte length of the data in the bound variable. The driver checks this length only if the variable is character or binary (Value Type is `SQL_C_CHAR` or `SQL_C_BINARY`).
- `SQL_NTS`. The data is a null-terminated string.
- `SQL_NULL_DATA`. The data value is NULL and the driver ignores the value of the bound variable.
- `SQL_DATA_AT_EXEC` or the result of the `SQL_LEN_DATA_AT_EXEC` macro. The value of the parameter is to be sent with `SQLPutData`. For more information, see [“Sending Long Data,”](#) later in this chapter.

The following table shows the values of the bound variable and the length/indicator buffer that the application sets for a variety of parameter values.

| Parameter value | Parameter (SQL) data type  | Variable (C) data type       | Value in bound variable   | Value in length/indicator buffer [d] |
|-----------------|----------------------------|------------------------------|---------------------------|--------------------------------------|
| “ABC”           | <code>SQL_CHAR</code>      | <code>SQL_C_CHAR</code>      | ABC\0 [a]                 | <code>SQL_NTS</code> or 3            |
| 10              | <code>SQL_INTEGER</code>   | <code>SQL_C_SLONG</code>     | 10                        | --                                   |
| 10              | <code>SQL_INTEGER</code>   | <code>SQL_C_CHAR</code>      | 10\0 [a]                  | <code>SQL_NTS</code> or 2            |
| 1 P.M.          | <code>SQL_TYPE_TIME</code> | <code>SQL_C_TYPE_TIME</code> | 13,0,0 [b]                | --                                   |
| 1 P.M.          | <code>SQL_TYPE_TIME</code> | <code>SQL_C_CHAR</code>      | {t '13:00:00'}\0 [a], [c] | <code>SQL_NTS</code> or 14           |
| NULL            | <code>SQL_SMALLINT</code>  | <code>SQL_C_SSHORT</code>    | --                        | <code>SQL_NULL_DATA</code>           |

[a]“\0” represents a null-termination character. The null-termination character is required only if the value in the length/indicator buffer is SQL\_NTS.

[b]The numbers in this list are the numbers stored in the fields of the TIME\_STRUCT structure.

[c]The string uses the ODBC date escape clause. For more information, see “[Date, Time, and Timestamp Literals](#)” in Chapter 8, “SQL Statements.”

[d]Drivers must always check this value to see if it is a special value such as SQL\_NULL\_DATA.

What a driver does with a parameter value at execution time is driver-dependent. If necessary, the driver converts the value from the C data type and byte length of the bound variable to the SQL data type, precision, and scale of the parameter. In most cases, the driver then sends the value to the data source. In some cases, it formats the value as text and inserts it into the SQL statement before sending the statement to the data source.

## **Sending Long Data**

DBMSs define long data as any character or binary data over a certain size, such as 254 characters. It may not be possible to store an entire item of long data in memory, such as when the item represents a long text document or a bitmap. Because such data cannot be stored in a single buffer, the data source sends it to the driver in parts with SQLPutData when the statement is executed. Parameters for which data is sent at execution time are known as data-at-execution parameters.

**Note** An application can actually send any type of data at execution time with SQLPutData, although only character and binary data can be sent in parts. However, if the data is small enough to fit in a single buffer, there is generally no reason to use SQLPutData. It is much easier to bind the buffer and let the driver retrieve the data from the buffer.

To send data at execution time, the application:

1. Passes a 32-bit value that identifies the parameter in the ParameterValuePtr argument in SQLBindParameter rather than the address of a buffer. This value is not analyzed by the driver. It will be returned to the application later, so it should mean something to the application. For example, it might be the number of the parameter or the handle of a file containing data.
2. Passes the address of a length/indicator buffer in the StrLen\_or\_IndPtr argument of SQLBindParameter.
3. Stores SQL\_DATA\_AT\_EXEC or the result of the SQL\_LEN\_DATA\_AT\_EXEC(length) macro in the length/indicator buffer. Both of these values indicate to the driver that the data for the parameter will be sent with SQLPutData. SQL\_LEN\_DATA\_AT\_EXEC(length) is used when sending long data to a data source that needs to know how many bytes of long data will be sent so that it can preallocate space. To determine if a data source requires this value, the application calls SQLGetInfo with the SQL\_NEED\_LONG\_DATA\_LEN option. All drivers must support this macro; if the data source does not require the byte length, the driver can ignore it.



4. Calls `SQLExecute` or `SQLExecDirect`. The driver discovers that a length/indicator buffer contains the value `SQL_DATA_AT_EXEC` or the result of the `SQL_LEN_DATA_AT_EXEC(length)` macro and returns `SQL_NEED_DATA` as the return value of the function.
5. Calls `SQLParamData` in response to the `SQL_NEED_DATA` return value. If long data needs to be sent, `SQLParamData` returns `SQL_NEED_DATA`. In the buffer pointed to by the `ValuePtrPtr` argument, the driver returns the value that identifies the data-at-execution parameter. If there is more than one data-at-execution parameter, the application must use this value to determine which parameter to send data for; the driver is not required to request data for data-at-execution parameters in any particular order.
6. Calls `SQLPutData` to send the parameter data to the driver. If the parameter data does not fit into a single buffer, as is often the case with long data, the application calls `SQLPutData` repeatedly to send the data in parts; it is up to the driver and data source to reassemble the data. If the application passes null-terminated string data, the driver or data source must remove the null-termination character as part of the reassembly process.
7. Calls `SQLParamData` again to indicate that it has sent all of the data for the parameter. If there are any data-at-execution parameters for which data has not been sent, the driver returns `SQL_NEED_DATA` and the value that identifies the next parameter; the application returns to step 6. If data has been sent for all data-at-execution parameters, the statement is executed. `SQLParamData` returns `SQL_SUCCESS` or `SQL_SUCCESS_WITH_INFO`, and can return any return value or diagnostic that `SQLExecute` or `SQLExecDirect` can return.

After `SQLExecute` or `SQLExecDirect` returns `SQL_NEED_DATA` and before data has been completely sent for the last data-at-execution parameter, the statement is in a Need Data state. While a statement is in a Need Data state, the application can call only `SQLPutData`, `SQLParamData`, `SQLCancel`, `SQLGetDiagField`, or `SQLGetDiagRec`; all other functions return `SQLSTATE HY010` (Function sequence error). Calling `SQLCancel` cancels execution of the statement and returns it to its previous state. For more information, see Appendix B, “ODBC State Transition Tables” contained on the Microsoft Web site (ODBC Programmer’s Reference).

For an example of sending data at execution time, see the `SQLPutData` function description.

## Retrieving Output Parameters by `SQLGetData`

An application may retrieve bound output parameters by calling `SQLBindParameter`, in which case output values are present in the application variables to which the respective parameters are bound. If the output parameters are unbound, on the other hand, the application can read argument values by calling `SQLGetData`. The application may use both these techniques if some parameters are bound and some are unbound.

For unbound output parameters following the highest-numbered bound parameter, portable applications obtain the parameter data by calling `SQLGetData` in ascending order of parameter number. It is driver-defined whether an application can obtain parameter data in a different sequence. It is driver-defined whether parameter data for lower-numbered, unbound parameters is available.

The application can achieve type conversion of the parameter data by specifying in the call to `SQLGetData` either the desired target type or the value `SQL_APD_TYPE`, which means that the APD indicates the desired target type even though the parameter is unbound.

If the length of the unbound output parameter value exceeds the length of the application's buffer, `SQLGetData` may be called multiple times to obtain the value of a single parameter of `SQL_CHAR` or `SQL_VARCHAR` type in pieces of manageable size.

## Procedure Parameters

Parameters in procedure calls can be input, input/output, or output parameters. This is different from parameters in all other SQL statements, which are always input parameters.

Input parameters are used to send values to the procedure. For example, suppose the `Parts` table has `PartID`, `Description`, and `Price` columns. The `InsertPart` procedure might have an input parameter for each column in the table. For example:

```
{call InsertPart(?, ?, ?)}
```

A driver should not modify the contents of an input buffer until `SQLExecDirect` or `SQLExecute` returns `SQL_SUCCESS`, `SQL_SUCCESS_WITH_INFO`, `SQL_ERROR`, `SQL_INVALID_HANDLE`, or `SQL_NO_DATA`. The contents of the input buffer should not be modified while `SQLExecDirect` or `SQLExecute` returns `SQL_NEED_DATA` or `SQL_STILL_EXECUTING`.

Input/output parameters are used both to send values to procedures and retrieve values from procedures. Using the same parameter as both an input and an output parameter tends to be confusing and should be avoided. For example, suppose a procedure accepts an order ID and returns the ID of the customer. This can be defined with a single input/output parameter:

```
{call GetCustID(?)}
```

It may be better to use two parameters: an input parameter for the order ID and an output or input/output parameter for the customer ID:

```
{call GetCustID(?, ?)}
```

Output parameters are used to retrieve the procedure return value and to retrieve values from procedure arguments; procedures that return values are sometimes known as functions. For example, suppose the `GetCustID` procedure just mentioned returns a value that indicates whether it was able to find the order. In the following call, the first parameter is an output parameter used to retrieve the procedure return value, the second parameter is an input parameter used to specify the order ID, and the third parameter is an output parameter used to retrieve the customer ID:

```
{? = call GetCustID(?, ?)}
```

Drivers handle values for input and input/output parameters in procedures no differently than input parameters in other SQL statements. When the statement is executed, they retrieve the values of the variables bound to these parameters and send them to the data source.

After the statement has been executed, drivers store the returned values of input/output and output parameters in the variables bound to those parameters. Note that these are not guaranteed to be set until after all results returned by the procedure have been fetched and `SQLMoreResults` has returned

SQL\_NO\_DATA. If executing the statement results in an error, the contents of the input/output parameter buffer or output parameter buffer are undefined.

An application calls `SQLProcedure` to determine if a procedure has a return value. It calls `SQLProcedureColumns` to determine the type (return value, input, input/output, or output) of each procedure parameter.

## Arrays of Parameter Values

It is often useful for applications to pass arrays of parameters. For example, using arrays of parameters and a parameterized **INSERT** statement, an application can insert a number of rows at once. There are several advantages to using arrays. First, network traffic is reduced, because the data for many statements is sent in a single packet (if the data source supports parameter arrays natively). Second, some data sources can execute SQL statements using arrays faster than executing the same number of separate SQL statements. Finally, when the data is stored in an array, as is often the case for screen data, the application can bind all of the rows in a particular column with a single call to **SQLBindParameter** and update them by executing a single statement.

Unfortunately, not many data sources support parameter arrays. However, a driver can emulate parameter arrays by executing an SQL statement once for each set of parameter values. This can lead to increases in speed, because the driver can then prepare the statement that it plans to execute once for each parameter set. It might also lead to simpler application code.

### *Binding Arrays of Parameters*

Applications that use arrays of parameters bind the arrays to the parameters in the SQL statement. There are two binding styles:

- Bind an array to each parameter. Each data structure (array) contains all the data for a single parameter. This is called *column-wise binding* because it binds a column of values for a single parameter.
- Define a structure to hold the parameter data for an entire set of parameters and bind an array of these structures. Each data structure contains the data for a single SQL statement. This is called *row-wise binding* because it binds a row of parameters.

As when the application binds single variables to parameters, it calls **SQLBindParameter** to bind arrays to parameters. The only difference is that the addresses passed are array addresses, not single-variable addresses. The application sets the `SQL_ATTR_PARAM_BIND_TYPE` statement attribute to specify whether it is using column-wise (the default) or row-wise binding. Whether to use column-wise or row-wise binding is largely a matter of application preference. Depending on how the processor accesses memory, row-wise binding might be faster. However, the difference is likely to be negligible except for very large numbers of rows of parameters.

### *Column-Wise Binding*

When using column-wise binding, an application binds one or two arrays to each parameter for which data is to be provided. The first array holds the data values and the second array holds length/indicator buffers. Each array contains as many elements as there are values for the parameter.

Column-wise binding is the default. The application also can change from row-wise binding to column-wise binding by setting the SQL\_ATTR\_PARAM\_BIND\_TYPE statement attribute. The following figure shows how column-wise binding works.

### Column-wise binding of parameters

For example, the following code binds 10-element arrays to parameters for the PartID, Description, and Price columns, and executes a statement to insert 10 rows. It uses column-wise binding.

```
#define DESC_LEN 51
#define ARRAY_SIZE 10

SQLCHAR * Statement = "INSERT INTO Parts (PartID, Description, Price) "
    "VALUES (?, ?, ?)";
SQLINTEGER PartIDArray[ARRAY_SIZE];
SQLCHAR DescArray[ARRAY_SIZE][DESC_LEN];
SQLREAL PriceArray[ARRAY_SIZE];
SQLINTEGER PartIDIndArray[ARRAY_SIZE], DescLenOrIndArray[ARRAY_SIZE],
    PriceIndArray[ARRAY_SIZE];
SQLUSMALLINT i, ParamsProcessed, ParamStatusArray[ARRAY_SIZE];

// Set the SQL_ATTR_PARAM_BIND_TYPE statement attribute to use column-wise
// binding.
SQLSetStmtAttr(hstmt, SQL_ATTR_PARAM_BIND_TYPE, SQL_PARAMETER_BIND_BY_COLUMN,
0);

// Specify the number of elements in each parameter array.
SQLSetStmtAttr(hstmt, SQL_ATTR_PARAMSET_SIZE, ARRAY_SIZE, 0);

// Specify an array in which to return the status of each set of parameters.
SQLSetStmtAttr(hstmt, SQL_ATTR_PARAM_STATUS_PTR, ParamStatusArray, 0);

// Specify an SQLINTEGER value in which to return the number of sets of
// parameters
// processed.
SQLSetStmtAttr(hstmt, SQL_ATTR_PARAMS_PROCESSED_PTR, &ParamsProcessed, 0);

// Bind the parameters in column-wise fashion.
SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_ULONG, SQL_INTEGER, 5, 0,
    PartIDArray, 0, PartIDIndArray);
SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR, DESC_LEN - 1,
0,
    DescArray, DESC_LEN, DescLenOrIndArray);
SQLBindParameter(hstmt, 3, SQL_PARAM_INPUT, SQL_C_FLOAT, SQL_REAL, 7, 0,
    PriceArray, 0, PriceIndArray);

// Set part ID, description, and price.
for (i = 0; i < ARRAY_SIZE; i++) {
    GetNewValues(&PartIDArray[i], DescArray[i], &PriceArray[i]);
    PartIDIndArray[i] = 0;
    DescLenOrIndArray[i] = SQL_NTS;
    PriceIndArray[i] = 0;
}

// Execute the statement.
```

```

SQLExecDirect(hstmt, Statement, SQL_NTS);

// Check to see which sets of parameters were processed successfully.
for (i = 0; i < ParamsProcessed; i++) {
    printf("Parameter Set    Status\n");
    printf("-----\n");
    switch (ParamStatusArray[i]) {
        case SQL_PARAM_SUCCESS:
        case SQL_PARAM_SUCCESS_WITH_INFO:
            printf("%13d    Success\n", i);
            break;

        case SQL_PARAM_ERROR:
            printf("%13d    Error\n", i);
            break;

        case SQL_PARAM_UNUSED:
            printf("%13d    Not processed\n", i);
            break;

        case SQL_PARAM_DIAG_UNAVAILABLE:
            printf("%13d    Unknown\n", i);
            break;
    }
}

```

## ***Row-Wise Binding***

When using row-wise binding, an application defines a structure for each set of parameters. The structure contains one or two elements for each parameter. The first element holds the parameter value and the second element holds the length/indicator buffer. The application then allocates an array of these structures, which contains as many elements as there are values for each parameter.

The application declares the size of the structure to the driver with the `SQL_ATTR_PARAM_BIND_TYPE` statement attribute. The application binds the addresses of the parameters in the first structure of the array. Thus, the driver can calculate the address of the data for a particular row and column as:

$$\text{Address} = \text{Bound Address} + ((\text{Row Number} - 1) * \text{Structure Size}) + \text{Offset}$$

where rows are numbered from 1 to the size of the parameter set, and the offset, if defined, is the value pointed to by the `SQL_ATTR_PARAM_BIND_OFFSET_PTR` statement attribute. The following figure shows how row-wise binding works. The parameters can be placed in the structure in any order, but are shown in sequential order for clarity.

### **Row-wise binding of parameters**

For example, the following code creates a structure with elements for the values to store in the PartID, Description, and Price columns. It then allocates a 10-element array of these structures and binds it to parameters for the PartID, Description, and Price columns, using row-wise binding. It then executes a statement to insert 10 rows.

```

#define DESC_LEN 51
#define ARRAY_SIZE 10

```

```

typedef tagPartStruct {
    SQLREAL    Price;
    SQLUINTEGER PartID;
    SQLCHAR    Desc[DESC_LEN];
    SQLINTEGER PriceInd;
    SQLINTEGER PartIDInd;
    SQLINTEGER DescLenOrInd;
} PartStruct;

PartStruct PartArray[ARRAY_SIZE];
SQLCHAR * Statement = "INSERT INTO Parts (PartID, Description, Price) "
    "VALUES (?, ?, ?)";
SQLUSMALLINT i, ParamsProcessed, ParamStatusArray[ARRAY_SIZE];

// Set the SQL_ATTR_PARAM_BIND_TYPE statement attribute to use column-wise
// binding.
SQLSetStmtAttr(hstmt, SQL_ATTR_PARAM_BIND_TYPE, sizeof(PartStruct), 0);

// Specify the number of elements in each parameter array.
SQLSetStmtAttr(hstmt, SQL_ATTR_PARAMSET_SIZE, ARRAY_SIZE, 0);

// Specify an array in which to return the status of each set of parameters.
SQLSetStmtAttr(hstmt, SQL_ATTR_PARAM_STATUS_PTR, ParamStatusArray, 0);

// Specify an SQLUINTEGER value in which to return the number of sets of
// parameters
// processed.
SQLSetStmtAttr(hstmt, SQL_ATTR_PARAMS_PROCESSED_PTR, &ParamsProcessed, 0);

// Bind the parameters in row-wise fashion.
SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_ULONG, SQL_INTEGER, 5, 0,
    &PartArray[0].PartID, 0, &PartArray[0].PartIDInd);
SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR, DESC_LEN - 1,
    0,
    PartArray[0].Desc, DESC_LEN, &PartArray[0].DescLenOrInd);
SQLBindParameter(hstmt, 3, SQL_PARAM_INPUT, SQL_C_FLOAT, SQL_REAL, 7, 0,
    &PartArray[0].Price, 0, &PartArray[0].PriceInd);

// Set part ID, description, and price.
for (i = 0; i < ARRAY_SIZE; i++) {
    GetNewValues(&PartArray[i].PartID, PartArray[i].Desc, &PartArray[i].Price);
    PartArray[0].PartIDInd = 0;
    PartArray[0].DescLenOrInd = SQL_NTS;
    PartArray[0].PriceInd = 0;
}

// Execute the statement.
SQLExecDirect(hstmt, Statement, SQL_NTS);

// Check to see which sets of parameters were processed successfully.
for (i = 0; i < ParamsProcessed; i++) {
    printf("Parameter Set Status\n");
    printf("-----\n");
}

```

```

switch (ParamStatusArray[i]) {
    case SQL_PARAM_SUCCESS:
    case SQL_PARAM_SUCCESS_WITH_INFO:
        printf("%13d  Success\n", i);
        break;

    case SQL_PARAM_ERROR:
        printf("%13d  Error\n", i);
        break;

    case SQL_PARAM_UNUSED:
        printf("%13d  Not processed\n", i);
        break;

    case SQL_PARAM_DIAG_UNAVAILABLE:
        printf("%13d  Unknown\n", i);
        break;
}

```

### ***Using Arrays of Parameters***

To use arrays of parameters, the application calls **SQLSetStmtAttr** with an *Attribute* argument of `SQL_ATTR_PARAMSET_SIZE` to specify the number of sets of parameters. It calls **SQLSetStmtAttr** with an *Attribute* argument of `SQL_ATTR_PARAMS_PROCESSED_PTR` to specify the address of a variable in which the driver can return the number of sets of parameters processed, including error sets. It calls **SQLSetStmtAttr** with an *Attribute* argument of `SQL_ATTR_PARAM_STATUS_PTR` to point to an array in which to return status information for each row of parameter values. The driver stores these addresses in the structure it maintains for the statement.

**Note** In ODBC 2.x, **SQLParamOptions** was called to specify multiple values for a parameter. In ODBC 3.x, the call to **SQLParamOptions** has been replaced by calls to **SQLSetStmtAttr** to set the `SQL_ATTR_PARAMSET_SIZE` and `SQL_ATTR_PARAMS_PROCESSED_ARRAY` attributes.

Before executing the statement, the application sets the value of each element of each bound array. When the statement is executed, the driver uses the information it stored to retrieve the parameter values and send them to the data source; if possible, the driver should send these values as arrays. Although the use of arrays of parameters is best implemented by executing the SQL statement with all of the parameters in the array with a single call to the data source, this capability is not widely available in DBMSs today. Thus, drivers can simulate it by executing an SQL statement multiple times, each with a single set of parameters.

Before an application uses arrays of parameters, it must be sure that they are supported by the drivers used by the application. There are two ways to do this:

- Use only drivers known to support arrays of parameters. The application can hard-code the names of these drivers or the user can be instructed to use only these drivers. Custom applications and vertical applications commonly use a limited set of drivers.

- Check for support of arrays of parameters at run time. A driver supports arrays of parameters if it is possible to set the `SQL_ATTR_PARAMSET_SIZE` statement attribute to a value greater than 1. Generic applications and vertical applications commonly check for support of arrays of parameters at run time.

The availability of row counts and result sets in parameterized execution can be determined by calling **SQLGetInfo** with the `SQL_PARAM_ARRAY_ROW_COUNTS` and `SQL_PARAM_ARRAY_SELECTS` options. For **INSERT**, **UPDATE**, and **DELETE** statements, the `SQL_PARAM_ARRAY_ROW_COUNTS` option indicates whether individual row counts (one for each parameter set) are available (`SQL_PARC_BATCH`), or row counts are rolled up into one (`SQL_PARC_NO_BATCH`). For **SELECT** statements, the `SQL_PARAM_ARRAY_SELECTS` option indicates whether a result set is available for each set of parameters (`SQL_PAS_BATCH`), or only one result set is available (`SQL_PAS_NO_BATCH`). If the driver does not allow result set – generating statements to be executed with an array of parameters, `SQL_PARAM_ARRAY_SELECTS` returns `SQL_PAS_NO_SELECT`. It is data source – specific whether arrays of parameters can be used with other types of statements, especially because the use of parameters in these statements would be data source – specific, and would not follow ODBC SQL grammar.

The array pointed to by the `SQL_ATTR_PARAM_OPERATION_PTR` statement attribute can be used to ignore rows of parameters. If an element of the array is set to `SQL_PARAM_IGNORE`, the set of parameters corresponding to that element is excluded from the **SQLExecute** or **SQLExecDirect** call. The array pointed to by the `SQL_ATTR_PARAM_OPERATION_PTR` attribute is allocated and filled in by the application, and read by the driver. If fetched rows are used as input parameters, the values of the row status array can be used in the parameter operation array.

## Error Processing

If an error occurs while executing the statement, the execution function returns an error and sets the row number variable to the number of the row containing the error. It is data source – specific whether all rows except the error set are executed, or all rows before (but not after) the error set are executed. Because it processes sets of parameters, the driver sets the buffer specified by the `SQL_ATTR_PARAMS_PROCESSED_PTR` statement attribute to the number of the row currently being processed. If all sets except the error set are executed, the driver sets this buffer to `SQL_ATTR_PARAMSET_SIZE` after all rows are processed.

If the `SQL_ATTR_PARAM_STATUS_PTR` statement attribute has been set, **SQLExecute** or **SQLExecDirect** returns the *parameter status array*, which provides the status of each set of parameters. The parameter status array is allocated by the application and filled in by the driver. Its elements indicate whether the SQL statement was executed successfully for the row of parameters, or whether an error occurred while processing the set of parameters. If an error occurred, the driver sets the corresponding value in the parameter status array to `SQL_PARAM_ERROR`, and returns `SQL_SUCCESS_WITH_INFO`. The application can check the status array to determine which rows were processed. Using the row number, the application can often correct the error and resume processing.

How the parameter status array is used is determined by the `SQL_PARAM_ARRAY_ROW_COUNTS` and `SQL_PARAM_ARRAY_SELECTS` options returned by a call to **SQLGetInfo**. For **INSERT**, **UPDATE**, and **DELETE** statements, the parameter status array is filled in with status information if `SQL_PARC_BATCH` is returned for `SQL_PARAM_ARRAY_ROW_COUNTS`, but not if `SQL_PARC_NO_BATCH` is returned. For **SELECT** statements, the parameter status array is filled in if



SQL\_PAS\_BATCH is returned for SQL\_PARAM\_ARRAY\_SELECT, but not if SQL\_PAS\_NO\_BATCH or SQL\_PAS\_NO\_SELECT is returned.

### ***Data-at-Execution Parameters***

If any of the values in the length/indicator array are SQL\_DATA\_AT\_EXEC or the result of the SQL\_LEN\_DATA\_AT\_EXEC(*length*) macro, the data for those values is sent with **SQLPutData** in the usual way. Two aspects of this process bear special comment, because they are not readily obvious:

- When the driver returns SQL\_NEED\_DATA, it must set the address of the row number variable to the row for which it needs data. As in the single-valued case, the application cannot make any assumptions about the order in which the driver will request parameter values within a single set of parameters. If an error occurs in the execution of a data-at-execution parameter, the buffer specified by the SQL\_ATTR\_PARAMS\_PROCESSED\_PTR statement attribute is set to the number of the row on which the error occurred, the status for the row in the row status array specified by the SQL\_ATTR\_PARAM\_STATUS\_PTR statement attribute is set to SQL\_PARAM\_ERROR, and the call to **SQLExecute**, **SQLExecDirect**, **SQLParamData**, or **SQLPutData** returns SQL\_ERROR. The contents of this buffer are undefined if **SQLExecute**, **SQLExecDirect**, or **SQLParamData** return SQL\_STILL\_EXECUTING.
- Because the driver does not interpret the value in the *ParameterValuePtr* argument of **SQLBindParameter** for data-at-execution parameters, if the application provides a pointer to an array, **SQLParamData** does not extract and return an element of this array to the application. Instead, it returns the scalar value the application had supplied. This means the value returned by **SQLParamData** is not sufficient to specify the parameter for which the application needs to send data; the application also needs to consider the current row number.

When only some of the elements of an array of parameters are data-at-execution parameters, the application must pass the address of an array in *ParameterValuePtr* that contains elements for all the parameters. This array is interpreted normally for the parameters that are not data-at-execution parameters. For the data-at-execution parameters, the value that **SQLParamData** provides to the application, which normally could be used to identify the data that the driver is requesting on this occasion, is always the address of the array.

## **Asynchronous Execution**

By default, drivers execute ODBC functions synchronously; that is, the application calls a function and the driver does not return control to the application until it has finished executing the function. However, some functions can be executed asynchronously; that is, the application calls the function, and the driver, after minimal processing, returns control to the application. The application can then call other functions while the first function is still executing.

Asynchronous execution is supported for most functions that are largely executed on the data source, such as the functions to prepare and execute SQL statements, retrieve metadata, and fetch data. It is most useful when the task being executed on the data source takes a long time, such as a complex query against a large database.

**Note** In general, applications should execute functions asynchronously only on single-threaded operating systems. On multithread operating systems, applications should execute functions on

separate threads, rather than executing them asynchronously on the same thread. No functionality is lost if drivers that operate only on multithread operating systems do not support asynchronous execution.

Asynchronous execution is controlled on either a per-statement or per-connection basis, depending on the data source. That is, the application specifies not that a particular function is to be executed asynchronously, but that any function executed on a particular statement or a particular connection is to be executed asynchronously. To find out which one is supported, an application calls **SQLGetInfo** with an option of `SQL_ASYNC_MODE`. `SQL_AM_CONNECTION` is returned if connection-level asynchronous execution is supported; `SQL_AM_STATEMENT` if statement-level asynchronous execution is supported.

To specify that functions executed with a particular statement are to be executed asynchronously, the application calls **SQLSetStmtAttr** with the `SQL_ATTR_ASYNC_ENABLE` attribute and sets it to `SQL_ASYNC_ENABLE_ON`. If connection-level asynchronous processing is supported, the `SQL_ATTR_ASYNC_ENABLE` statement attribute is read-only, and its value is the same as the connection attribute of the connection on which the statement was allocated. It is driver-specific whether the value of the statement attribute is set at statement allocation time or later. Attempting to set it will return `SQL_ERROR` and `SQLSTATE HYC00` (Optional feature not implemented).

To determine the maximum number of active concurrent statements in asynchronous mode that the driver can support on a given connection, the application calls **SQLGetInfo** with the `SQL_MAX_ASYNC_CONCURRENT_STATEMENTS` option.

To specify that functions executed with a particular connection are to be executed asynchronously, the application calls **SQLSetConnectAttr** with the `SQL_ATTR_ASYNC_ENABLE` attribute and sets it to `SQL_ASYNC_ENABLE_ON`. All future statement handles allocated on the connection will be enabled for asynchronous execution; it is driver-defined whether existing statement handles will be enabled by this action. If `SQL_ATTR_ASYNC_ENABLE` is set to `SQL_ASYNC_ENABLE_OFF`, all statements on the connection are in synchronous mode. Note that an error is returned if asynchronous execution is enabled while there is an active statement on the connection.

When the application executes a function with a statement that is enabled for asynchronous processing, the driver performs a minimal amount of processing (such as checking arguments for errors), hands processing to the data source, and returns control to the application with the `SQL_STILL_EXECUTING` return code. The application then performs other tasks. To determine when the asynchronous function has finished, the application polls the driver at regular intervals by calling the function with the same arguments as it originally used. If the function is still executing, it returns `SQL_STILL_EXECUTING`; if it has finished executing, it returns the code it would have returned had it executed synchronously, such as `SQL_SUCCESS`, `SQL_ERROR`, or `SQL_NEED_DATA`. For example:

```
SQLHSTMT hstmt1;
SQLRETURN rc;

// Specify that the statement is to be executed asynchronously.
SQLSetStmtAttr(hstmt1, SQL_ATTR_ASYNC_ENABLE, SQL_ASYNC_ENABLE_ON, 0);

// Execute a SELECT statement asynchronously.
while ((rc=SQLExecDirect(hstmt1,"SELECT * FROM
Orders",SQL_NTS))==SQL_STILL_EXECUTING) {
    // While the statement is still executing, do something else.
```

```
// Do not use hstmt1, because it is being used asynchronously.
}
```

```
// When the statement has finished executing, retrieve the results.
```

Whether the driver executes the function synchronously or asynchronously is up to the driver. For example, suppose the result set metadata is cached in the driver. In this case, it takes very little time to execute **SQLDescribeCol** and the driver should simply execute the function, rather than artificially delaying execution. On the other hand, if the driver needs to retrieve the metadata from the data source, it should return control to the application while it is doing this. Thus, the application must be able to handle a return code other than **SQL\_STILL\_EXECUTING** when it first executes a function asynchronously.

While the function is being executed asynchronously, the application can call functions on any other statements. It can also call functions on any connection except the one associated with the asynchronous statement. On the asynchronous statement, the application can call only the asynchronously executing function, or **SQLCancel**, **SQLGetDiagField**, or **SQLGetDiagRec**. **SQLGetDiagField** or **SQLGetDiagRec** can be called on an asynchronously executing statement handle to return a header diagnostic field but not a record diagnostic field. On the connection associated with the asynchronous statement, the application can call only **SQLAllocHandle** (to allocate a statement handle), **SQLGetDiagField**, **SQLGetDiagRec**, or **SQLGetFunctions**. If the application calls any other function with the asynchronous statement or with the connection associated with that statement, the function returns **SQLSTATE HY010** (Function sequence error). The application can call any function using handles other than the original statement handle or the original connection handle. For example:

```
SQLHDBC    hdbc1, hdbc2;
SQLHSTMT   hstmt1, hstmt2, hstmt3;
SQLCHAR    *SQLStatement = "SELECT * FROM Orders";
SQLINTEGER InfoValue;
SQLRETURN rc;

SQLAllocHandle(SQL_HANDLE_STMT, hdbc1, &hstmt1);
SQLAllocHandle(SQL_HANDLE_STMT, hdbc1, &hstmt2);
SQLAllocHandle(SQL_HANDLE_STMT, hdbc2, &hstmt3);

// Specify that hstmt1 is to be executed asynchronously.
SQLSetStmtAttr(hstmt1, SQL_ATTR_ASYNC_ENABLE, SQL_ASYNC_ENABLE_ON, 0);

// Execute hstmt1 asynchronously.
while ((rc = SQLExecDirect(hstmt1, SQLStatement, SQL_NTS)) ==
SQL_STILL_EXECUTING) {
    // The following calls return HY010 because the previous call to SQLExecDirect
    is
    // still executing asynchronously on hstmt1. The first call uses hstmt1 and the
    // second call uses hdbc1, on which hstmt1 is allocated.
    SQLExecDirect(hstmt1, SQLStatement, SQL_NTS);           // Error!
    SQLGetInfo(hdbc1, SQL_UNION, (SQLPOINTER) &InfoValue, 0, NULL); // Error!

    // The following calls do not return errors. They use a statement handle other
    than
    // hstmt1 or a connection handle other than hdbc1.
    SQLExecDirect(hstmt2, SQLStatement, SQL_NTS);           // OK
    SQLTables(hstmt3, NULL, 0, NULL, 0, NULL, 0, NULL, 0);  // OK
    SQLGetInfo(hdbc2, SQL_UNION, (SQLPOINTER) &InfoValue, 0, NULL); // OK
}
```

```
}
```

When an application calls a function to determine if it is still executing asynchronously, it must use the original statement handle. This is because asynchronous execution is tracked on a per-statement basis. The application must also supply valid values for the other arguments—the original arguments will do—to get past error checking in the Driver Manager. However, after the driver checks the statement handle and determines that the statement is executing asynchronously, it ignores all other arguments.

While a function is executing asynchronously—that is, after it has returned `SQL_STILL_EXECUTING` and before it returns a different code—the application can cancel it by calling **SQLCancel** with the same statement handle. This is not guaranteed to cancel function execution. For example, the function might have already finished. Furthermore, the code returned by **SQLCancel** indicates whether **SQLCancel** successfully attempted to cancel the function, not whether it actually did cancel the function. To determine if the function was canceled, the application calls the function again. If the function was canceled, it returns `SQL_ERROR` and `SQLSTATE HY008` (Operation canceled). If the function was not canceled, it returns another code, such as `SQL_SUCCESS`, `SQL_STILL_EXECUTING`, or `SQL_ERROR` with a different `SQLSTATE`.

To disable asynchronous execution of a particular statement when the driver supports statement-level asynchronous processing, the application calls **SQLSetStmtAttr** with the `SQL_ATTR_ASYNC_ENABLE` attribute and sets it to `SQL_ASYNC_ENABLE_OFF`. If the driver supports connection-level asynchronous processing, the application calls **SQLSetConnectAttr** to set `SQL_ATTR_ASYNC_ENABLE` to `SQL_ASYNC_ENABLE_OFF`, which disables asynchronous execution of all statements on the connection.

When **SQLGetDiagField** is called on a statement handle on which a function is currently executing asynchronously, the following values are returned:

- What the `SQL_DIAG_CURSOR_ROW_COUNT`, `SQL_DIAG_DYNAMIC_FUNCTION`, `SQL_DIAG_DYNAMIC_FUNCTION_CODE`, and `SQL_DIAG_ROW_COUNT` header fields return are undefined.
- The `SQL_DIAG_NUMBER` header field returns 0.
- The `SQL_DIAG_RETURNCODE` header field returns `SQL_STILL_EXECUTING`.
- All record fields return `SQL_NO_DATA`.

**SQLGetDiagRec** always returns `SQL_NO_DATA` when it is called on a statement handle on which a function is currently executing asynchronously.

For a list of functions that can be executed asynchronously, see the `SQL_ATTR_ASYNC_ENABLE` attribute in the **SQLSetStmtAttr** function description.

## Freeing a Statement Handle

As mentioned earlier, it is more efficient to reuse statements than drop them and allocate new ones. Before executing a new SQL statement on a statement, applications should be sure that the current statement

settings are appropriate. These include statement attributes, parameter bindings, and result set bindings. Generally, parameters and result sets for the old SQL statement need to be unbound (by calling `SQLFreeStmt` with the `SQL_RESET_PARAMS` and `SQL_UNBIND` options) and rebound for the new SQL statement.

When the application has finished using the statement, it calls `SQLFreeHandle` to free the statement. After freeing the statement, it is an application programming error to use the statement's handle in a call to an ODBC function; doing so has undefined but probably fatal consequences.

When `SQLFreeHandle` is called, the driver releases the structure used to store information about the statement. Note that `SQLDisconnect` automatically frees all statements on a connection.

## Chapter 10: Overview of Retrieving Results (Basic)

A *result set* is a set of rows on the data source that matches certain criteria. It is a conceptual table that results from a query and that is available to an application in tabular form. **SELECT** statements, catalog functions, and some procedures create result sets. For example, the first SQL statement creates a result set containing all the rows and all the columns in the Orders table, and the second SQL statement creates a result set containing OrderID, SalesPerson, and Status columns for the rows in the Orders table in which the Status is OPEN.

```
SELECT * FROM Orders
SELECT OrderID, SalesPerson, Status FROM Orders WHERE Status = 'OPEN'
```

A result set can be empty, which is different from no result set at all. For example, the following SQL statement creates an empty result set:

```
SELECT * FROM Orders WHERE 1 = 2
```

An empty result set is no different from any other result set except that it has no rows. For example, the application can retrieve metadata for the result set, can attempt to fetch rows, and must close the cursor over the result set.

The process of retrieving rows from the data source and returning them to the application is called *fetching*. This chapter explains the basic parts of that process. For information about more advanced topics, such as block and scrollable cursors, see “[Block Cursors](#)” and “[Scrollable Cursor Types](#)” in Chapter 11, “Retrieving Results (Advanced).” For information about updating, deleting, and inserting rows, “[Chapter 12: Overview of Updating Data](#).”

### Was a Result Set Created?

In most situations, application programmers know whether the statements their application executes will create a result set. This is the case if the application uses hard-coded SQL statements written by the programmer. It is usually the case when the application constructs SQL statements at run time: The programmer can easily include code that flags whether a **SELECT** statement or an **INSERT** statement is being constructed. In a few situations, the programmer cannot possibly know whether a statement will create a result set. This is true if the application provides a way for the user to enter and execute an SQL statement. It is also true when the application constructs a statement at run time to execute a procedure.

In such cases, the application calls **SQLNumResultCols** to determine the number of columns in the result set. If this is 0, the statement did not create a result set; if it is any other number, the statement did create a result set.

The application can call **SQLNumResultCols** at any time after the statement is prepared or executed. However, because some data sources cannot easily describe the result sets that will be created by prepared statements, performance will suffer if **SQLNumResultCols** is called after a statement is prepared, but before it is executed.

Some data sources also support determining the number of rows that an SQL statement returns in a result set. To do so, the application calls **SQLRowCount**. Exactly what the row count represents is indicated by the setting of the SQL\_DYNAMIC\_CURSOR\_ATTRIBUTES2, SQL\_FORWARD\_ONLY\_CURSOR\_ATTRIBUTES2, SQL\_KEYSET\_CURSOR\_ATTRIBUTES2, or

`SQL_STATIC_CURSOR_ATTRIBUTES2` option (depending on the type of the cursor) returned by a call to **SQLGetInfo**. This bitmask indicates for each cursor type whether the row count returned is exact or approximate, or is not available at all. Whether row counts for static or keyset-driven cursors are affected by changes made through **SQLBulkOperations** or **SQLSetPos**, or by positioned update or delete statements, depends upon other bits returned by the same option arguments listed previously. For more information, see the **SQLGetInfo** function description in the Part II PDF file, “ODBC API Reference” available on the Solid Web site.

## Result Set Metadata

*Metadata* is data that describes other data. For example, result set metadata describes the result set, such as the number of columns in the result set, the data types of those columns, their names, precision, nullability, and so on.

Interoperable applications should always check the metadata of result set columns. The metadata for a column in a result set might differ from the metadata for the column as returned by a catalog function. For example, suppose that an updatable column is included in a result set created by joining two tables. While **SQLColumnPrivileges** might indicate that a user can update the column, the result set metadata might not if the column is on the “many” side of the join; many data sources can update columns on the “one” side of a join but not on the “many” side. Even data types cannot be assumed to be the same, because the data source might promote the data type while creating the result set.

## How is Metadata Used?

Applications require metadata for most result set operations. For example, the application uses the data type of a column to determine what kind of variable to bind to that column. It uses the byte length of a character column to determine how much space it needs to display data from that column. How an application determines the metadata for a column depends on the type of the application.

Vertical applications work with predefined tables and perform predefined operations on those tables. Because the result set metadata for such applications is defined before the application is even written and is controlled by the application developer, it can be hard-coded into the application. For example, if an order ID column is defined as a 4-byte integer in the data source, the application can always bind a 4-byte integer to that column. When metadata is hard-coded in the application, a change to the tables used by the application generally implies a change to the application code. This is rarely a problem, because such changes are generally made as part of a new release of the application.

Like vertical applications, custom applications generally work with predefined tables and perform predefined operations on those tables. For example, an application might be written to transfer data among three different data sources; the data to be transferred is generally known when the application is written. Thus, custom applications also tend to have hard-coded metadata.

Generic applications, especially those that support ad-hoc queries, almost never know the metadata of the result sets they create. Therefore, they must discover the metadata at run time using the functions **SQLNumResultCols**, **SQLDescribeCol**, and **SQLColAttribute**, which are described in the next section, “**SQLDescribeCol** and **SQLColAttribute**.”

All applications, regardless of their type, can hard-code metadata for the result sets returned by the catalog functions. These result sets are defined in the reference section of this manual.

## SQLDescribeCol and SQLColAttribute

**SQLDescribeCol** and **SQLColAttribute** are used to retrieve result set metadata. The difference between these two functions is that **SQLDescribeCol** always returns the same five pieces of information (a column's name, data type, precision, scale, and nullability), while **SQLColAttribute** returns a single piece of information requested by the application. However, **SQLColAttribute** can return a much richer selection of metadata, including a column's case sensitivity, display size, updatability, and searchability.

Many applications, especially ones that only display data, only require the metadata returned by **SQLDescribeCol**. For these applications, it is faster to use **SQLDescribeCol** than **SQLColAttribute** because the information is returned in a single call. Other applications, especially ones that update data, require the additional metadata returned by **SQLColAttribute** and so use both functions. In addition, **SQLColAttribute** supports driver-specific metadata; for more information, see “[Driver-Specific Data Types, Descriptor Types, Information Types, Diagnostic Types, and Attributes](#)” in Chapter 17, “Programming Considerations.”

An application can retrieve result set metadata at any time after a statement has been prepared or executed, and before the cursor over the result set is closed. Very few applications require result set metadata after the statement is prepared and before it is executed. If possible, applications should wait to retrieve metadata until after the statement is executed. The reason is that some data sources cannot return metadata for prepared statements, and emulating this capability in the driver is often a slow process. For example, the driver might generate a zero row result set by replacing the **WHERE** clause of a **SELECT** statement with the clause **WHERE 1 = 2** and executing the resulting statement.

Metadata is often expensive to retrieve from the data source. Because of this, drivers should cache any metadata they retrieve from the server and hold it for as long as the cursor over the result set is open. Also, applications should request only the metadata they absolutely need.

## Binding Columns

Data fetched from the data source is returned to the application in variables that the application has allocated for this purpose. Before this can be done, the application must associate, or *bind*, these variables to the columns of the result set; conceptually, this process is the same as binding application variables to statement parameters. When the application binds a variable to a result set column, it describes that variable — address, data type, and so on — to the driver. The driver stores this information in the structure it maintains for that statement and uses the information to return the value from the column when the row is fetched.

## Binding Result Set Columns

Applications can bind as many or as few columns of the result set as they choose, including binding no columns at all. When a row of data is fetched, the driver returns the data for the bound columns to the application. Whether the application binds all of the columns in the result set depends on the application. For example, applications that generate reports usually have a fixed format; such applications create a result set containing all of the columns used in the report, then bind and retrieve the data for all of these columns. Applications that display screens full of data sometimes allow the user to decide which columns to display; such applications create a result set containing all columns the user might want, but bind and retrieve the data only for those columns chosen by the user.



Data can be retrieved from unbound columns by calling **SQLGetData**. This is commonly called to retrieve long data, which often exceeds the length of a single buffer and must be retrieved in parts.

Columns can be bound at any time, even after rows have been fetched. However, the new bindings do not take effect until the next time a row is fetched; they are not applied to data from rows already fetched.

A variable remains bound to a column until a different variable is bound to the column; the column is unbound by calling **SQLBindCol** with a null pointer as the variable's address; all columns are unbound by calling **SQLFreeStmt** with the `SQL_UNBIND` option; or the statement is released. For this reason, the application must be sure that all bound variables remain valid as long as they are bound. For more information, see "[Allocating and Freeing Buffers](#)" in Chapter 4, "ODBC Fundamentals."

Because column bindings are just information associated with the statement structure, they can be set in any order. They are also independent of the result set. For example, suppose an application binds the columns of the result set generated by the following SQL statement:

```
SELECT * FROM Orders
```

If the application then executes the SQL statement:

```
SELECT * FROM Lines
```

on the same statement handle, the column bindings for the first result set are still in effect because those are the bindings stored in the statement structure. In most cases, this is a poor programming practice and should be avoided. Instead, the application should call **SQLFreeStmt** with the `SQL_UNBIND` option to unbind all the old columns and then bind new ones.

## Using SQLBindCol

The application binds columns by calling **SQLBindCol**. This function binds one column at a time. With it, the application specifies:

The column number. Column 0 is the bookmark column; this column is not included in some result sets. All other columns are numbered starting with the number 1. It is an error to bind a higher numbered column than there are columns in the result set; this error cannot be detected until the result set has been created, so it is returned by **SQLFetch**, not **SQLBindCol**.

The C data type, address, and byte length of the variable bound to the column. It is an error to specify a C data type to which the SQL data type of the column cannot be converted; this error might not be detected until the result set has been created, so it is returned by **SQLFetch**, not **SQLBindCol**. For a list of supported conversions, see Appendix D, "Data Types" in the **SOLID Programmer Guide**. For information about the byte length, see "[Data Buffer Length](#)" in Chapter 4, "ODBC Fundamentals."

The address of a length/indicator buffer. The length/indicator buffer is optional. It is used to return the byte length of binary or character data or return `SQL_NULL_DATA` if the data is NULL. For more information, see "[Using Length/Indicator Values](#)" in Chapter 4, "ODBC Fundamentals."

When **SQLBindCol** is called, the driver associates this information with the statement. When each row of data is fetched, it uses the information to place the data for each column in the bound application variables.

For example, the following code binds variables to the `SalesPerson` and `CustID` columns. Data for the columns will be returned in *SalesPerson* and *CustID*. Because *SalesPerson* is a character buffer, the

application specifies its byte length (11) so the driver can determine whether to truncate the data. The byte length of the returned title, or whether it is NULL, will be returned in *SalesPersonLenOrInd*.

Because *CustID* is an integer variable and has fixed length, there is no need to specify its byte length; the driver assumes it is **sizeof(SQLINTEGER)**. The byte length of the returned customer ID data, or whether it is NULL, will be returned in *CustIDInd*. Note that the application is only interested in whether the salary is NULL, because the byte length is always **sizeof(SQLINTEGER)**.

```
SQLCHAR    SalesPerson[11];
SQLINTEGER CustID;
SQLINTEGER SalesPersonLenOrInd, CustIDInd;
SQLRETURN rc;
SQLHSTMT   hstmt;

// Bind SalesPerson to the SalesPerson column and CustID to the CustID column.
SQLBindCol(hstmt, 1, SQL_C_CHAR, SalesPerson, sizeof(SalesPerson),
           &SalesPersonLenOrInd);
SQLBindCol(hstmt, 2, SQL_C_ULONG, &CustID, 0, &CustIDInd);

// Execute a statement to get the sales person/customer of all orders.
SQLExecDirect(hstmt, "SELECT SalesPerson, CustID FROM Orders ORDER BY
SalesPerson",
              SQL_NTS);

// Fetch and print the data.  Print "NULL" if the data is NULL.  Code to check
if rc
// equals SQL_ERROR or SQL_SUCCESS_WITH_INFO not shown.
while ((rc = SQLFetch(hstmt)) != SQL_NO_DATA) {
    if (SalesPersonLenOrInd == SQL_NULL_DATA)
        printf("NULL    ");
    else
        printf("%10s ", SalesPerson);
    if (CustIDInd == SQL_NULL_DATA)
        printf("NULL\n");
    else
        printf("%d\n", CustID);
}

// Close the cursor.
SQLCloseCursor(hstmt);
```

The following code executes a **SELECT** statement entered by the user and prints each row of data in the result set. Because the application cannot predict the shape of the result set created by the **SELECT** statement, it cannot bind hard-coded variables to the result set as in the previous example. Instead, the application allocates a buffer that holds the data and a length/indicator buffer for each column in that row. For each column, it calculates the offset to the start of the memory for the column and adjusts this offset so that the data and length/indicator buffers for the column start on alignment boundaries. It then binds the memory starting at the offset to the column. From the driver's point of view, the address of this memory is indistinguishable from the address of a variable bound in the previous example. For more information about alignment, see "[Alignment](#)" in Chapter 17, "Programming Considerations."

```
// This application allocates a buffer at run time. For each column, this buffer
// contains memory for the column's data and length/indicator. For example:
//
```

```

// column 1      column 2 column 3 column 4
// <-----><-----><-----><----->
// db1 li1  db2   li2 db3 li3  db4 li4
// |   |   |   |   |   |   |
// ____V____V____V____V____V____V____V____V____
// |_____|_|_____|_|_____|_|_____|_|_____|_|
//
// dbn = data buffer for column n
// lin = length/indicator buffer for column n

// Define a macro to increase the size of a buffer so it is a multiple of the
// alignment
// size. Thus, if a buffer starts on an alignment boundary, it will end just
// before the
// next alignment boundary. In this example, an alignment size of 4 is used
// because
// this is the size of the largest data type used in the application's buffer --
// the
// size of an SDWORD and of the largest default C data type are both 4. If a
// larger
// data type (such as _int64) was used, it would be necessary to align for that
// size.
#define ALIGNSIZE 4
#define ALIGNBUF(Length) Length % ALIGNSIZE ? \
    Length + ALIGNSIZE - (Length % ALIGNSIZE) : Length

SQLCHAR    SelectStmt[100];
SQLSMALLINT NumCols, *CTypeArray, i;
SQLINTEGER *ColLenArray, *OffsetArray, SQLType, *DataPtr;
SQLRETURN rc;
SQLHSTMT    hstmt;

// Get a SELECT statement from the user and execute it.
GetSelectStmt(SelectStmt, 100);
SQLExecDirect(hstmt, SelectStmt, SQL_NTS);

// Determine the number of result set columns. Allocate arrays to hold the C
// type,
// byte length, and buffer offset to the data.
SQLNumResultCols(hstmt, &NumCols);
CTypeArray = (SQLSMALLINT *) malloc(NumCols * sizeof(SQLSMALLINT));
ColLenArray = (SQLINTEGER *) malloc(NumCols * sizeof(SQLINTEGER));
OffsetArray = (SQLINTEGER *) malloc(NumCols * sizeof(SQLINTEGER));

OffsetArray[0] = 0;
for (i = 0; i < NumCols; i++){
    // Determine the column's SQL type. GetDefaultCType contains a switch statement
    // that
    // returns the default C type for each SQL type.
    SQLColAttribute(hstmt, ((SQLUSMALLINT) i) + 1, SQL_DESC_TYPE, NULL, 0, NULL,
    (SQLPOINTER) &SQLType);
    CTypeArray[i] = GetDefaultCType(SQLType);
}

```

```

    // Determine the column's byte length. Calculate the offset in the buffer to
    the
    // data as the offset to the previous column, plus the byte length of the
    previous
    // column, plus the byte length of the previous column's length/indicator
    buffer.
    // Note that the byte length of the column and the length/indicator buffer are
    // increased so that, assuming they start on an alignment boundary, they will
    end on
    // the byte before the next alignment boundary. Although this might leave some
    holes
    // in the buffer, it is a relatively inexpensive way to guarantee alignment.
    SQLColAttribute(hstmt, ((SQLUSMALLINT) i)+1, SQL_DESC_OCTET_LENGTH, NULL, 0,
    NULL, &ColLenArray[i]);
    ColLenArray[i] = ALIGNBUF(ColLenArray[i]);
    if (i)
        OffsetArray[i] = OffsetArray[i-1]+ColLenArray[i-
    1]+ALIGNBUF(sizeof(SQLINTEGER));
}

// Allocate the data buffer. The size of the buffer is equal to the offset to
the data
// buffer for the final column, plus the byte length of the data buffer and
// length/indicator buffer for the last column.
void *DataPtr = malloc(OffsetArray[NumCols - 1] +
    ColLenArray[NumCols - 1] + ALIGNBUF(sizeof(SQLINTEGER)));

// For each column, bind the address in the buffer at the start of the memory
allocated
// for that column's data and the address at the start of the memory allocated
for that
// column's length/indicator buffer.
for (i = 0; i < NumCols; i++)
    SQLBindCol(hstmt,
        ((SQLUSMALLINT) i) + 1,
        CTypeArray[i],
        (SQLPOINTER)((SQLCHAR *)DataPtr + OffsetArray[i]),
        ColLenArray[i],
        (SQLINTEGER *)((SQLCHAR *)DataPtr + OffsetArray[i] + ColLenArray[i]));

// Retrieve and print each row. PrintData accepts a pointer to the data, its C
type,
// and its byte length/indicator. It contains a switch statement that casts and
prints
// the data according to its type. Code to check if rc equals SQL_ERROR or
// SQL_SUCCESS_WITH_INFO not shown.
while ((rc = SQLFetch(hstmt)) != SQL_NO_DATA) {
    for (i = 0; i < NumCols; i++) {
        PrintData((SQLCHAR *)DataPtr[OffsetArray[i]], CTypeArray[i],
            (SQLINTEGER *)((SQLCHAR *)DataPtr[OffsetArray[i] + ColLenArray[i]]));
    }
}

// Close the cursor.

```

```
SQLCloseCursor(hstmt);
```

## Fetching Data

The process of retrieving rows from the result set and returning them to the application is called *fetching*. This section describes how to fetch data.

## Cursors

An application fetches data with a *cursor*. A cursor is different from a result set: A result set is the set of rows that matches a particular search criteria, while a cursor is the software that returns those rows to the application. The name “cursor” as it applies to databases probably originated from the blinking cursor on a computer terminal. Just as that cursor indicates the current position on the screen and where the typed words will appear next, a cursor on a result set indicates the current position in the result set and what row will be returned next.

The cursor model in ODBC is based on the cursor model in embedded SQL. One notable difference between these models is the way cursors are opened. In embedded SQL, a cursor must be explicitly declared and opened before it can be used. In ODBC, a cursor is implicitly opened when a statement that creates a result set is executed. When the cursor is opened, it is positioned before the first row of the result set. In both embedded SQL and ODBC, a cursor must be closed after the application has finished using it.

Different cursors have different characteristics. The most common type of cursor, which is called a *forward-only cursor*, can only move forward through the result set. To return to a previous row, the application must close and reopen the cursor, then read rows from the beginning of the result set until it reaches the required row. Forward-only cursors provide a fast mechanism for making a single pass through a result set.

Forward-only cursors are less useful for screen-based applications, in which the user scrolls backward and forward through the data. Such applications can use a forward-only cursor by reading the result set once, caching the data locally, and performing scrolling themselves. However, this works well only with small amounts of data. A better solution is to use a *scrollable cursor*, which provides random access to the result set. Such applications can also increase performance by fetching more than one row of data at a time, using what is called a *block cursor*. For more information on block cursors, see “[Using Block Cursors](#),” in Chapter 11, “Retrieving Results (Advanced).”

The forward-only cursor is the default cursor type in ODBC and is discussed in the following sections. For more information about block cursors and scrollable cursors, see “[Block Cursors](#)” and “[Scrollable Cursors](#)” in Chapter 11, “Retrieving Results (Advanced).”

**Important** Committing or rolling back a transaction, either by explicitly calling **SQLEndTran** or by operating in autocommit mode, causes some data sources to close all the cursors on all statements on a connection. For more information, see the **SQL\_CURSOR\_COMMIT\_BEHAVIOR** and **SQL\_CURSOR\_ROLLBACK\_BEHAVIOR** attributes in the **SQLGetInfo** function description in the Part II PDF file, “ODBC API Reference” available on the Solid Web site.

## Fetching a Row of Data

To fetch a row of data, an application calls `SQLFetch`. `SQLFetch` can be called with any kind of cursor, but it only moves the rowset cursor in a forward-only direction. `SQLFetch` advances the cursor to the next row and returns the data for any columns that were bound with calls to `SQLBindCol`. When the cursor reaches the end of the result set, `SQLFetch` returns `SQL_NO_DATA`. For examples of calling `SQLFetch`, see [“Using SQLBindCol,”](#) earlier in this chapter.

Exactly how `SQLFetch` is implemented is driver-specific, but the general pattern is for the driver to retrieve the data for any bound columns from the data source, convert it according to the types of the bound variables, and place the converted data in those variables. If the driver cannot convert any data, `SQLFetch` returns an error. The application can continue fetching rows, but the data for the current row is lost. What happens to the data for unbound columns depends on the driver, but most drivers either retrieve and discard it or never retrieve it at all.

The driver also sets the values of any length/indicator buffers that have been bound. If the data value for a column is `NULL`, the driver sets the corresponding length/indicator buffer to `SQL_NULL_DATA`. If the data value is not `NULL`, the driver sets the length/indicator buffer to the byte length of the data after conversion. If this length cannot be determined, as is sometimes the case with long data that is retrieved by more than one function call, the driver sets the length/indicator buffer to `SQL_NO_TOTAL`. Note that for fixed-length data types, such as integers and date structures, the byte length is the size of the data type.

For variable-length data, such as character and binary data, the driver checks the byte length of the converted data against the byte length of the buffer bound to the column; the buffer’s length is specified in the `BufferLength` argument in `SQLBindCol`. If the byte length of the converted data is greater than the byte length of the buffer, the driver truncates the data to fit in the buffer, returns the untruncated length in the length/indicator buffer, returns `SQL_SUCCESS_WITH_INFO`, and places `SQLSTATE 01004` (Data truncated) in the diagnostics. The only exception to this is if a variable-length bookmark is truncated when returned by `SQLFetch`, which returns `SQLSTATE 22001` (String data, right truncated).

Fixed-length data is never truncated, because the driver assumes that the size of the bound buffer is the size of the data type. Data truncation tends to be rare, because the application usually binds a buffer large enough to hold the entire data value; it determines the necessary size from the metadata. However, the application might explicitly bind a buffer it knows to be too small. For example, it might retrieve and display the first 20 characters of a part description or the first 100 characters of a long text column.

Character data must be null-terminated by the driver before it is returned to the application, even if it has been truncated. The null-termination character is not included in the returned byte length, but does require space in the bound buffer. For example, suppose an application uses strings composed of character data in the ASCII character set, a driver has 50 characters of data to return, and the application’s buffer is 25 bytes long. In the application’s buffer, the driver returns the first 24 characters followed by a null-termination character. In the length/indicator buffer, it returns a byte length of 50.

The application can restrict the number of rows in the result set by setting the `SQL_ATTR_MAX_ROWS` statement attribute before executing the statement that creates the result set. For example, the preview mode in an application used to format reports needs only enough data to display the first page of the report. By restricting the size of the result set, such a feature would run faster. This statement attribute is intended to reduce network traffic and might not be supported by all drivers.

## Getting Long Data

DBMSs define *long data* as any character or binary data over a certain size, such as 255 characters. This data may be small enough to be stored in a single buffer, such as a part description of several thousand characters. However, it may be too long to store in memory, such as long text documents or bitmaps. Because such data cannot be stored in a single buffer, it is retrieved from the driver in parts with **SQLGetData** after the other data in the row has been fetched.

**Note** An application can actually retrieve any type of data with **SQLGetData**, not just long data, although only character and binary data can be retrieved in parts. However, if the data is small enough to fit in a single buffer, there is generally no reason to use **SQLGetData**. It is much easier to bind a buffer to the column and let the driver return the data in the buffer.

To retrieve long data from a column, an application first calls **SQLFetchScroll** or **SQLFetch** to move to a row and fetch the data for bound columns. The application then calls **SQLGetData**. **SQLGetData** has the same arguments as **SQLBindCol**: a statement handle; a column number; the C data type, address, and byte length of an application variable; and the address of a length/indicator buffer. Both functions have the same arguments because they perform essentially the same task: They both describe an application variable to the driver and specify that the data for a particular column should be returned in that variable. The major differences are that **SQLGetData** is called after a row is fetched (and is sometimes called *late binding* for this reason), and that the binding specified by **SQLGetData** lasts only for the duration of the call.

With respect to a single column, **SQLGetData** behaves in the same manner as **SQLFetch**: It retrieves the data for the column, converts it to the type of the application variable, and returns it in that variable. It also returns the byte length of the data in the length/indicator buffer. For more information on how **SQLFetch** returns data, see the immediately preceding section, “[Fetching a Row of Data](#).”

**SQLGetData** differs from **SQLFetch** in one important respect. If it is called more than once in succession for the same column, each call returns a successive part of the data. Each call except the last call returns **SQL\_SUCCESS\_WITH\_INFO** and **SQLSTATE 01004** (String data, right truncated); the last call returns **SQL\_SUCCESS**. This is how **SQLGetData** is used to retrieve long data in parts. When there is no more data to return, **SQLGetData** returns **SQL\_NO\_DATA**. The application is responsible for putting the long data together, which might mean concatenating the parts of the data. Each part is null-terminated; the application must remove the null-termination character if concatenating the parts. Retrieving data in parts can be done for variable-length bookmarks, as well as other long data. The value returned in the length/indicator buffer decreases in each call by the number of bytes returned in the previous call, although it is common for the driver to be unable to discover the amount of available data and return a byte length of **SQL\_NO\_TOTAL**. For example:

```
// Declare a binary buffer to retrieve 5000 bytes of data at a time.
SQLCHAR    BinaryPtr[5000];
SQLINTEGER PartID;
SQLINTEGER PartIDInd, BinaryLenOrInd, NumBytes;
SQLRETURN rc;
SQLHSTMT   hstmt;

// Create a result set containing the ID and picture of each part.
SQLExecDirect(hstmt, "SELECT PartID, Picture FROM Pictures", SQL_NTS);

// Bind PartID to the PartID column.
```

```

SQLBindCol(hstmt, 1, SQL_C_ULONG, &PartID, 0, &PartIDInd);

// Retrieve and display each row of data.
while ((rc = SQLFetch(hstmt)) != SQL_NO_DATA) {
    // Display the part ID and initialize the picture.
    DisplayID(PartID, PartIDInd);
    InitPicture();

    // Retrieve the picture data in parts. Send each part and the number of bytes
in
    // each part to a function that displays it. The number of bytes is always 5000
if
    // there were more than 5000 bytes available to return (cbBinaryBuffer > 5000).
    // Code to check if rc equals SQL_ERROR or SQL_SUCCESS_WITH_INFO not shown.
    while ((rc = SQLGetData(hstmt, 2, SQL_C_BINARY, BinaryPtr, sizeof(BinaryPtr),
        &BinaryLenOrInd)) != SQL_NO_DATA) {
        NumBytes = (BinaryLenOrInd > 5000) || (BinaryLenOrInd == SQL_NO_TOTAL) ?
            5000 : BinaryLenOrInd;
        DisplayNextPictPart(BinaryPtr, NumBytes);
    }
}

// Close the cursor.
SQLCloseCursor(hstmt);

```

- There are a number of restrictions on using **SQLGetData**. In general, columns accessed with **SQLGetData**:
  - Must be accessed in order of increasing column number (because of the way the columns of a result set are read from the data source). For example, it is an error to call **SQLGetData** for column 5 and then call it for column 4.
  - Cannot be bound.
  - Must have a higher column number than the last bound column. For example, if the last bound column is column 3, it is an error to call **SQLGetData** for column 2. For this reason, applications should be careful to place long data columns at the end of the select list.

Cannot be used if **SQLFetch** or **SQLFetchScroll** was called to retrieve more than one row. For more information, see [“Using Block Cursors”](#) in Chapter 11, “Retrieving Results (Advanced).”

Some drivers do not enforce these restrictions. Interoperable applications should either assume they exist or determine which restrictions are not enforced by calling **SQLGetInfo** with the **SQL\_GETDATA\_EXTENSIONS** option.

If the application does not need all of the data in a character or binary data column, it can reduce network traffic in DBMS-based drivers by setting the **SQL\_ATTR\_MAX\_LENGTH** statement attribute before executing the statement. This restricts the number of bytes of data that will be returned for any character or binary column. For example, suppose a column contains long text documents. An application that browses the table containing this column might need to display only the first page of each document. Although this statement attribute can be simulated in the driver, there is no reason to do so. In particular, if an application



wants to truncate character or binary data, it should bind a small buffer to the column with **SQLBindCol** and let the driver truncate the data.

## Closing the Cursor

When an application has finished using a cursor, it calls **SQLCloseCursor** to close the cursor. For example:

```
SQLCloseCursor(hstmt);
```

Until the application closes the cursor, the statement on which the cursor is opened cannot be used for most other operations, such as executing another SQL statement. For a complete list of functions that can be called while a cursor is open, see Appendix B, “ODBC Transition State Tables” contained on the Microsoft Web site (ODBC Programmer’s Reference).

**Note** To close a cursor, an application should call **SQLCloseCursor**, not **SQLCancel**.

Cursors remain open until they are explicitly closed, except when a transaction is committed or rolled back, in which case some data sources close the cursor. In particular, reaching the end of the result set, when **SQLFetch** returns **SQL\_NO\_DATA**, does not close a cursor. Even cursors on empty result sets (result sets created when a statement executed successfully but which returned no rows) must be explicitly closed.

## Chapter 11: Overview of Retrieving Results (Advanced)

An application can specify that an offset is added to bound data buffer addresses and the corresponding length/indicator buffer addresses when **SQLBulkOperations**, **SQLFetch**, **SQLFetchScroll**, or **SQLSetPos** is called. The results of these additions determine the addresses used in these operations.

Bind offsets allow an application to change bindings without calling **SQLBindCol** for previously bound columns. A call to **SQLBindCol** to rebind data changes the buffer address and the length/indicator pointer. Rebinding with an offset, on the other hand, simply adds an offset to the existing bound data buffer address and length/indicator buffer address. When offsets are used, the bindings are a “template” of how the application buffers are laid out and the application can move this “template” to different areas of memory by changing the offset. A new offset can be specified at any time, and is always added to the originally bound values.

To specify a bind offset, the application sets the `SQL_ATTR_ROW_BIND_OFFSET_PTR` statement attribute to the address of an `SQLINTEGER` buffer. Before the application calls a function that uses the bindings, such as **SQLBulkOperations**, **SQLFetch**, **SQLFetchScroll**, or **SQLSetPos**, it places an offset in bytes in this buffer, as long as neither the data buffer address nor the length/indicator buffer address is 0, and the bound column is in the result set. The sum of the address and the offset must be a valid address. (This means that either or both of the offset and the address to which the offset is added can be invalid, as long as the sum of them is a valid address.) The `SQL_ATTR_ROW_BIND_OFFSET_PTR` statement attribute is a pointer so that the offset value can be applied to more than one set of binding data, all of which can be changed by changing one offset value. An application must make sure that the pointer remains valid until the cursor is closed.

**Note** Binding offsets are not supported by ODBC 2.x drivers.

### Block Cursors

Many applications spend a significant amount of time bringing data across the network. Part of this time is spent actually bringing the data across the network and part of it is spent on network overhead, such as the call made by the driver to request a row of data. The latter time can be reduced if the application makes efficient use of block, or fat, cursors, which can return more than one row at a time.

An application always has the option of using a block cursor. On data sources from which only one row at a time can be fetched, block cursors must be simulated in the driver. This can be done by performing multiple single-row fetches. While this is unlikely to provide any performance gains, it opens opportunities for applications. Such applications will then experience performance increases as DBMSs implement block cursors natively and the drivers associated with those DBMSs expose them.

The rows returned in a single fetch with a block cursor are called the rowset. It is important not to confuse the rowset with the result set. The result set is maintained at the data source, while the rowset is maintained in application buffers. While the result set is fixed, the rowset is not—it changes position and contents each time a new set of rows is fetched. Just as a single-row cursor such as the traditional SQL forward-only cursor points to a current row, a block cursor points to the rowset, which can be thought of as “current rows.”

To perform operations that operate on a single row when multiple rows have been fetched, the application must first indicate which row is the current row. The current row is required by calls to `SQLGetData` and positioned update and delete statements. When a block cursor first returns a rowset, the current row is the first row of the rowset. To change the current row, the application calls `SQLSetPos` or `SQLBulkOperations` (to update by bookmark). The following figure shows the relationship of the result set, rowset, current row, rowset cursor, and block cursor. For more information, see [“Using Block Cursors,”](#) later in this chapter, and [“Positioned Update and Delete Statements”](#) and [“Updating Data with SQLSetPos”](#) in Chapter 12, “Updating Data.”

### ***Block cursor and rowset cursor***

Whether a cursor is a block cursor is independent of whether it is scrollable. For example, most of the work in a report application is spent retrieving and printing rows. Because of this, it will work fastest with a forward-only, block cursor. It uses a forward-only cursor to avoid the expense of a scrollable cursor, and a block cursor to reduce the network traffic.

## **Binding Columns for Use with Block Cursors**

Because block cursors return multiple rows, applications that use them must bind an array of variables to each column instead of a single variable. These arrays are collectively known as the *rowset buffers*. Here are the two styles of binding:

- Bind an array to each column. This is called *column-wise binding* because each data structure (array) contains data for a single column.
- Define a structure to hold the data for an entire row and bind an array of these structures. This is called *row-wise binding* because each data structure contains the data for a single row.

As when the application binds single variables to columns, it calls **SQLBindCol** to bind arrays to columns. The only difference is that the addresses passed are array addresses, not single variable addresses. The application sets the `SQL_ATTR_ROW_BIND_TYPE` statement attribute to specify whether it is using column-wise or row-wise binding. Whether to use column-wise or row-wise binding is largely a matter of application preference. Row-wise binding might correspond more closely to the application’s layout of data, in which case it would provide better performance.

### ***Column-Wise Binding***

When using column-wise binding, an application binds one or two, or in some cases three, arrays to each column for which data is to be returned. The first array holds the data values and the second array holds length/indicator buffers. Indicators and length values can be stored in separate buffers by setting the `SQL_DESC_INDICATOR_PTR` and `SQL_DESC_OCTET_LENGTH_PTR` descriptor fields to different values; if this is done, a third array is bound. Each array contains as many elements as there are rows in the rowset.

The application declares that it is using column-wise binding with the `SQL_ATTR_ROW_BIND_TYPE` statement attribute, which determines the bind type for rowset buffers, as opposed to parameter set buffers. The driver returns the data for each row in successive elements of each array. This figure shows how column-wise binding works.

## *Column-wise binding of column data*

For example, the following code binds 10-element arrays to the OrderID, SalesPerson, and Status columns.

```
#define ROW_ARRAY_SIZE 10

SQLINTEGER OrderIDArray[ROW_ARRAY_SIZE], NumRowsFetched;
SQLCHAR SalesPersonArray[ROW_ARRAY_SIZE][11], StatusArray[ROW_ARRAY_SIZE][7];
SQLINTEGER OrderIDIndArray[ROW_ARRAY_SIZE],
SalesPersonLenOrIndArray[ROW_ARRAY_SIZE],
StatusLenOrIndArray[ROW_ARRAY_SIZE];
SQLUSMALLINT RowStatusArray[ROW_ARRAY_SIZE], i;
SQLRETURN rc;
SQLHSTMT hstmt;

// Set the SQL_ATTR_ROW_BIND_TYPE statement attribute to use column-wise
binding.
// Declare the rowset size with the SQL_ATTR_ROW_ARRAY_SIZE statement attribute.
Set the
// SQL_ATTR_ROW_STATUS_PTR statement attribute to point to the row status array.
Set
// the SQL_ATTR_ROWS_FETCHED_PTR statement attribute to point to cRowsFetched.
SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_BIND_TYPE, SQL_BIND_BY_COLUMN, 0);
SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_ARRAY_SIZE, ROW_ARRAY_SIZE, 0);
SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_STATUS_PTR, RowStatusArray, 0);
SQLSetStmtAttr(hstmt, SQL_ATTR_ROWS_FETCHED_PTR, &NumRowsFetched, 0);

// Bind arrays to the OrderID, SalesPerson, and Status columns.
SQLBindCol(hstmt, 1, SQL_C_ULONG, OrderIDArray, 0, OrderIDIndArray);
SQLBindCol(hstmt, 2, SQL_C_CHAR, SalesPersonArray, sizeof(SalesPersonArray[0]),
SalesPersonLenOrIndArray);
SQLBindCol(hstmt, 3, SQL_C_CHAR, StatusArray, sizeof(StatusArray[0]),
StatusLenOrIndArray);

// Execute a statement to retrieve rows from the Orders table.
SQLExecDirect(hstmt, "SELECT OrderID, SalesPerson, Status FROM Orders",
SQL_NTS);

// Fetch up to the rowset size number of rows at a time. Print the actual number
of rows fetched; this number
// is returned in NumRowsFetched. Check the row status array to print only those
rows
// successfully fetched. Code to check if rc equals SQL_SUCCESS_WITH_INFO or
SQL_ERROR
// not shown.
while ((rc = SQLFetchScroll(hstmt,SQL_FETCH_NEXT,0)) != SQL_NO_DATA) {
    for (i = 0; i < NumRowsFetched; i++) {
        if ((RowStatusArray[i] == SQL_ROW_SUCCESS) ||
            (RowStatusArray[i] == SQL_ROW_SUCCESS_WITH_INFO)) {
            if (OrderIDIndArray[i] == SQL_NULL_DATA)
                printf(" NULL ");
            else
                printf("%d\t", OrderIDArray[i]);
        }
    }
}
```

```

        if (SalesPersonLenOrIndArray[i] == SQL_NULL_DATA)
            printf(" NULL  ");
        else
            printf("%s\t", SalesPersonArray[i]);
        if (StatusLenOrIndArray[i] == SQL_NULL_DATA)
            printf(" NULL\n");
        else
            printf("%s\n", StatusArray[i]);
    }
}
}

// Close the cursor.
SQLCloseCursor(hstmt);

```

## ***Row-Wise Binding***

When using row-wise binding, an application defines a structure containing one or two, or in some cases three, elements for each column for which data is to be returned. The first element holds the data value and the second element holds the length/indicator buffer. Indicators and length values can be stored in separate buffers by setting the `SQL_DESC_INDICATOR_PTR` and `SQL_DESC_OCTET_LENGTH_PTR` descriptor fields to different values; if this is done, the structure contains a third element. The application then allocates an array of these structures, which contains as many elements as there are rows in the rowset.

The application declares the size of the structure to the driver with the `SQL_ATTR_ROW_BIND_TYPE` statement attribute and binds the address of each member in the first element of the array. Thus, the driver can calculate the address of the data for a particular row and column as:

$$\text{Address} = \text{Bound Address} + ((\text{Row Number} - 1) * \text{Structure Size})$$

where rows are numbered from 1 to the size of the rowset. (One is subtracted from the row number because array indexing in C is zero-based.) The following figure shows how row-wise binding works. Generally, only columns that will be bound are included in the structure. The structure can contain fields that are unrelated to result set columns. The columns can be placed in the structure in any order, but are shown in sequential order for clarity.

## ***Row-wise binding of column data***

For example, the following code creates a structure with elements in which to return data for the `OrderID`, `SalesPerson`, and `Status` columns, and length/indicators for the `SalesPerson` and `Status` columns. It allocates 10 of these structures and binds them to the `OrderID`, `SalesPerson`, and `Status` columns.

```

#define ROW_ARRAY_SIZE 10

// Define the ORDERINFO struct and allocate an array of 10 structs.
typedef struct {
    SQLINTEGER OrderID;
    SQLINTEGER OrderIDInd;
    SQLCHAR    SalesPerson[11];
    SQLINTEGER SalesPersonLenOrInd;
    SQLCHAR    Status[7];
}

```

```

    SQLINTEGER StatusLenOrInd;
} ORDERINFO;
ORDERINFO OrderInfoArray[ROW_ARRAY_SIZE];

SQLUIINTEGER NumRowsFetched;
SQLUSMALLINT RowStatusArray[ROW_ARRAY_SIZE], i;
SQLRETURN rc;
SQLHSTMT hstmt;

// Specify the size of the structure with the SQL_ATTR_ROW_BIND_TYPE statement
// attribute. This also declares that row-wise binding will be used. Declare the
rowset
// size with the SQL_ATTR_ROW_ARRAY_SIZE statement attribute. Set the
// SQL_ATTR_ROW_STATUS_PTR statement attribute to point to the row status array.
Set
// the SQL_ATTR_ROWS_FETCHED_PTR statement attribute to point to NumRowsFetched.
SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_BIND_TYPE, sizeof(ORDERINFO), 0);
SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_ARRAY_SIZE, ROW_ARRAY_SIZE, 0);
SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_STATUS_PTR, RowStatusArray, 0);
SQLSetStmtAttr(hstmt, SQL_ATTR_ROWS_FETCHED_PTR, &NumRowsFetched, 0);

// Bind elements of the first structure in the array to the OrderID,
SalesPerson, and
// Status columns.
SQLBindCol(hstmt, 1, SQL_C_ULONG, &OrderInfoArray[0].OrderID, 0,
&OrderInfoArray[0].OrderIDInd);
SQLBindCol(hstmt, 2, SQL_C_CHAR, OrderInfoArray[0].SalesPerson,
sizeof(OrderInfoArray[0].SalesPerson),
&OrderInfoArray[0].SalesPersonLenOrInd);
SQLBindCol(hstmt, 3, SQL_C_CHAR, OrderInfoArray[0].Status,
sizeof(OrderInfoArray[0].Status), &OrderInfoArray[0].StatusLenOrInd);

// Execute a statement to retrieve rows from the Orders table.
SQLExecDirect(hstmt, "SELECT OrderID, SalesPerson, Status FROM Orders",
SQL_NTS);

// Fetch up to the rowset size number of rows at a time. Print the actual number
of rows fetched; this number
// is returned in NumRowsFetched. Check the row status array to print only those
rows
// successfully fetched. Code to check if rc equals SQL_SUCCESS_WITH_INFO or
SQL_ERROR
// not shown.
while ((rc = SQLFetchScroll(hstmt,SQL_FETCH_NEXT,0)) != SQL_NO_DATA) {
    for (i = 0; i < NumRowsFetched; i++) {
        if (RowStatusArray[i] == SQL_ROW_SUCCESS || RowStatusArray[i] ==
SQL_ROW_SUCCESS_WITH_INFO) {
            if (OrderInfoArray[i].OrderIDInd == SQL_NULL_DATA)
                printf(" NULL ");
            else
                printf("%d\t", OrderInfoArray[i].OrderID);
            if (OrderInfoArray[i].SalesPersonLenOrInd == SQL_NULL_DATA)
                printf(" NULL ");
            else

```

```

        printf("%s\t", OrderInfoArray[i].SalesPerson);
        if (OrderInfoArray[i].StatusLenOrInd == SQL_NULL_DATA)
            printf(" NULL\n");
        else
            printf("%s\n", OrderInfoArray[i].Status);
    }
}

// Close the cursor.
SQLCloseCursor(hstmt);

```

## Using Block Cursors

Support for block cursors is built into ODBC 3.x. `SQLFetch` can be used only for multirow fetches when called in ODBC 3.x; if an ODBC 2.x application calls `SQLFetch`, it will open only a single-row, forward-only cursor. When an ODBC 3.x application calls `SQLFetch` in an ODBC 2.x driver, it returns a single row unless the driver supports `SQLExtendedFetch`. For more information, see Appendix G, “Driver Guidelines for Backward Compatibility” contained on the Microsoft Web site (ODBC Programmer’s Guide).

To use block cursors, the application sets the rowset size, binds the rowset buffers (as described in the previous section), optionally sets the `SQL_ATTR_ROWS_FETCHED_PTR` and `SQL_ATTR_ROW_STATUS_PTR` statement attributes, and calls `SQLFetch` or `SQLFetchScroll` to fetch a block of rows. Note that the application can change the rowset size and bind new rowset buffers (by calling `SQLBindCol` or specifying a bind offset) even after rows have been fetched.

## Rowset Size

Which rowset size to use depends on the application. Screen-based applications commonly follow one of two strategies. The first is to set the rowset size to the number of rows displayed on the screen; if the user resizes the screen, the application changes the rowset size accordingly. The second is to set the rowset size to a larger number, such as 100, which reduces the number of calls to the data source. The application scrolls locally within the rowset when possible and fetches new rows only when it scrolls outside the rowset.

Other applications, such as reports, tend to set the rowset size to the largest number of rows the application can reasonably handle—with a larger rowset, the network overhead per row is sometimes reduced. Exactly how large a rowset can be depends on the size of each row and the amount of memory available.

Rowset size is set by a call to **`SQLSetStmtAttr`** with an *Attribute* argument of `SQL_ATTR_ROW_ARRAY_SIZE`. The application can change the rowset size, bind new rowset buffers (by calling **`SQLBindCol`** or specifying a binding offset) even after rows have been fetched, or both. The implications of changing the rowset size depend on the function:

- **`SQLFetch`** and **`SQLFetchScroll`** use the rowset size at the time of the call to determine how many rows to fetch. Note, however, that **`SQLFetchScroll`** with a *FetchOrientation* of `SQL_FETCH_NEXT` increments the cursor based on the rowset of the previous fetch, and then fetches a rowset based on the current rowset size.

- **SQLSetPos** uses the rowset size that is in effect as of the preceding call to **SQLFetch** or **SQLFetchScroll**, because **SQLSetPos** operates on a rowset that has already been set. **SQLSetPos** also will pick up the new rowset size if **SQLBulkOperations** has been called after the rowset size was changed.
- **SQLBulkOperations** uses the rowset size in effect at the time of the call, because it performs operations on a table independent of any fetched rowset.

## Number of Rows Fetched and Status

If the `SQL_ATTR_ROWS_FETCHED_PTR` statement attribute has been set, it specifies a buffer that returns the number of rows fetched by the call to **SQLFetch** or **SQLFetchScroll**, and error rows. (This number is a count of all rows that do not have the status `SQL_ROW_NO_ROWS`.) After a call to **SQLBulkOperations** or **SQLSetPos**, the buffer contains the number of rows that were affected by a bulk operation performed by the function. If the `SQL_ATTR_ROW_STATUS_PTR` statement attribute has been set, **SQLFetch** or **SQLFetchScroll** returns the row status array, which provides the status of each returned row. Both of the buffers pointed to by these fields are allocated by the application and populated by the driver. An application must make sure that these pointers remain valid until the cursor is closed.

Entries in the row status array state whether each row was fetched successfully, whether it was updated, added, or deleted since it was last fetched, and whether an error occurred while fetching the row. Note that if **SQLFetch** or **SQLFetchScroll** encounters an error while retrieving one row of a multirow rowset, or if **SQLBulkOperations** with an `Operation` argument of `SQL_FETCH_BY_BOOKMARK` encounters an error while performing a bulk fetch, it sets the corresponding value in the row status array to `SQL_ROW_ERROR`, continues fetching rows, and returns `SQL_SUCCESS_WITH_INFO`. For more information about error handling and the row status array, see the **SQLFetch** and **SQLFetchScroll** function descriptions.

## SQLGetData and Block Cursors

**SQLGetData** operates on a single column of a single row and cannot fetch an array containing data from multiple rows. The reason for this is that the primary use of **SQLGetData** is to fetch long data in parts, and there is little or no reason to do this for more than one row at a time.

To use **SQLGetData** with a block cursor, an application first calls **SQLSetPos** to position the cursor on a single row. It then calls **SQLGetData** for a column in that row. However, this behavior is optional. To determine if a driver supports the use of **SQLGetData** with block cursors, an application calls **SQLGetInfo** with the `SQL_GETDATA_EXTENSIONS` option.

## Row Status Array

In addition to data, **SQLFetch** and **SQLFetchScroll** can return an array that gives the status of each row in the rowset. This array is specified through the `SQL_ATTR_ROW_STATUS_PTR` statement attribute. This array is allocated by the application and must have as many elements as are specified by the `SQL_ATTR_ROW_ARRAY_SIZE` statement attribute. The values in the array are set by **SQLBulkOperations**, **SQLFetch**, **SQLFetchScroll**, and **SQLSetPos**. The values describe the status of the row and whether that status has changed since it was last fetched:



| Row status array value    | Description  |
|---------------------------|--|
| SQL_ROW_SUCCESS           | The row was successfully fetched and has not changed since it was last fetched.  |
| SQL_ROW_SUCCESS_WITH_INFO | The row was successfully fetched and has not changed since it was last fetched. However, a warning was returned about the row.   |
| SQL_ROW_ERROR             | An error occurred while fetching the row.  |
| SQL_ROW_UPDATED           | <p>The row was successfully fetched and has been updated since it was last fetched. If the row is fetched again, or refreshed by <b>SQLSetPos</b>, its status is changed to the new status.</p> <p>Some drivers cannot detect changes to data, and therefore cannot return this value. To determine whether a driver can detect updates to refetched rows, an application calls <b>SQLGetInfo</b> with the SQL_ROW_UPDATES option.</p> |
| SQL_ROW_DELETED           | The row has been deleted since it was last fetched.  |
| SQL_ROW_ADDED             | <p>The row was inserted by <b>SQLBulkOperations</b>. If the row is fetched again, or is refreshed by <b>SQLSetPos</b>, its status is SQL_ROW_SUCCESS.</p> <p>This value is not set by <b>SQLFetch</b> or <b>SQLFetchScroll</b>.</p>  |
| SQL_ROW_NOROW             | The rowset overlapped the end of the result set and no row was returned that corresponded to this element of the row status array.   |

## Scrollable Cursors

In modern screen-based applications, the user scrolls backward and forward through the data. For such applications, returning to a previously fetched row is a problem. One possibility is to close and reopen the cursor, then fetch rows until the cursor reaches the required row. Another possibility is to read the result set, cache it locally, and implement scrolling in the application. Both possibilities work well only with small result sets, and the latter possibility is difficult to implement. A better solution is to use a *scrollable cursor*, which can move backward and forward in the result set.

A *scrollable cursor* is commonly used in modern screen-based applications in which the user scrolls back and forth through the data. However, applications should use scrollable cursors only when forward-only cursors will not do the job, as scrollable cursors are generally more expensive than forward-only cursors.

The ability to move backward raises a question not applicable to forward-only cursors: Should a scrollable cursor detect changes made to rows previously fetched? That is, should it detect updated, deleted, and newly inserted rows?

This question arises because the definition of a result set—the set of rows that matches certain criteria—does not state when rows are checked to see if they match that criteria, nor does it state whether rows must contain the same data each time they are fetched. The former omission makes it possible for scrollable cursors to detect whether rows have been inserted or deleted, while the latter makes it possible for them to detect updated data.

The ability to detect changes is sometimes useful, sometimes not. For example, an accounting application needs a cursor that ignores all changes; balancing books is impossible if the cursor shows the latest changes. On the other hand, an airline reservation system needs a cursor that shows the latest changes to the data; without such a cursor, it must continually requery the database to show the most up-to-date flight availability.

To cover the needs of different applications, ODBC defines four different types of scrollable cursors. These cursors vary both in expense and in their ability to detect changes to the result set. Note that if a scrollable cursor can detect changes to rows, it can only detect them when it attempts to refetch those rows; there is no way for the data source to notify the cursor of changes to the currently fetched rows. Note also that visibility of changes is also controlled by the transaction isolation level; for more information, see [“Transaction Isolation”](#) in Chapter 14, “Transactions.”

## Scrollable Cursor Types

The four types of scrollable cursors are: static, dynamic, keyset-driven, and mixed. Static cursors detect few or no changes, but are relatively cheap to implement. Dynamic cursors detect all changes, but are expensive to implement. Keyset-driven and mixed cursors lie in between, detecting most changes but at less expense than dynamic cursors.

The following terms are used to define the characteristics of each type of scrollable cursor:

- **Own updates, deletes, and inserts.** Updates, deletes, and inserts made through the cursor, either with a call to `SQLBulkOperations` or `SQLSetPos` or with a positioned update or delete statement.
- **Other updates, deletes, and inserts.** Updates, deletes, and inserts not made by the cursor, including those made by other operations in the same transaction, those made through other transactions, and those made by other applications.
- **Membership.** The set of rows in the result set.
- **Order.** The order in which rows are returned by the cursor.
- **Values.** The values in each row in the result set.

For information about how to update, delete, and insert data, see [“Chapter 12: Overview of Updating Data.”](#)

### *Static Cursors*

A static cursor is one in which the result set appears to be static. It does not usually detect changes made to the membership, order, or values of the result set after the cursor is opened. For example, suppose a static cursor fetches a row, and another application then updates that row. If the static cursor refetches the row, the values it sees are unchanged, despite the changes made by the other application.

Static cursors may detect their own updates, deletes, and inserts, although they are not required to do so. Whether a particular static cursor detects these changes is reported through the

SQL\_STATIC\_SENSITIVITY option in SQLGetInfo. Static cursors never detect other updates, deletes, and inserts.

The row status array specified by the SQL\_ATTR\_ROW\_STATUS\_PTR statement attribute can contain SQL\_ROW\_SUCCESS, SQL\_ROW\_SUCCESS\_WITH\_INFO, or SQL\_ROW\_ERROR for any row. It returns SQL\_ROW\_UPDATED, SQL\_ROW\_DELETED, or SQL\_ROW\_ADDED for rows updated, deleted, or inserted by the cursor, assuming that the cursor is capable of detecting such changes.

Static cursors are commonly implemented by locking the rows in the result set or making a copy, or snapshot, of the result set. While locking rows is relatively easy to do, it has the drawback of significantly reducing concurrency. Making a copy allows greater concurrency and allows the cursor to keep track of its own updates, deletes, and inserts by modifying the copy. However, a copy is more expensive to make and can diverge from the underlying data as that data is changed by others.

### ***Keyset-Driven Cursors***

A keyset-driven cursor lies between a static and a dynamic cursor in its ability to detect changes. Like a static cursor, it does not always detect changes to the membership and order of the result set. Like a dynamic cursor, it does detect changes to the values of rows in the result set (subject to the isolation level of the transaction, as set by the SQL\_ATTR\_TXN\_ISOLATION connection attribute).

When a keyset-driven cursor is opened, it saves the keys for the entire result set; this fixes the apparent membership and order of the result set. As the cursor scrolls through the result set, it uses the keys in this *keyset* to retrieve the current data values for each row. For example, suppose a keyset-driven cursor fetches a row, and another application then updates that row. If the cursor refetches the row, the values it sees are the new ones, because it refetched the row using its key. Because of this, the keyset-driven cursors always detect changes made by themselves and others.

When the cursor attempts to retrieve a row that has been deleted, this row appears as a “hole” in the result set: The key for the row exists in the keyset but the row no longer exists in the result set. If the key values in a row are updated, the row is considered to have been deleted and then inserted, so such rows also appear as holes in the result set. While a keyset-driven cursor can always detect rows deleted by others, it can optionally remove the keys for rows it deletes itself from the keyset. Keyset-driven cursors that do this cannot detect their own deletes. Whether a particular keyset-driven cursor detects its own deletes is reported through the SQL\_STATIC\_SENSITIVITY option in **SQLGetInfo**.

Rows inserted by others are never visible to a keyset-driven cursor because no keys for these rows exist in the keyset. However, a keyset-driven cursor can optionally add the keys for rows it inserts itself to the keyset. Keyset-driven cursors that do this can detect their own inserts. Whether a particular keyset-driven cursor detects its own inserts is reported through the SQL\_STATIC\_SENSITIVITY option in **SQLGetInfo**.

The row status array specified by the SQL\_ATTR\_ROW\_STATUS\_PTR statement attribute can contain SQL\_ROW\_SUCCESS, SQL\_ROW\_SUCCESS\_WITH\_INFO, or SQL\_ROW\_ERROR for any row. It returns SQL\_ROW\_UPDATED, SQL\_ROW\_DELETED, or SQL\_ROW\_ADDED for rows it detects as updated, deleted, or inserted.

Keyset-driven cursors are commonly implemented by creating a temporary table that contains the keys for each row in the result set. Because the cursor must also determine if rows have been updated, this table also commonly contains a column with row versioning information.

To scroll over the original result set, the keyset-driven cursor opens a static cursor over the temporary table. To retrieve a row in the original result set, the cursor first retrieves the appropriate key from the temporary table, and then retrieves the current values for the row. If block cursors are used, the cursor must retrieve multiple keys and rows.

### ***Mixed Cursors***

A mixed cursor is a combination of a keyset-driven cursor and a dynamic cursor. It is used when the result set is too large to reasonably save keys for the entire result set. Mixed cursors are implemented by creating a keyset that is smaller than the entire result set but larger than the rowset.

As long as the application scrolls within the keyset, the behavior is keyset-driven. When the application scrolls outside the keyset, the behavior is dynamic: The cursor fetches the requested rows and creates a new keyset. Note that after the new keyset is created, the behavior reverts to keyset-driven within that keyset.

For example, suppose a result set has 1,000 rows and uses a mixed cursor with a keyset size of 100 and a rowset size of 10. When the first rowset is fetched, the cursor creates a keyset consisting of the keys for the first 100 rows. It then returns the first 10 rows, as requested.

Now suppose another application deletes rows 11 and 101. If the cursor attempts to retrieve row 11, it will encounter a hole because it has a key for this row but no row exists; this is keyset-driven behavior. If the cursor attempts to retrieve row 101, the cursor will not detect that the row is missing because it does not have a key for the row. Instead, it will retrieve what was previously row 102. This is dynamic cursor behavior.

A mixed cursor is equivalent to a keyset-driven cursor when the keyset size is equal to the result set size. A mixed cursor is equivalent to a dynamic cursor when the keyset size is equal to 1.

## **Using Scrollable Cursors**

Using a scrollable cursor requires these three steps:

1. Determine the cursor capabilities.
2. Set up the cursor.
3. Scroll and fetch rows.

## **Determining Cursor Capabilities**

The following four options in **SQLGetInfo** describe what types of cursors are supported and what their capabilities are:

- **SQL\_CURSOR\_SENSITIVITY**. Indicates whether a cursor is sensitive to changes made by another cursor.
- **SQL\_SCROLL\_OPTIONS**. Lists the supported cursor types (forward-only, static, keyset-driven, dynamic, or mixed). All data sources must support forward-only cursors.

- **SQL\_DYNAMIC\_CURSOR\_ATTRIBUTES1**, **SQL\_FORWARD\_ONLY\_CURSOR\_ATTRIBUTES1**, **SQL\_KEYSET\_CURSOR\_ATTRIBUTES1**, or **SQL\_STATIC\_CURSOR\_ATTRIBUTES1** (depending on the type of the cursor). Lists the fetch types supported by scrollable cursors. The bits in the return value correspond to the fetch types in **SQLFetchScroll**.
- **SQL\_KEYSET\_CURSOR\_ATTRIBUTES2** or **SQL\_STATIC\_CURSOR\_ATTRIBUTES2** (depending on the type of the cursor). Lists whether static and keyset-driven cursors can detect their own updates, deletes, and inserts.

An application can determine cursor capabilities at run time by calling **SQLGetInfo** with these options. This is commonly done by generic applications. Cursor capabilities also can be determined during application development and their use hard-coded into the application. This is commonly done by vertical and custom applications, but can also be done by generic applications that use a client-side cursor implementation such as the ODBC cursor library.

## Setting Up the Cursor

The application can specify the cursor type before executing a statement that creates a result set. It does this with the **SQL\_ATTR\_CURSOR\_TYPE** statement attribute. If the application does not explicitly specify a type, a forward-only cursor will be used. To get a mixed cursor, an application specifies a keyset-driven cursor but declares a keyset size less than the result set size.

For keyset-driven and mixed cursors, the application can also specify the keyset size. It does this with the **SQL\_ATTR\_KEYSET\_SIZE** statement attribute. If the keyset size is set to 0—which is the default—the keyset size is set to the result set size and a keyset-driven cursor is used. Note that the keyset size can be changed after the cursor has been opened.

The application can also set the rowset size; for more information, see [“Using Block Cursors,”](#) earlier in this chapter.

## Cursor Characteristics and Cursor Type

An application can specify the characteristics of a cursor rather than specifying the cursor type (forward-only, static, keyset-driven, or dynamic). To do so, the application selects the cursor’s scrollability (by setting the **SQL\_ATTR\_CURSOR\_SCROLLABLE** statement attribute) and sensitivity (by setting the **SQL\_ATTR\_CURSOR\_SENSITIVITY** statement attribute) before opening the cursor on the statement handle. The driver then chooses the cursor type that most efficiently provides the characteristics that the application requested.

Whenever an application sets any of the statement attributes **SQL\_ATTR\_CONCURRENCY**, **SQL\_ATTR\_CURSOR\_SCROLLABLE**, **SQL\_ATTR\_CURSOR\_SENSITIVITY**, or **SQL\_ATTR\_CURSOR\_TYPE**, the driver makes any required change to the other statement attributes in this set of four attributes, so that their values remain consistent. As a result, when the application specifies a cursor characteristic, the driver can change the attribute that indicates cursor type based on this implicit selection; when the application specifies a type, the driver can change any of the other attributes to be consistent with the characteristics of the selected type. For more information on these statement attributes, see the **SQLSetStmtAttr** function description in the Part II PDF file, “ODBC API Reference” available on the Solid Web site.

An application that sets statement attributes to specify both a cursor type and cursor characteristics runs the risk of obtaining a cursor that is not the most efficient method available on that driver of meeting the application's requirements.

The implicit setting of statement attributes is driver-defined except that it must follow these rules:

- Forward-only cursors are never scrollable; see the definition of SQL\_ATTR\_CURSOR\_SCROLLABLE in SQLSetStmtAttr function description in the Part II PDF file, "ODBC API Reference" available on the Solid Web site.
- Insensitive cursors are never updatable (and thus their concurrency is read-only); this is based on their definition of insensitive cursors in the ISO SQL standard.

Consequently, the implicit setting of statement attributes occurs in the following cases:

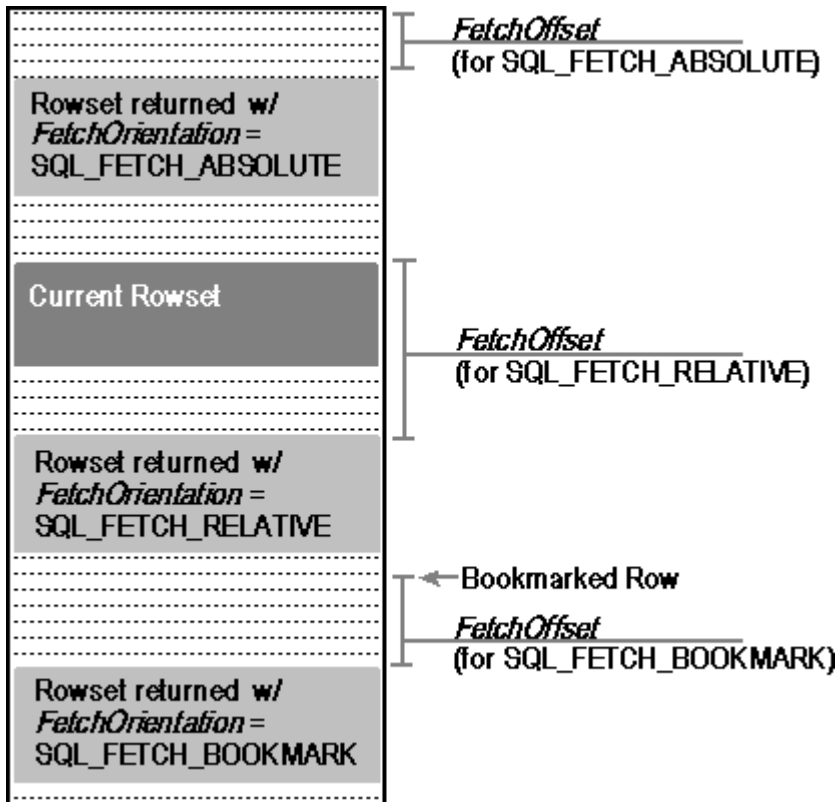
| Application sets attribute to  | Other attributes set implicitly   |
|--|---|
| SQL_ATTR_CONCURRENCY to SQL_CONCUR_READ_ONLY                                     | SQL_ATTR_CURSOR_SENSITIVITY to SQL_INSENSITIVE  |
| SQL_ATTR_CONCURRENCY to SQL_CONCUR_LOCK, SQL_CONCUR_ROWVER, or SQL_CONCUR_VALUES | SQL_ATTR_CURSOR_SENSITIVITY to SQL_UNSPECIFIED or SQL_SENSITIVE, as defined by the driver. It can never be set to SQL_INSENSITIVE, because insensitive cursors are always read-only.  |
| SQL_ATTR_CURSOR_SCROLLABLE to SQL_NONSCROLLABLE                                  | SQL_ATTR_CURSOR_TYPE to SQL_CURSOR_FORWARD_ONLY   |
| SQL_ATTR_CURSOR_SCROLLABLE to SQL_SCROLLABLE                                     | SQL_ATTR_CURSOR_TYPE to either SQL_CURSOR_STATIC, SQL_CURSOR_KEYSET_DRIVEN, or SQL_CURSOR_DYNAMIC, as specified by the driver. It is never set to SQL_CURSOR_FORWARD_ONLY.  |
| SQL_ATTR_CURSOR_SENSITIVITY to SQL_INSENSITIVE                                   | SQL_ATTR_CONCURRENCY to SQL_CONCUR_READ_ONLY<br>SQL_ATTR_CURSOR_TYPE to SQL_CURSOR_STATIC   |
| SQL_ATTR_CURSOR_SENSITIVITY to SQL_SENSITIVE                                     | SQL_ATTR_CONCURRENCY to SQL_CONCUR_LOCK, SQL_CONCUR_ROWVER, or SQL_CONCUR_VALUES, as specified by the driver. It is never set to SQL_CONCUR_READ_ONLY<br>SQL_ATTR_CURSOR_TYPE to SQL_CURSOR_FORWARD_ONLY, SQL_CURSOR_STATIC, SQL_CURSOR_KEYSET_DRIVEN, or |

|  |   |
|--|---|
|  | SQL_CURSOR_DYNAMIC, as specified by the driver  |
| SQL_ATTR_CURSOR_SENSITIVITY to SQL_UNSPECIFIED   | SQL_ATTR_CONCURRENCY to SQL_CONCUR_READ_ONLY, SQL_CONCUR_LOCK, SQL_CONCUR_ROWVER, or SQL_CONCUR_VALUES, as specified by the driver<br><br>SQL_ATTR_CURSOR_TYPE to SQL_CURSOR_FORWARD_ONLY, SQL_CURSOR_STATIC, SQL_CURSOR_KEYSET_DRIVEN, or SQL_CURSOR_DYNAMIC, as specified by the driver |
| SQL_ATTR_CURSOR_TYPE to SQL_CURSOR_DYNAMIC       | SQL_ATTR_SCROLLABLE to SQL_SCROLLABLE<br><br>SQL_ATTR_CURSOR_SENSITIVITY to SQL_SENSITIVE (but only if SQL_ATTR_CONCURRENCY is not equal to SQL_CONCUR_READ_ONLY. Updatable dynamic cursors are always sensitive to changes made in their own transaction.)                               |
| SQL_ATTR_CURSOR_TYPE to SQL_CURSOR_FORWARD_ONLY  | SQL_ATTR_CURSOR_SCROLLABLE to SQL_NONSCROLLABLE   |
| SQL_ATTR_CURSOR_TYPE to SQL_CURSOR_KEYSET_DRIVEN | SQL_ATTR_SCROLLABLE to SQL_SCROLLABLE<br><br>SQL_ATTR_SENSITIVITY to SQL_UNSPECIFIED or SQL_SENSITIVE (according to driver-defined criteria, if SQL_ATTR_CONCURRENCY is not SQL_CONCUR_READ_ONLY)   |
| SQL_ATTR_CURSOR_TYPE to SQL_CURSOR_STATIC        | SQL_ATTR_SCROLLABLE to SQL_SCROLLABLE<br><br>SQL_ATTR_SENSITIVITY to SQL_INSENSITIVE (if SQL_ATTR_CONCURRENCY is SQL_CONCUR_READ_ONLY)<br><br>SQL_ATTR_SENSITIVITY to SQL_UNSPECIFIED or SQL_SENSITIVE (if SQL_ATTR_CONCURRENCY is not SQL_CONCUR_READ_ONLY)                              |

## Scrolling and Fetching Rows

When using a scrollable cursor, applications call **SQLFetchScroll** to position the cursor and fetch rows. **SQLFetchScroll** supports relative scrolling (next, prior, and relative *n* rows), absolute scrolling (first, last, and row *n*), and positioning by bookmark. The *FetchOrientation* and *FetchOffset* arguments in **SQLFetchScroll** specify which rowset to fetch, as shown in the following diagrams.

### Fetching next, prior, first, and last rowsets



### Fetching absolute, relative, and bookmarked rowsets

**SQLFetchScroll** positions the cursor to the specified row and returns the rows in the rowset starting with that row. If the specified rowset overlaps the end of the result set, a partial rowset is returned. If the specified rowset overlaps the start of the result set, the first rowset in the result set is usually returned; for complete details, see the **SQLFetchScroll** function description in the Part II PDF file, "ODBC API Reference," available on the Solid Web site.

In some cases, the application may want to position the cursor without retrieving any data. For example, it might want to test whether a row exists or just get the bookmark for the row without bringing other data across the network. To do this, it sets the **SQL\_ATTR\_RETRIEVE\_DATA** statement attribute to



SQL\_RD\_OFF. Note that the variable bound to the bookmark column (if any) is always updated, regardless of the setting of this statement attribute.

After the rowset has been retrieved, the application can call **SQLSetPos** to position to a particular row in the rowset or refresh rows in the rowset. For more information on using **SQLSetPos**, see “[Updating Data with SQLSetPos](#)” in Chapter 12, “Updating Data.”

**Note** Scrolling is supported in ODBC 2.x drivers by **SQLExtendedFetch**. For more information, see “Block Cursors, Scrollable Cursors, and Backward Compatibility” in Appendix G, “Driver Guidelines for Backward Compatibility” contained on the Microsoft Web site (ODBC Programmer’s Reference).

## Relative and Absolute Scrolling

Most of the scrolling options in **SQLFetchScroll** position the cursor relative to the current position or to an absolute position. **SQLFetchScroll** supports fetching the next, prior, first, and last rowsets, as well as relative fetching (fetch the rowset *n* rows from the start of the current rowset) and absolute fetching (fetch the rowset starting at row *n*). If *n* is negative in an absolute fetch, rows are counted from the end of the result set. Thus, an absolute fetch of row – 1 means to fetch the rowset that starts with the last row in the result set.

Dynamic cursors detect rows inserted into and deleted from the result set, so there is no easy way for dynamic cursors to retrieve the row at a particular number other than reading from the start of the result set, which is likely to be slow. Furthermore, absolute fetching is not very useful in dynamic cursors because row numbers change as rows are inserted and deleted; thus, successively fetching the same row number can yield different rows.

Applications that use **SQLFetchScroll** only for its block cursor capabilities, such as reports, are likely to pass through the result set a single time, using only the option to fetch the next rowset. Screen-based applications, on the other hand, can take advantage of all the capabilities of **SQLFetchScroll**. If the application sets the rowset size to the number of rows displayed on the screen and binds the screen buffers to the result set, it can translate scroll bar operations directly to calls to **SQLFetchScroll**:

| Scroll bar operation       | SQLFetchScroll scrolling option                         |
|----------------------------|---|
| Page up                    | SQL_FETCH_PRIOR   |
| Page down                  | SQL_FETCH_NEXT  |
| Line up                    | SQL_FETCH_RELATIVE with <i>FetchOffset</i> equal to – 1 |
| Line down                  | SQL_FETCH_RELATIVE with <i>FetchOffset</i> equal to 1   |
| Scroll box at top          | SQL_FETCH_FIRST   |
| Scroll box at bottom       | SQL_FETCH_LAST  |
| Random scroll box position | SQL_FETCH_ABSOLUTE                                      |

Such applications also need to position the scroll box after a scrolling operation, which requires the current row number and the number of rows. For the current row number, applications can either keep track of the current row number or call **SQLGetStmtAttr** with the SQL\_ATTR\_ROW\_NUMBER attribute to retrieve it.

The number of rows in the cursor, which is the size of the result set, is available as the `SQL_DIAG_CURSOR_ROW_COUNT` field of the diagnostic header. The value in this field is defined only after **SQLExecute**, **SQLExecDirect**, or **SQLMoreResult** has been called. This count can be either an approximate count or an exact count, depending on the capabilities of the driver. The driver's support can be determined by calling **SQLGetInfo** with the cursor attributes information types, and checking whether the `SQL_CA2_CRC_APPROXIMATE` or `SQL_CA2_CRC_EXACT` bit is returned for the type of cursor.

An exact row count is never supported for a dynamic cursor. For other types of cursors, the driver can either support exact or approximate row counts, but not both. If the driver supports neither exact nor approximate row counts for a specific cursor type, then the `SQL_DIAG_CURSOR_ROW_COUNT` field contains the number of rows that have been fetched so far. Regardless of what the driver supports, **SQLFetchScroll** with an *Operation* of `SQL_FETCH_LAST` will cause the `SQL_DIAG_CURSOR_ROW_COUNT` field to contain the exact row count.

## Bookmarks

A bookmark is a value used to identify a row of data. The meaning of the bookmark value is known only to the driver or data source. For example, it might be as simple as a row number or as complex as a disk address. Bookmarks in ODBC are a bit different from bookmarks in real books. In a real book, the reader places a bookmark at a specific page, then looks for that bookmark to return to the page. In ODBC, the application requests a bookmark for a particular row, stores it, and passes it back to the cursor to return to the row. Thus, bookmarks in ODBC are similar to a reader writing down a page number, remembering it, and then looking up the page again.

To determine a driver's support of bookmarks, an application calls **SQLGetInfo** with the `SQL_BOOKMARK_PERSISTENCE` option. The bits in this value describe what operations bookmarks survive, such as whether bookmarks are still valid after the cursor is closed.

### Bookmark Types

All bookmarks in ODBC 3.x are variable-length bookmarks. This allows a primary key or a unique index associated with a table to be used as a bookmark. The bookmark also can be a 32-bit value, as was used in ODBC 2.x. To specify that a bookmark is used with a cursor, an ODBC 3.x application sets the `SQL_ATTR_USE_BOOKMARK` statement attribute to `SQL_UB_VARIABLE`. A variable-length bookmark is automatically used.

An application can call **SQLColAttribute** with the `FieldIdentifier` argument set to `SQL_DESC_OCTET_LENGTH` to obtain the length of the bookmark. Because a variable-length bookmark can be a long value, an application should not bind to column 0 unless it will use the bookmark for many of the rows in the rowset.

Fixed-length bookmarks are supported only for backward compatibility. If an ODBC 2.x application working with an ODBC 3.x driver calls **SQLSetStmtOption** to set `SQL_USE_BOOKMARKS` to `SQL_UB_ON`, it is mapped in the Driver Manager to `SQL_UB_VARIABLE`. A variable-length bookmark is used, even if only 32 bits of it are populated. If a driver supports fixed-length bookmarks, it will support variable-length bookmarks. If an ODBC 3.x application working with an ODBC 2.x driver calls **SQLSetStmtAttr** to set `SQL_ATTR_USE_BOOKMARKS` to `SQL_UB_VARIABLE`, it is mapped in the Driver Manager to `SQL_UB_ON`, and a 32-bit fixed-length bookmark is used. The `SQL_ATTR_FETCH_BOOKMARK_PTR` statement attribute must then point to a 32-bit bookmark. If the bookmarks used are longer than 32-bits, such as when primary keys are used as bookmarks, the cursor must

map the actual values to 32-bit values. It could, for example, build a hash table of them. When an ODBC 3.x application working with an ODBC 2.x driver binds a bookmark, the buffer length must be 4.

### ***Retrieving Bookmarks***

If the application will use bookmarks, it must set the `SQL_ATTR_USE_BOOKMARKS` statement attribute to `SQL_UB_VARIABLE` before preparing or executing the statement. This is necessary because building and maintaining bookmarks can be an expensive operation, so bookmarks should be enabled only when an application can make good use of them.

Bookmarks are returned as column 0 of the result set. There are three ways an application can retrieve them:

- Bind column 0 of the result set. **SQLFetch** or **SQLFetchScroll** returns the bookmarks for each row in the rowset along with the data for other bound columns.
- Call **SQLSetPos** to position to a row in the rowset, then call **SQLGetData** for column 0. Note that if a driver supports bookmarks, it must always support the ability to call **SQLGetData** for column 0, even if it does not allow applications to call **SQLGetData** for other columns before the last bound column.
- Call **SQLBulkOperations** with the *Operation* argument set to `SQL_ADD`, and column 0 bound. The cursor inserts the row and returns the bookmark for the row in the bound buffer.

### ***Scrolling by Bookmark***

When fetching rows with **SQLFetchScroll**, an application can use a bookmark as a basis for selecting the starting row. This is a form of absolute addressing because it does not depend on the current cursor position. To scroll to a bookmarked row, the application calls **SQLFetchScroll** with a *FetchOrientation* of `SQL_FETCH_BOOKMARK`. This operation uses the bookmark pointed to by the `SQL_ATTR_FETCH_BOOKMARK_PTR` statement attribute. It returns the rowset starting with the row identified by that bookmark. An application can specify an offset for this operation in the *FetchOffset* argument of the call to **SQLFetchScroll**. When an offset is specified, the first row of the returned rowset is determined by adding the number in the *FetchOffset* argument to the number of the row identified by the bookmark. This use of the *FetchOffset* argument is not supported when used with ODBC 2.x drivers; when an application calls **SQLFetchScroll** in an ODBC 2.x driver with *FetchOrientation* set to `SQL_FETCH_BOOKMARK`, the *FetchOffset* argument must be set to 0.

### ***Updating, Deleting, or Fetching by Bookmark***

Bookmarks can be used to identify data to be updated in the result set, deleted from the result set, or fetched from the result set to the rowset buffers. These operations are performed by a call to **SQLBulkOperations** with an *Option* argument of `SQL_UPDATE_BY_BOOKMARK`, `SQL_DELETE_BY_BOOKMARK`, or `SQL_FETCH_BY_BOOKMARK`. The bookmarks used in these operations are stored in column 0 of the rowset buffers. When updating by bookmark, the data that result set columns are updated to is retrieved from the rowset buffers. For more information, see [“Updating Data with SQLBulkOperations”](#) in Chapter 12, “Updating Data.”

## Comparing Bookmarks

Because bookmarks are byte-comparable, they can be compared for equality or inequality. To do so, an application treats each bookmark as an array of bytes, and compares two bookmarks byte-by-byte. Because bookmarks are guaranteed to be distinct only within a result set, it makes no sense to compare bookmarks that were obtained from different result sets.

## The ODBC Cursor Library

Block and scrollable cursors are very useful additions to many applications. However, not all drivers support block and scrollable cursors. The same is true of positioned update and delete statements and **SQLSetPos**, which are discussed in “[Chapter 12: Overview of Updating Data](#).”

Because of this, the ODBC component of the Data Access SDK includes a cursor library. The cursor library implements block, static cursors, positioned update and delete statements, and **SQLSetPos** for any driver that meets the X/Open Standard CLI conformance level. The cursor library may be redistributed with ODBC applications; see the licensing agreement in the SDK for details.

To use the cursor library, an application sets the **SQL\_ATTR\_ODBC\_CURSORS** connection attribute before connecting to the data source. For more information about the cursor library, see “ODBC Cursor Library” contained on the Microsoft Web site (ODBC Programmer’s Reference).

## Multiple Results

A *result* is something returned by the data source after a statement is executed. ODBC has two types of results: result sets and row counts. Row counts are the number of rows affected by an update, delete, or insert statement. Batches—described in “[Batches of SQL Statements](#)” in Chapter 9, “Executing Statements”—can generate multiple results.

The following table lists the **SQLGetInfo** options an application uses to determine whether a data source returns multiple results for each different type of batch. In particular, a data source can return a single row count for the entire batch of statements or individual row counts for each statement in the batch. In the case of a result set – generating statement executed with an array of parameters, the data source can return a single result set for all sets of parameters or individual result sets for each set of parameters.

| Batch type           | Row counts                  | Result sets              |
|----------------------|-----------------------------|--------------------------|
| Explicit batch       | SQL_BATCH_ROW_COUNT [a]     | -- [b]                   |
| Procedure            | SQL_BATCH_ROW_COUNT [a]     | -- [b]                   |
| Arrays of parameters | SQL_PARAM_ARRAYS_ROW_COUNTS | SQL_PARAM_ARRAYS_SELECTS |

[a]Row count – generating statements in a batch may be supported, yet the return of the row counts not supported. The **SQL\_BATCH\_SUPPORT** option in **SQLGetInfo** indicates whether row count – generating statements are allowed in batches; the **SQL\_BATCH\_ROW\_COUNTS** option indicates whether these row counts are returned to the application.

[b]Explicit batches and procedures always return multiple result sets when they include multiple result set – generating statements.

**Note** The SQL\_MULT\_RESULT\_SETS option introduced in ODBC 1.0 provides only general information about whether multiple result sets can be returned. In particular, it is set to “Y” if the SQL\_BS\_SELECT\_EXPLICIT or SQL\_BS\_SELECT\_PROC bits are returned for SQL\_BATCH\_SUPPORT or if SQL\_PAS\_BATCH is returned for SQL\_PARAM\_ARRAYS\_SELECT.

To process multiple results, an application calls **SQLMoreResults**. This function discards the current result and makes the next result available. It returns SQL\_NO\_DATA when no more results are available. For example, suppose the following statements are executed as a batch:

```
SELECT * FROM Parts WHERE Price > 100.00;  
UPDATE Parts SET Price = 0.9 * Price WHERE Price > 100.00
```

After these statements are executed, the application fetches rows from the result set created by the **SELECT** statement. When it is done fetching rows, it calls **SQLMoreResults** to make available the number of parts that were repriced. If necessary, **SQLMoreResults** discards unfetched rows and closes the cursor. The application then calls **SQLRowCount** to determine how many parts were repriced by the **UPDATE** statement.

It is driver-specific whether the entire batch statement is executed before any results are available. In some implementations, this is the case; in others, calling **SQLMoreResults** triggers the execution of the next statement in the batch.

If one of the statements in a batch fails, **SQLMoreResults** will return either SQL\_ERROR or SQL\_SUCCESS\_WITH\_INFO. If the batch was aborted when the statement failed, or the failed statement was the last statement in the batch, **SQLMoreResults** will return SQL\_ERROR. If the batch was not aborted when the statement failed, and the failed statement was not the last statement in the batch, **SQLMoreResults** will return SQL\_SUCCESS\_WITH\_INFO. SQL\_SUCCESS\_WITH\_INFO indicates that at least one result set or count was generated, and that the batch was not aborted.

## Chapter 12: Overview of Updating Data

Applications can update data either by executing SQL statements or by calling **SQLSetPos** or **SQLBulkOperations**. **UPDATE**, **DELETE**, and **INSERT** statements act directly on the data source and are usually supported by drivers. Searched update and delete statements contain a specification of the rows to change. Positioned update and delete statements and **SQLSetPos** act on the data source through a cursor and are less widely supported.

Whether cursors can detect changes made to the result set with the methods described in this chapter depends on the type of the cursor and how it is implemented. Forward-only cursors do not revisit rows and therefore will not detect any changes. For information about whether scrollable cursors can detect changes, see “[Scrollable Cursors](#)” in Chapter 11, “Retrieving Results (Advanced).”

### UPDATE, DELETE, and INSERT Statements

SQL-based applications make changes to tables by executing the **UPDATE**, **DELETE**, and **INSERT** statements. These statements are part of the Minimum SQL grammar conformance level and must be supported by all drivers and data sources.

The syntax of these statements is:

```
UPDATE table-name  
SET column-identifier = {expression | NULL}  
[, column-identifier = {expression | NULL}]...  
[WHERE search-condition]
```

```
DELETE FROM table-name [WHERE search-condition]
```

```
INSERT INTO table-name [(column-identifier [, column-identifier]...)]  
{query-specification | VALUES (insert-value [, insert-value]...)}
```

Note that the *query-specification* element is valid only in the Core and Extended SQL grammars, and that the *expression* and *search-condition* elements become more complex in the Core and Extended SQL grammars.

Like other SQL statements, **UPDATE**, **DELETE**, and **INSERT** statements are often more efficient when they use parameters. For example, the following statement can be prepared and repeatedly executed to insert multiple rows in the Orders table:

```
INSERT INTO Orders (PartID, Description, Price) VALUES (?, ?, ?)
```

This efficiency can be increased by passing arrays of parameter values. For more information about statement parameters and arrays of parameter values, see “Statement Parameters” in Chapter 9, “Executing Statements.”

### Positioned Update and Delete Statements

Applications can update or delete the current row in a result set with a positioned update or delete statement. Positioned update and delete statements are supported by some data sources, but not all of them.

To determine whether a data source supports positioned update and delete statements, an application calls **SQLGetInfo** with the **SQL\_DYNAMIC\_CURSOR\_ATTRIBUTES1**, **SQL\_FORWARD\_ONLY\_CURSOR\_ATTRIBUTES1**, **SQL\_KEYSET\_CURSOR\_ATTRIBUTES1**, or **SQL\_STATIC\_CURSOR\_ATTRIBUTES1** *InfoType* (depending on the type of the cursor). Note that the ODBC cursor library simulates positioned update and delete statements.

To use a positioned update or delete statement, the application must create a result set with a **SELECT FOR UPDATE** statement. The syntax of this statement is:

```
SELECT [ALL | DISTINCT] select-list
FROM table-reference-list
[WHERE search-condition]
FOR UPDATE OF [column-name [, column-name]...]
```

The application then positions the cursor on the row to be updated or deleted. It can do this by calling **SQLFetchScroll** to retrieve a rowset containing the required row and calling **SQLSetPos** to position the rowset cursor on that row. The application then executes the positioned update or delete statement on a different statement than the statement being used by the result set. The syntax of these statements is:

```
UPDATE table-name
SET column-identifier = {expression | NULL}
[, column-identifier = {expression | NULL}]...
WHERE CURRENT OF cursor-name
```

```
DELETE FROM table-name WHERE CURRENT OF cursor-name
```

Notice that these statements require a cursor name. The application either can specify a cursor name with **SQLSetCursorName** before executing the statement that creates the result set or can let the data source automatically generate a cursor name when the cursor is created. In the latter case, the application retrieves this cursor name for use in positioned update and delete statements by calling **SQLGetCursorName**.

For example, the following code allows a user to scroll through the Customers table and delete customer records or update their addresses and phone numbers. It calls **SQLSetCursorName** to specify a cursor name before it creates the result set of customers and uses three statement handles: *hstmtCust* for the result set, *hstmtUpdate* for a positioned update statement, and *hstmtDelete* for a positioned delete statement. Although the code could bind separate variables to the parameters in the positioned update statement, it updates the rowset buffers and binds the elements of these buffers. This keeps the rowset buffers synchronized with the updated data.

```
#define POSITIONED_UPDATE 100
#define POSITIONED_DELETE 101

SQLINTEGER      CustIDArray[10];
SQLCHAR         NameArray[10][51], AddressArray[10][51],
                PhoneArray[10][11];
SQLINTEGER      CustIDIndArray[10], NameLenOrIndArray[10],
                AddressLenOrIndArray[10],
                PhoneLenOrIndArray[10];
SQLUSMALLINT    RowStatusArray[10], Action, RowNum;
SQLHSTMT        hstmtCust, hstmtUpdate, hstmtDelete;
```

```

// Set the SQL_ATTR_BIND_TYPE statement attribute to use column-wise
// binding. Declare the rowset size with the SQL_ATTR_ROW_ARRAY_SIZE
// statement attribute. Set the SQL_ATTR_ROW_STATUS_PTR statement
// attribute to point to the row status array.
SQLSetStmtAttr(hstmtCust, SQL_ATTR_ROW_BIND_TYPE, SQL_BIND_BY_COLUMN, 0);
SQLSetStmtAttr(hstmtCust, SQL_ATTR_ROW_ARRAY_SIZE, 10, 0);
SQLSetStmtAttr(hstmtCust, SQL_ATTR_ROW_STATUS_PTR, RowStatusArray, 0);

// Bind arrays to the CustID, Name, Address, and Phone columns.
SQLBindCol(hstmtCust, 1, SQL_C_ULONG, CustIDArray, 0, CustIDIndArray);
SQLBindCol(hstmtCust, 2, SQL_C_CHAR, NameArray, sizeof(NameArray[0]),
    NameLenOrIndArray);
SQLBindCol(hstmtCust, 3, SQL_C_CHAR, AddressArray, sizeof(AddressArray[0]),
    AddressLenOrIndArray);
SQLBindCol(hstmtCust, 4, SQL_C_CHAR, PhoneArray, sizeof(PhoneArray[0]),
    PhoneLenOrIndArray);

// Set the cursor name to Cust.
SQLSetCursorName(hstmtCust, "Cust", SQL_NTS);

// Prepare positioned update and delete statements.
SQLPrepare(hstmtUpdate,
    "UPDATE Customers SET Address = ?, Phone = ? WHERE CURRENT OF Cust",
    SQL_NTS);
SQLPrepare(hstmtDelete, "DELETE FROM Customers WHERE CURRENT OF Cust", SQL_NTS);

// Execute a statement to retrieve rows from the Customers table.
SQLExecDirect(hstmtCust,
    "SELECT CustID, Name, Address, Phone FROM Customers FOR UPDATE OF Address,
    Phone",
    SQL_NTS);

// Fetch and display the first 10 rows.
SQLFetchScroll(hstmtCust, SQL_FETCH_NEXT, 0);
DisplayData(CustIDArray, CustIDIndArray, NameArray, NameLenOrIndArray,
    AddressArray,
    AddressLenOrIndArray, PhoneArray, PhoneLenOrIndArray,
    RowStatusArray);

// Call GetAction to get an action and a row number from the user.
while (GetAction(&Action, &RowNum)) {
    switch (Action) {

        case SQL_FETCH_NEXT:
        case SQL_FETCH_PRIOR:
        case SQL_FETCH_FIRST:
        case SQL_FETCH_LAST:
        case SQL_FETCH_ABSOLUTE:
        case SQL_FETCH_RELATIVE:
            // Fetch and display the requested data.
            SQLFetchScroll(hstmtCust, Action, RowNum);
            DisplayData(CustIDArray, CustIDIndArray, NameArray, NameLenOrIndArray,
                AddressArray, AddressLenOrIndArray, PhoneArray,
                PhoneLenOrIndArray, RowStatusArray);

```



```

        break;

    case POSITIONED_UPDATE:
        // Get the new data and place it in the rowset buffers.
        GetNewData(AddressArray[RowNum - 1], &AddressLenOrIndArray[RowNum - 1],
                   PhoneArray[RowNum - 1], &PhoneLenOrIndArray[RowNum - 1]);

        // Bind the elements of the arrays at position RowNum-1 to the
        // parameters of the positioned update statement.
        SQLBindParameter(hstmtUpdate, 1, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR,
                          50, 0, AddressArray[RowNum - 1],
sizeof(AddressArray[0]),
                          &AddressLenOrIndArray[RowNum - 1]);
        SQLBindParameter(hstmtUpdate, 2, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR,
                          10, 0, PhoneArray[RowNum - 1], sizeof(PhoneArray[0]),
                          &PhoneLenOrIndArray[RowNum - 1]);

        // Position the rowset cursor. The rowset is 1-based.
        SQLSetPos(hstmtCust, RowNum, SQL_POSITION, SQL_LOCK_NO_CHANGE);

        // Execute the positioned update statement to update the row.
        SQLExecute(hstmtUpdate);
        break;

    case POSITIONED_DELETE:
        // Position the rowset cursor. The rowset is 1-based.
        SQLSetPos(hstmtCust, RowNum, SQL_POSITION, SQL_LOCK_NO_CHANGE);

        // Execute the positioned delete statement to delete the row.
        SQLExecute(hstmtDelete);
        break;
    }
}

// Close the cursor.
SQLCloseCursor(hstmtCust);

```

## Simulating Positioned Update and Delete Statements

If the data source does not support positioned update and delete statements, the driver can simulate these. For example, the ODBC cursor library simulates positioned update and delete statements. The general strategy for simulating positioned update and delete statements is to convert positioned statements to searched ones. This is done by replacing the **WHERE CURRENT OF** clause with a searched **WHERE** clause that identifies the current row.

For example, because the CustID column uniquely identifies each row in the Customers table, the positioned delete statement:

```
DELETE FROM Customers WHERE CURRENT OF CustCursor
```

might be converted to:

```
DELETE FROM Customers WHERE (CustID = ?)
```

The driver may use one of the following *row identifiers* in the **WHERE** clause:

- Columns whose values serve to identify uniquely every row in the table. For example, calling **SQLSpecialColumns** with **SQL\_BEST\_ROWID** returns the optimal column or set of columns that serve this purpose.
- Pseudo-columns, provided by some data sources, for the purpose of uniquely identifying every row. These may also be retrievable by calling **SQLSpecialColumns**.
- A unique index, if available.
- All the columns in the result set.

Exactly which columns a driver should use in the **WHERE** clause it constructs depends on the driver. On some data sources, determining a row identifier can be costly. However, it is faster to execute and guarantees that a simulated statement updates or deletes at most one row. Depending on the capabilities of the underlying DBMS, using a row identifier can be expensive to set up. However, it is faster to execute and guarantees that a simulated statement will update or delete only one row. Using all the columns in the result set is usually much easier to set up. However, it is slower to execute and, if the columns do not uniquely identify a row, can result in rows being unintentionally updated or deleted, especially when the select list for the result set does not contain all the columns that exist in the underlying table.

Depending upon which of these strategies the driver supports, an application can choose which strategy it wants the driver to use with the **SQL\_ATTR\_SIMULATE\_CURSOR** statement attribute. Although it might seem odd for an application to risk unintentionally updating or deleting a row, the application can remove this risk by ensuring that the columns in the result set uniquely identify each row in the result set. This saves the driver the effort of having to do this.

If the driver chooses to use a row identifier, it intercepts the **SELECT FOR UPDATE** statement that creates the result set. If the columns in the select list do not effectively identify a row, the driver adds the necessary columns to the end of the select list. Some data sources have a single column that always uniquely identifies a row, such as the **ROWID** column in Oracle; if so, the driver uses this. Otherwise, the driver calls **SQLSpecialColumns** for each table in the **FROM** clause to retrieve a list of the columns that uniquely identify each row. A common restriction that results from this technique is that cursor simulation fails if there is more than one table in the **FROM** clause.

No matter how the driver identifies rows, it usually strips the **FOR UPDATE OF** clause off the **SELECT FOR UPDATE** statement before sending it to the data source. The **FOR UPDATE OF** clause is used only with positioned update and delete statements. Data sources that do not support positioned update and delete statements generally do not support it.

When the application submits a positioned update or delete statement for execution, the driver replaces the **WHERE CURRENT OF** clause with a **WHERE** clause containing the row identifier. The values of these columns are retrieved from a cache maintained by the driver for each column it uses in the **WHERE** clause. After the driver has replaced the **WHERE** clause, it sends the statement to the data source for execution.

For example, suppose that the application submits the following statement to create a result set:

```
SELECT Name, Address, Phone FROM Customers FOR UPDATE OF Phone, Address
```

If the application has set `SQL_ATTR_SIMULATE_CURSOR` to request a guarantee of uniqueness and if the data source does not provide a pseudo-column that always uniquely identifies a row, the driver calls **SQLSpecialColumns** for the Customers table, discovers that `CustID` is the key to the Customers table and adds this to the select list, and strips the **FOR UPDATE OF** clause:

```
SELECT Name, Address, Phone, CustID FROM Customers
```

If the application has not requested a guarantee of uniqueness, the driver strips only the **FOR UPDATE OF** clause:

```
SELECT Name, Address, Phone FROM Customers
```

Suppose the application scrolls through the result set and submits the following positioned update statement for execution, where `Cust` is the name of the cursor over the result set:

```
UPDATE Customers SET Address = ?, Phone = ? WHERE CURRENT OF Cust
```

If the application has not requested a guarantee of uniqueness, the driver replaces the **WHERE** clause and binds the `CustID` parameter to the variable in its cache:

```
UPDATE Customers SET Address = ?, Phone = ? WHERE (CustID = ?)
```

If the application has not requested a guarantee of uniqueness, the driver replaces the **WHERE** clause and binds the `Name`, `Address`, and `Phone` parameters in this clause to the variables in its cache:

```
UPDATE Customers SET Address = ?, Phone = ?  
WHERE (Name = ?) AND (Address = ?) AND (Phone = ?)
```

## Determining the Number of Affected Rows

After an application updates, deletes, or inserts rows, it can call **SQLRowCount** to determine how many rows were affected. **SQLRowCount** returns this value regardless of whether the rows were updated, deleted, or inserted by executing an **UPDATE**, **DELETE**, or **INSERT** statement, by executing a positioned update or delete statement, or by calling **SQLSetPos**.

If a batch of SQL statements is executed, the count of affected rows might be a total count for all statements in the batch or individual counts for each statement in the batch. For more information, see [“Batches of SQL Statements”](#) in Chapter 9, “Executing Statements,” and [“Multiple Results”](#) in Chapter 11, “Retrieving Results (Advanced).”

The number of affected rows is also returned in the `SQL_DIAG_ROW_COUNT` diagnostic header field in the diagnostic area associated with the statement handle. However, the data in this field is reset after every function call on the same statement handle, whereas the value returned by **SQLRowCount** remains the same until a call to **SQLBulkOperations**, **SQLExecute**, **SQLExecDirect**, **SQLPrepare**, or **SQLSetPos**.

## Updating Data with SQLSetPos

Applications can update or delete any row in the rowset with **SQLSetPos**. Calling **SQLSetPos** is a convenient alternative to constructing and executing an SQL statement. It lets an ODBC driver support

positioned updates even when the data source does not support positioned SQL statements. It is part of the paradigm of achieving complete database access by means of function calls.

**SQLSetPos** operates on the current rowset and can be used only after a call to **SQLFetchScroll**. The application specifies the number of the row to update, delete, or insert, and the driver retrieves the new data for that row from the rowset buffers. **SQLSetPos** can also be used to designate a specified row as the current row, or to refresh a particular row in the rowset from the data source.

Rowset size is set by a call to **SQLSetStmtAttr** with an *Attribute* argument of **SQL\_ATTR\_ROW\_ARRAY\_SIZE**. **SQLSetPos** uses a new rowset size, however, only after a call to **SQLFetch** or **SQLFetchScroll**. For example, if the rowset size is changed, then **SQLSetPos** is called, then **SQLFetch** or **SQLFetchScroll** is called, the call to **SQLSetPos** uses the old rowset size, but **SQLFetch** or **SQLFetchScroll** uses the new rowset size.

The first row in the rowset is row number 1. The *RowNumber* argument in **SQLSetPos** must identify a row in the rowset; that is, its value must be in the range between 1 and the number of rows that were most recently fetched (which may be less than the rowset size). If *RowNumber* is 0, the operation applies to every row in the rowset.

Because most interaction with relational databases is done through SQL, **SQLSetPos** is not widely supported. However, a driver can easily emulate it by constructing and executing an **UPDATE** or **DELETE** statement.

To determine what operations **SQLSetPos** supports, an application calls **SQLGetInfo** with the **SQL\_DYNAMIC\_CURSOR\_ATTRIBUTES1**, **SQL\_FORWARD\_ONLY\_CURSOR\_ATTRIBUTES1**, **SQL\_KEYSET\_CURSOR\_ATTRIBUTES1**, or **SQL\_STATIC\_CURSOR\_ATTRIBUTES1** information option (depending on the type of the cursor).

## Updating Rows in the Rowset with SQLSetPos

The update operation of **SQLSetPos** makes the data source update one or more selected rows of a table, using data in the application buffers for each bound column (unless the value in the length/indicator buffer is **SQL\_COLUMN\_IGNORE**). Columns that are not bound will not be updated.

To update rows with **SQLSetPos**, the application does the following:

1. Places the new data values in the rowset buffers. For information on how to send long data with **SQLSetPos**, see “Long Data and SQLSetPos and SQLBulkOperations,” later in this chapter.
2. Sets the value in the length/indicator buffer of each column as necessary. This is the byte length of the data or **SQL\_NTS** for columns bound to string buffers, the byte length of the data for columns bound to binary buffers, and **SQL\_NULL\_DATA** for any columns to be set to NULL.
3. Sets the value in the length/indicator buffer of those columns which are not to be updated to **SQL\_COLUMN\_IGNORE**. Although the application can skip this step and resend existing data, this is inefficient and risks sending values to the data source that were truncated when they were read.

4. Calls **SQLSetPos** with *Operation* set to `SQL_UPDATE` and *RowNumber* set to the number of the row to update. If *RowNumber* is 0, all rows in the rowset are updated.

After **SQLSetPos** returns, the current row is set to the updated row.

When updating all rows of the rowset (*RowNumber* is equal to 0), an application can disable the update of certain rows by setting the corresponding elements of the row operation array (pointed to by the `SQL_ATTR_ROW_OPERATION_PTR` statement attribute) to `SQL_ROW_IGNORE`. The row operation array corresponds in size and number of elements to the row status array (pointed to by the `SQL_ATTR_ROW_STATUS_PTR` statement attribute). To update only those rows in the result set that were successfully fetched and have not been deleted from the rowset, the application uses the row status array from the function that fetched the rowset as the row operation array to **SQLSetPos**.

For every row that is sent to the data source as an update, the application buffers should have valid row data. If the application buffers were filled by fetching and if a row status array has been maintained, its values at each of these row positions should not be `SQL_ROW_DELETED`, `SQL_ROW_ERROR`, or `SQL_ROW_NOROW`.

For example, the following code allows a user to scroll through the Customers table and update, delete, or add new rows. It places the new data in the rowset buffers before calling **SQLSetPos** to update or add new rows. An extra row is allocated at the end of the rowset buffers to hold new rows; this prevents existing data from being overwritten when data for a new row is placed in the buffers.

```
#define UPDATE_ROW 100
#define DELETE_ROW 101
#define ADD_ROW 102

SQLINTEGER CustIDArray[11];
SQLCHAR NameArray[11][51], AddressArray[11][51], PhoneArray[11][11];
SQLINTEGER CustIDIndArray[11], NameLenOrIndArray[11], AddressLenOrIndArray[11],
        PhoneLenOrIndArray[11];
SQLUSMALLINT RowStatusArray[10], Action, RowNum;
SQLRETURN rc;
SQLHSTMT hstmt;

// Set the SQL_ATTR_ROW_BIND_TYPE statement attribute to use column-wise
binding.
// Declare the rowset size with the SQL_ATTR_ROW_ARRAY_SIZE statement attribute.
// Set the SQL_ATTR_ROW_STATUS_PTR statement attribute to point to the row
status
// array.
SQLSetStmtAttr(hstmt, SQL_ATTR_CURSOR_TYPE, SQL_CURSOR_KEYSET_DRIVEN, 0);
SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_BIND_TYPE, SQL_BIND_BY_COLUMN, 0);
SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_ARRAY_SIZE, 10, 0);
SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_STATUS_PTR, RowStatusArray, 0);

// Bind arrays to the CustID, Name, Address, and Phone columns.
SQLBindCol(hstmt, 1, SQL_C_ULONG, CustIDArray, 0, CustIDIndArray);
SQLBindCol(hstmt, 2, SQL_C_CHAR, NameArray, sizeof(NameArray[0]),
NameLenOrIndArray);
SQLBindCol(hstmt, 3, SQL_C_CHAR, AddressArray, sizeof(AddressArray[0]),
AddressLenOrIndArray);
```

```

SQLBindCol(hstmt, 4, SQL_C_CHAR, PhoneArray, sizeof(PhoneArray[0]),
    PhoneLenOrIndArray);

// Execute a statement to retrieve rows from the Customers table.
SQLExecDirect(hstmt, "SELECT CustID, Name, Address, Phone FROM Customers",
    SQL_NTS);

// Fetch and display the first 10 rows.
rc = SQLFetchScroll(hstmt, SQL_FETCH_NEXT, 0);
DisplayData(CustIDArray, CustIDIndArray, NameArray, NameLenOrIndArray,
    AddressArray,
        AddressLenOrIndArray, PhoneArray, PhoneLenOrIndArray, RowStatusArray);

// Call GetAction to get an action and a row number from the user.
while (GetAction(&Action, &RowNum)) {
    switch (Action) {

        case SQL_FETCH_NEXT:
        case SQL_FETCH_PRIOR:
        case SQL_FETCH_FIRST:
        case SQL_FETCH_LAST:
        case SQL_FETCH_ABSOLUTE:
        case SQL_FETCH_RELATIVE:
            // Fetch and display the requested data.
            SQLFetchScroll(hstmt, Action, RowNum);
            DisplayData(CustIDArray, CustIDIndArray,
                NameArray, NameLenOrIndArray,
                AddressArray, AddressLenOrIndArray,
                PhoneArray, PhoneLenOrIndArray, RowStatusArray);
            break;

        case UPDATE_ROW:
            // Place the new data in the rowset buffers and update the specified row.
            GetNewData(&CustIDArray[RowNum - 1], &CustIDIndArray[RowNum - 1],
                NameArray[RowNum - 1], &NameLenOrIndArray[RowNum - 1],
                AddressArray[RowNum - 1], &AddressLenOrIndArray[RowNum - 1],
                PhoneArray[RowNum - 1], &PhoneLenOrIndArray[RowNum - 1]);
            SQLSetPos(hstmt, RowNum, SQL_UPDATE, SQL_LOCK_NO_CHANGE);
            break;

        case DELETE_ROW:
            // Delete the specified row.
            SQLSetPos(hstmt, RowNum, SQL_DELETE, SQL_LOCK_NO_CHANGE);
            break;

        case ADD_ROW:
            // Place the new data in the rowset buffers at index 10. This is an extra
            // element for new rows so rowset data is not overwritten. Insert the new
            // row. Row 11 corresponds to index 10.
            GetNewData(&CustIDArray[10], &CustIDIndArray[10],
                NameArray[10], &NameLenOrIndArray[10],
                AddressArray[10], &AddressLenOrIndArray[10],
                PhoneArray[10], &PhoneLenOrIndArray[10]);
            SQLSetPos(hstmt, 11, SQL_ADD, SQL_LOCK_NO_CHANGE);

```

```

        break;
    }
}

// Close the cursor.
SQLCloseCursor(hstmt);

```

## Deleting Rows in the Rowset with SQLSetPos

The delete operation of **SQLSetPos** makes the data source delete one or more selected rows of a table. To delete rows with **SQLSetPos**, the application calls **SQLSetPos** with *Operation* set to **SQL\_DELETE** and *RowNumber* set to the number of the row to delete. If *RowNumber* is 0, all rows in the rowset are deleted.

After **SQLSetPos** returns, the deleted row is the current row, and its status is **SQL\_ROW\_DELETED**. The row cannot be used in any further positioned operations, such as calls to **SQLGetData** or **SQLSetPos**.

When deleting all rows of the rowset (*RowNumber* is equal to 0), the application can prevent the driver from deleting certain rows by using the row operation array, in the same way as for the update operation of **SQLSetPos** (see “Updating Rows in the Rowset with SQLSetPos” earlier in this chapter).

Every row that is deleted should be a row that exists in the result set. If the application buffers were filled by fetching and if a row status array has been maintained, its values at each of these row positions should not be **SQL\_ROW\_DELETED**, **SQL\_ROW\_ERROR**, or **SQL\_ROW\_NOROW**.

## Updating Data with SQLBulkOperations

Applications can perform bulk update, delete, fetch, or insertion operations on the underlying table at the data source with a call to **SQLBulkOperations**. Calling **SQLBulkOperations** is a convenient alternative to constructing and executing an SQL statement. It lets an ODBC driver support positioned updates even when the data source does not support positioned SQL statements. It is part of the paradigm of achieving complete database access by means of function calls.

**SQLBulkOperations** operates on the current rowset and can be used only after a call to **SQLFetch** or **SQLFetchScroll**. The application specifies the rows to update, delete, or refresh by caching their bookmarks. The driver retrieves the new data for rows to be updated, or the new data to be inserted into the underlying table, from the rowset buffers.

The rowset size to be used by **SQLBulkOperations** is set by a call to **SQLSetStmtAttr** with an *Attribute* argument of **SQL\_ATTR\_ROW\_ARRAY\_SIZE**. Unlike **SQLSetPos**, which uses a new rowset size only after a call to **SQLFetch** or **SQLFetchScroll**, **SQLBulkOperations** uses the new rowset size after the call to **SQLSetStmtAttr**.

Because most interaction with relational databases is done through SQL, **SQLBulkOperations** is not widely supported. However, a driver can easily emulate it by constructing and executing an **UPDATE**, **DELETE**, or **INSERT** statement.

To determine what operations **SQLBulkOperation** supports, an application calls **SQLGetInfo** with the **SQL\_DYNAMIC\_CURSOR\_ATTRIBUTES1**, **SQL\_FORWARD\_ONLY\_CURSOR\_ATTRIBUTES1**, **SQL\_KEYSET\_CURSOR\_ATTRIBUTES1**, or **SQL\_STATIC\_CURSOR\_ATTRIBUTES1** information option (depending on the type of the cursor).

## Updating Rows by Bookmark with SQLBulkOperations

When updating a row by bookmark, **SQLBulkOperations** makes the data source update one or more rows of the table. The rows are identified by the bookmark in a bound bookmark column. The row is updated using data in the application buffers for each bound column (except when the value in the length/indicator buffer for a column is `SQL_COLUMN_IGNORE`). Unbound columns will not be updated.

To update rows by bookmark with **SQLBulkOperations**, the application:

1. Retrieves and caches the bookmarks of all rows to be updated. If there is more than one bookmark and column-wise binding is used, the bookmarks are stored in an array; if there is more than one bookmark and row-wise binding is used, the bookmarks are stored in an array of row structures.
2. Sets the `SQL_ATTR_ROW_ARRAY_SIZE` statement attribute to the number of bookmarks, and binds the buffer containing the bookmark value, or the array of bookmarks, to column 0.
3. Places the new data values in the rowset buffers. For information on how to send long data with **SQLBulkOperations**, see “[Long Data and SQLSetPos and SQLBulkOperations](#)” later in this chapter.
4. Sets the value in the length/indicator buffer of each column as necessary. This is the byte length of the data or `SQL_NTS` for columns bound to string buffers, the byte length of the data for columns bound to binary buffers, and `SQL_NULL_DATA` for any columns to be set to NULL.
5. Sets the value in the length/indicator buffer of those columns that are not to be updated to `SQL_COLUMN_IGNORE`. Although the application can skip this step and resend existing data, this is inefficient and risks sending values to the data source that were truncated when they were read.
6. Calls **SQLBulkOperations** with the *Operation* argument set to `SQL_UPDATE_BY_BOOKMARK`.

For every row that is sent to the data source as an update, the application buffers should have valid row data. If the application buffers were filled by fetching, a row status array has been maintained, and the status value for a row is `SQL_ROW_DELETED`, `SQL_ROW_ERROR`, or `SQL_ROW_NOROW`, invalid data could inadvertently be sent to the data source.

## Deleting Rows by Bookmark with SQLBulkOperations

When deleting a row by bookmark, **SQLBulkOperations** makes the data source delete one or more selected rows of the table. The rows are identified by the bookmark in a bound bookmark column.

To delete rows by bookmark with **SQLBulkOperations**, the application:



1. Retrieves and caches the bookmarks of all rows to be deleted. If there is more than one bookmark and column-wise binding is used, the bookmarks are stored in an array; if there is more than one bookmark and row-wise binding is used, the bookmarks are stored in an array of row structures.
2. Sets the `SQL_ATTR_ROW_ARRAY_SIZE` statement attribute to the number of bookmarks, and binds the buffer containing the bookmark value, or the array of bookmarks, to column 0.
3. Calls **SQLBulkOperations** with *Operation* set to `SQL_DELETE_BY_BOOKMARK`.

## Inserting Rows with SQLBulkOperations

Inserting data with **SQLBulkOperations** is similar to updating data with **SQLBulkOperations** because it uses data from the bound application buffers.

So that each column in the new row has a value, all bound columns with a length/indicator value of `SQL_COLUMN_IGNORE` and all unbound columns must either accept NULL values or have a default.

To insert rows with **SQLBulkOperations**, the application:

1. Sets the `SQL_ATTR_ROW_ARRAY_SIZE` statement attribute to the number of rows to insert, and places the new data values in the bound application buffers. For information on how to send long data with **SQLBulkOperations**, see “[Long Data and SQLSetPos and SQLBulkOperations](#)” later in this chapter.
2. Sets the value in the length/indicator buffer of each column as necessary. This is the byte length of the data or `SQL_NTS` for columns bound to string buffers, the byte length of the data for columns bound to binary buffers, and `SQL_NULL_DATA` for any columns to be set to NULL. The application sets the value in the length/indicator buffer of those columns which are to be set to their default (if one exists) or NULL (if one does not) to `SQL_COLUMN_IGNORE`.
3. Calls **SQLBulkOperations** with the *Operation* argument set to `SQL_ADD`.

After **SQLBulkOperations** returns, the current row is unchanged. If the bookmark column (column 0) is bound, **SQLBulkOperations** returns the bookmarks of the inserted rows in the rowset buffer bound to that column.

## Fetching Rows with SQLBulkOperations

Data can be refetched into a rowset using bookmarks by a call to **SQLBulkOperations**. The rows to be fetched are identified by the bookmarks in a bound bookmark column. Columns with a value of `SQL_COLUMN_IGNORE` are not fetched.

To perform bulk fetches with **SQLBulkOperations**, the application:

1. Retrieves and caches the bookmarks of all rows to be updated. If there is more than one bookmark and column-wise binding is used, the bookmarks are stored in an array; if there is more than one bookmark and row-wise binding is used, the bookmarks are stored in an array of row structures.
2. Sets the `SQL_ATTR_ROW_ARRAY_SIZE` statement attribute to the number of rows to fetch, and binds the buffer containing the bookmark value, or the array of bookmarks, to column 0.
3. Sets the value in the length/indicator buffer of each column as necessary. This is the byte length of the data or `SQL_NTS` for columns bound to string buffers, the byte length of the data for columns bound to binary buffers, and `SQL_NULL_DATA` for any columns to be set to NULL. The application sets the value in the length/indicator buffer of those columns which are to be set to their default (if one exists) or NULL (if one does not) to `SQL_COLUMN_IGNORE`.
4. Calls **SQLBulkOperations** with the *Operation* argument set to `SQL_FETCH_BY_BOOKMARK`.

There is no need for the application to use the row operation array to prevent the operation to be performed on certain columns. The application selects the rows it wants to fetch by copying only the bookmarks for those rows into the bound bookmark array.

## Long Data and SQLSetPos and SQLBulkOperations

As is the case with parameters in SQL statements, long data can be sent when updating rows with **SQLBulkOperations** or **SQLSetPos** or inserting rows with **SQLBulkOperations**. The data is sent in parts with multiple calls to **SQLPutData**. Columns for which data is sent at execution time are known as *data-at-execution columns*.

**Note** An application actually can send any type of data at execution time with **SQLPutData**, although only character and binary data can be sent in parts. However, if the data is small enough to fit in a single buffer, there is generally no reason to use **SQLPutData**. It is much easier to bind the buffer and let the driver retrieve the data from the buffer.

Because long data columns typically are not bound, the application must bind the column before calling **SQLBulkOperations** or **SQLSetPos**, and unbind it after calling **SQLBulkOperations** or **SQLSetPos**. The column must be bound because **SQLBulkOperations** or **SQLSetPos** operates only on bound columns, and must be unbound so **SQLGetData** can be used to retrieve data from the column.

To send data at execution time, the application:

1. Places a 32-bit value in the rowset buffer instead of a data value. This value will be returned to the application later, so the application should set it to a meaningful value, such as the number of the column or the handle of a file containing data.
2. Sets the value in the length/indicator buffer to the result of the `SQL_LEN_DATA_AT_EXEC(length)` macro. This value indicates to the driver that the data for the parameter will be sent with **SQLPutData**. The *length* value is used when sending long data to a data source that needs to know how many bytes of long data will be sent so that it can

preallocate space. To determine if a data source requires this value, the application calls **SQLGetInfo** with the **SQL\_NEED\_LONG\_DATA\_LEN** option. All drivers must support this macro; if the data source does not require the byte length, the driver can ignore it.

3. Calls **SQLBulkOperations** or **SQLSetPos**. The driver discovers that a length/indicator buffer contains the result of the **SQL\_LEN\_DATA\_AT\_EXEC(*length*)** macro and returns **SQL\_NEED\_DATA** as the return value of the function.
4. Calls **SQLParamData** in response to the **SQL\_NEED\_DATA** return value. If long data needs to be sent, **SQLParamData** returns **SQL\_NEED\_DATA**. In the buffer pointed to by the *ValuePtrPtr* argument, the driver returns the unique value that the application placed in the rowset buffer. If there is more than one data-at-execution column, the application uses this value to determine which column to send data for; the driver is not required to request data for data-at-execution columns in any particular order.
5. Calls **SQLPutData** to send the column data to the driver. If the column data does not fit in a single buffer, as is often the case with long data, the application calls **SQLPutData** repeatedly to send the data in parts; it is up to the driver and data source to reassemble the data. If the application passes null-terminated string data, the driver or data source must remove the null-termination character as part of the reassembly process.
6. Calls **SQLParamData** again to indicate that it has sent all of the data for the column. If there are any data-at-execution columns for which data has not been sent, the driver returns **SQL\_NEED\_DATA** and the unique value for the next data-at-execution column; the application returns to Step 5. If data has been sent for all data-at-execution columns, the data for the row is sent to the data source. **SQLParamData** then returns **SQL\_SUCCESS** or **SQL\_SUCCESS\_WITH\_INFO**, and can return any **SQLSTATE** that **SQLBulkOperations** or **SQLSetPos** can return.

After **SQLBulkOperations** or **SQLSetPos** returns **SQL\_NEED\_DATA** and before data has been completely sent for the last data-at-execution column, the statement is in a Need Data state. In this state, the application can call only **SQLPutData**, **SQLParamData**, **SQLCancel**, **SQLGetDiagField**, or **SQLGetDiagRec**; all other functions return **SQLSTATE HY010** (Function sequence error). Calling **SQLCancel** cancels execution of the statement and returns it to its previous state. For more information, see Appendix B, “ODBC State Transition Tables” contained on the Microsoft Web site (ODBC Programmer’s Reference).

## Chapter 13: Overview of Descriptors

A descriptor handle refers to a data structure that holds information about either columns or dynamic parameters.

ODBC functions that operate on column and parameter data implicitly set and retrieve descriptor fields. For instance, when **SQLBindCol** is called to bind column data, it sets descriptor fields that completely describe the binding. When **SQLColAttribute** is called to describe column data, it returns data stored in descriptor fields.

An application calling ODBC functions need not concern itself with descriptors. No database operation requires that the application gain direct access to descriptors. However, for some applications, gaining direct access to descriptors streamlines many operations. For example, direct access to descriptors provides a way to rebind column data that may be more efficient than calling **SQLBindCol** again.

**Note** The physical representation of the descriptor is not defined. Applications gain direct access to a descriptor only by manipulating its fields by calling ODBC functions with the descriptor handle.

### Types of Descriptors

A descriptor is used to describe one of the following:

- A set of zero or more parameters. A parameter descriptor can be used to describe:
  - The *application parameter buffer*, which contains either the input dynamic arguments as set by the application or the output dynamic arguments following the execution of a **CALL** statement of SQL.
  - The *implementation parameter buffer*. For input dynamic arguments, this contains the same arguments as the application parameter buffer, after any data conversion the application may specify. For output dynamic arguments, this contains the returned arguments, before any data conversion that the application may specify.

For input dynamic arguments, the application must operate on an application parameter descriptor before executing any SQL statement that contains dynamic parameter markers. For both input and output dynamic arguments, the application may specify different data types from those in the implementation parameter descriptor to achieve data conversion.

- A single row of database data. A row descriptor can be used to describe:
  - The *implementation row buffer*, which contains the row from the database. (These buffers conceptually contain data as written to, or read from, the database. However, the stored form of database data is not specified. A database could perform additional conversion on the data from its form in the implementation buffer.)
  - The *application row buffer*, which contains the row of data as presented to the application, following any data conversion that the application may specify.

The application operates on the application row descriptor in any case where column data from the database must appear in application variables. The application may specify different data types from those in the implementation row descriptor to achieve data conversion of column data.

The descriptor types are summarized in the following table:

|                              | Rows                                | Dynamic parameters                        |
|------------------------------|-------------------------------------|---|
| <b>Application buffer</b>    | Application row descriptor (ARD)    | Application parameter descriptor (APD)    |
| <b>Implementation buffer</b> | Implementation row descriptor (IRD) | Implementation parameter descriptor (IPD) |

For either the parameter or row buffers, if the application specifies different data types in corresponding records of the implementation and application descriptor, the driver performs data conversion when it uses the descriptors. For example, it may convert numeric and datetime values to character-string format. (For valid conversions, see Appendix D, “Data Types” in the **SOLID Programmer Guide**.)

A descriptor may perform different roles. Different statements can share any descriptor that the application explicitly allocates. A row descriptor in one statement can serve as a parameter descriptor in another statement.

It is always known whether a given descriptor is an application descriptor or an implementation descriptor, even if the descriptor has not yet been used in a database operation. For the descriptors that the implementation implicitly allocates, the implementation records the predefined row relative to the statement handle. Any descriptor that the application allocates by calling **SQLAllocHandle** is an application descriptor.

## Descriptor Fields

Descriptors contain *header* and *record* fields that completely describe columns or parameters.

A descriptor contains a single copy of the following header fields. Changing a header field affects all columns or parameters.

|                           |                             |
|---------------------------|-----------------------------|
| SQL_DESC_ALLOC_TYPE       | SQL_DESC_BIND_TYPE          |
| SQL_DESC_ARRAY_SIZE       | SQL_DESC_COUNT              |
| SQL_DESC_ARRAY_STATUS_PTR | SQL_DESC_ROWS_PROCESSED_PTR |
| SQL_DESC_BIND_OFFSET_PTR  |                             |

A descriptor contains zero or more descriptor records. Each record describes a column or parameter, depending on the type of descriptor. When a new column or parameter is bound, a new record is added to the descriptor. When a column or parameter is unbound, a record is removed from the descriptor. Each record contains a single copy of the following fields:

|                                      |                           |
|--------------------------------------|---------------------------|
| SQL_DESC_AUTO_UNIQUE_VALUE           | SQL_DESC_LOCAL_TYPE_NAME  |
| SQL_DESC_BASE_COLUMN_NAME            | SQL_DESC_NAME             |
| SQL_DESC_BASE_TABLE_NAME             | SQL_DESC_NULLABLE         |
| SQL_DESC_CASE_SENSITIVE              | SQL_DESC_OCTET_LENGTH     |
| SQL_DESC_CATALOG_NAME                | SQL_DESC_OCTET_LENGTH_PTR |
| SQL_DESC_CONCISE_TYPE                | SQL_DESC_PARAMETER_TYPE   |
| SQL_DESC_DATA_PTR                    | SQL_DESC_PRECISION        |
| SQL_DESC_DATETIME_INTERVAL_CODE      | SQL_DESC_SCALE            |
| SQL_DESC_DATETIME_INTERVAL_PRECISION | SQL_DESC_SCHEMA_NAME      |
| SQL_DESC_DISPLAY_SIZE                | SQL_DESC_SEARCHABLE       |
| SQL_DESC_FIXED_PREC_SCALE            | SQL_DESC_TABLE_NAME       |
| SQL_DESC_INDICATOR_PTR               | SQL_DESC_TYPE             |
| SQL_DESC_LABEL                       | SQL_DESC_TYPE_NAME        |
| SQL_DESC_LENGTH                      | SQL_DESC_UNNAMED          |
| SQL_DESC_LITERAL_PREFIX              | SQL_DESC_UNSIGNED         |
| SQL_DESC_LITERAL_SUFFIX              | SQL_DESC_UPDATABLE        |

Many statement attributes correspond to the header field of a descriptor. Setting these attributes through a call to **SQLSetStmtAttr** and setting the corresponding descriptor header field by calling **SQLSetDescField** have the same effect. The same is true for **SQLGetStmtAttr** and **SQLGetDescField**, both of which retrieve the same information. Calling the statement functions instead of the descriptor functions has the advantage that a descriptor handle does not have to be retrieved.

The following header fields can be set by setting statement attributes:

|                           |                             |
|---------------------------|-----------------------------|
| SQL_DESC_ARRAY_SIZE       | SQL_DESC_BIND_TYPE          |
| SQL_DESC_ARRAY_STATUS_PTR | SQL_DESC_ROWS_PROCESSED_PTR |
| SQL_DESC_BIND_OFFSET_PTR  |                             |

## Record Count

The SQL\_DESC\_COUNT header field of a descriptor is the one-based index of the highest-numbered record that contains data. This field is not a count of all columns or parameters that are bound. When a descriptor is allocated, the initial value of SQL\_DESC\_COUNT is 0.

The driver takes any action necessary to allocate and maintain whatever storage it requires to hold descriptor information. The application does not explicitly specify the size of a descriptor nor allocate new records. When the application provides information for a descriptor record whose number is higher than the value of SQL\_DESC\_COUNT, the driver automatically increases SQL\_DESC\_COUNT. When the

application unbinds the highest-numbered descriptor record, the driver automatically decreases SQL\_DESC\_COUNT to contain the number of the highest remaining bound record.

## Bound Descriptor Records

When the application sets the SQL\_DESC\_DATA\_PTR field of a descriptor record so that it no longer contains a null value, the record is said to be *bound*.

If the descriptor is an APD, then each bound record constitutes a bound parameter. For input parameters, the application must bind a parameter for each dynamic parameter marker in the SQL statement before executing the statement. For output parameters, the application need not bind the parameter.

If the descriptor is an ARD, which describes a row of database data, then each bound record constitutes a bound column.

## Deferred Fields

The values of *deferred fields* are not used when they are set, but the driver saves the addresses of the variables for a deferred effect. For an application parameter descriptor, the driver uses the contents of the variables at the time of the call to **SQLExecDirect** or **SQLExecute**. For an application row descriptor, the driver uses the contents of the variables at the time of the fetch.

The following are deferred fields:

- The SQL\_DESC\_DATA\_PTR and SQL\_DESC\_INDICATOR\_PTR fields of a descriptor record.
- The SQL\_DESC\_OCTET\_LENGTH\_PTR field of an application descriptor record.
- In the case of a multirow fetch, the SQL\_DESC\_ARRAY\_STATUS\_PTR and SQL\_DESC\_ROWS\_PROCESSED\_PTR fields of a descriptor header.

When a descriptor is allocated, the deferred fields of each descriptor record initially have a null value. The meaning of the null value is as follows:

- If SQL\_DESC\_ARRAY\_STATUS\_PTR has a null value, a multirow fetch fails to return this component of the per-row diagnostic information.
- If SQL\_DESC\_DATA\_PTR has a null value, the record is unbound.
- If the SQL\_DESC\_OCTET\_LENGTH\_PTR field of an ARD has a null value, the driver does not return length information for that column.
- If the SQL\_DESC\_OCTET\_LENGTH\_PTR field of an APD has a null value, and the parameter is a character string, the driver assumes that string is null-terminated. For output dynamic parameters, a null value in this field prevents the driver from returning length information. (If the

SQL\_DESC\_TYPE field does not indicate a character-string parameter, the SQL\_DESC\_OCTET\_LENGTH\_PTR field is ignored.)

The application must not deallocate or discard variables used for deferred fields between the time it associates them with the fields and the time the driver reads or writes them.

## Consistency Check

A consistency check is performed by the driver automatically whenever an application sets the SQL\_DESC\_DATA\_PTR field of the APD, ARD, or IPD. Whenever this field is set, the driver checks that the value of the SQL\_DESC\_TYPE field and the values applicable to the SQL\_DESC\_TYPE field in the same record are valid and consistent.

The SQL\_DESC\_DATA\_PTR field of an IPD is not normally set; however, an application can do so to force a consistency check of IPD fields. The value that the SQL\_DESC\_DATA\_PTR field of the IPD is set to is not actually stored, and cannot be retrieved by a call to **SQLGetDescField** or **SQLGetDescRec**; the setting is made only to force the consistency check. A consistency check cannot be performed on an IRD.

For more information on the consistency check, see **SQLSetDescRec** in the Part II PDF file, “ODBC API Reference” available on the Solid Web site.

## Allocating and Freeing Descriptors

Descriptors are either implicitly or explicitly allocated, as described in the following sections.

### Implicitly Allocated Descriptors

When a statement handle is allocated, the application implicitly allocates one set of four descriptors. The application can obtain the handles of these implicitly allocated descriptors as attributes of the statement handle. When the application frees the statement handle, the driver frees all implicitly allocated descriptors on that handle.

### Explicitly Allocated Descriptors

An application can explicitly allocate an application descriptor on a connection at any time it is connected to the database. By specifying that descriptor handle as an attribute of a statement handle using **SQLSetStmtAttr**, the application directs the driver to use that descriptor in place of the corresponding implicitly allocated application descriptors. The application cannot specify alternate implementation descriptors.

An application can associate an explicitly allocated descriptor with more than one statement. Only when an application is actually connected to the database can a descriptor be an explicitly allocated descriptor. The application can free such a descriptor explicitly, or implicitly by freeing its connection.

## Initialization of Descriptor Fields

When an application row descriptor is allocated, its fields receive initial values as indicated **SQLSetDescRec** in the Part II PDF file, “ODBC API Reference” available on the Solid Web site. The initial value of the SQL\_DESC\_TYPE field is SQL\_DEFAULT. This provides for a standard treatment of database data for presentation to the application. The application may specify different treatment of the data by setting fields of the descriptor record.



The initial value of `SQL_DESC_ARRAY_SIZE` in the descriptor header is 1. The application can modify this field to enable multirow fetch.

The concept of a default value is not valid for the fields of an IRD. An application can gain access to the fields of an IRD only when there is a prepared or executed statement associated with it.

Certain fields of an IPD are defined only after the IPD has been automatically populated by the driver. If not, they are undefined. These fields are `SQL_DESC_CASE_SENSITIVE`, `SQL_DESC_FIXED_PREC_SCALE`, `SQL_DESC_TYPE_NAME`, `SQL_DESC_UNSIGNED`, and `SQL_DESC_LOCAL_TYPE_NAME`.

## Automatic Population of the IPD

Some drivers are capable of setting the fields of the IPD after a parameterized query has been prepared. The descriptor fields are automatically populated with information about the parameter, including the data type, precision, scale, and other characteristics. This is equivalent to supporting **SQLDescribeParam**. This information can be particularly valuable to an application when it has no other way to discover it, such as when an ad-hoc query is performed with parameters that the application does not know about.

An application determines whether the driver supports automatic population by calling **SQLGetConnectAttr** with an *Attribute* of `SQL_ATTR_AUTO_IPD`. If `SQL_TRUE` is returned, the driver supports it, and the application can enable it by setting the `SQL_ATTR_ENABLE_AUTO_IPD` statement attribute to `SQL_TRUE`.

When automatic population is supported and enabled, the driver populates the fields of the IPD after an SQL statement containing parameter markers has been prepared by a call to **SQLPrepare**. An application can retrieve this information by calling **SQLGetDescField** or **SQLGetDescRec**, or **SQLDescribeParam**. The application can use the information to bind the most appropriate application buffer for a parameter, or to specify a data conversion for it.

Automatic population of the IPD may produce a performance penalty. An application can turn it off by resetting the `SQL_ATTR_ENABLE_AUTO_IPD` statement attribute to `SQL_FALSE` (which is the default value).

## Freeing Descriptors

Explicitly allocated descriptors can be freed either explicitly, by calling **SQLFreeHandle** with *HandleType* of `SQL_HANDLE_DESC`, or implicitly, when the connection handle is freed. When an explicitly allocated descriptor is freed, all statement handles to which the freed descriptor applied automatically revert to the descriptors implicitly allocated for them.

Implicitly allocated descriptors can be freed only by calling **SQLDisconnect**, which drops any statements or descriptors open on the connection, or by calling **SQLFreeHandle** with a *HandleType* of `SQL_HANDLE_STMT` to free a statement handle and all the implicitly allocated descriptors associated with the statement. An implicitly allocated descriptor cannot be freed by calling **SQLFreeHandle** with a *HandleType* of `SQL_HANDLE_DESC`.

Even when freed, an implicitly allocated descriptor remains valid, and **SQLGetDescField** can be called on its fields.

## Getting and Setting Descriptor Fields

The following sections describe the methods an application can use to retrieve or set the values in descriptor fields.

### Obtaining Descriptor Handles

An application obtains the handle of any explicitly allocated descriptor as an output argument of the call to **SQLAllocHandle**. The handle of an implicitly allocated descriptor is obtained by calling **SQLGetStmtAttr**.

### Retrieving the Values in Descriptor Fields

An application can call **SQLGetDescField** to obtain a single field of a descriptor record. **SQLGetDescField** gives the application access to all the descriptor fields defined in ODBC, and to driver-defined fields as well.

**SQLGetDescRec** can be called to retrieve the settings of multiple descriptor fields that affect the data type and storage of column or parameter data.

### Setting Descriptor Fields

To modify the fields of a descriptor, an application can call **SQLSetDescField**. Some fields are read-only and cannot be set (see the **SQLSetDescField** function description in the Part II PDF file, “ODBC API Reference” available on the Solid Web site).

Descriptor record fields are set with a record number (*RecNumber*) of 1 or higher, while descriptor header fields are set with a record number of 0. A record number of 0 is also used to set bookmark fields, in accordance with the convention that bookmarks are contained in column 0. This may leave the impression that bookmark fields are contained within the descriptor header, but this is not the case. Bookmark fields are distinct from header fields.

When setting fields individually, the application should follow the sequence defined in **SQLSetDescField** in the Part II PDF file, “ODBC API Reference” available on the Solid Web site. Setting some fields causes the driver to set other fields. This ensures that the descriptor is always ready to use once the application has specified a data type. When the application sets the **SQL\_DESC\_TYPE** field, the driver checks that other fields that specify the type are valid and consistent.

If a function call that would set a descriptor field fails, the contents of the descriptor field are undefined after the failed function call.

### Copying Descriptors

The **SQLCopyDesc** function is called to copy the fields of one descriptor to another descriptor. Fields can only be copied to an application descriptor or an IPD, but not to an IRD. Fields can be copied from any type of descriptor. Only those fields that are defined for both the source and target descriptors are copied. **SQLCopyDesc** does not copy the **SQL\_DESC\_ALLOC\_TYPE** field, because a descriptor’s allocation type cannot be changed. Copied fields overwrite the existing fields.

An ARD on one statement handle can serve as the APD on another statement handle. This allows an application to copy rows between tables without copying data at the application level. To do this, a row descriptor that describes a fetched row of a table is reused as a parameter descriptor for a parameter in an INSERT statement. The `SQL_MAX_CONCURRENT_ACTIVITIES` information type must be greater than 1 for this operation to succeed.

## Using Concise Functions

Some ODBC functions gain implicit access to descriptors. Application writers may find them more convenient than calling **SQLSetDescField** or **SQLGetDescField**. These functions are called *concise* functions because they perform a number of functions, including setting or getting descriptor fields. Some concise functions let an application set or retrieve several related descriptor fields in a single function call.

Concise functions can be called without first retrieving a descriptor handle for use as an argument. These functions work with the descriptor fields associated with the statement handle that they are called on.

The concise functions **SQLBindCol** and **SQLBindParameter** bind a column or parameter by setting the descriptor fields that correspond to their arguments. Each of these functions performs more tasks than simply setting descriptors. **SQLBindCol** and **SQLBindParameter** provide a complete specification of the binding of a data column or dynamic parameter. An application can, however, change individual details of a binding by calling **SQLSetDescField** or **SQLSetDescRec**, and can completely bind a column or parameter by making a series of suitable calls to these functions.

The concise functions `SQLColAttribute`, `SQLDescribeCol`, `SQLDescribeParam`, `SQLNumParams`, and `SQLNumResultCols` retrieve values in descriptor fields.

**SQLSetDescRec** and **SQLGetDescRec** are concise functions that, with one call, set or get multiple descriptor fields that affect the data type and storage of column or parameter data. **SQLSetDescRec** is an effective way to change the binding of column or parameter data in one step.

**SQLSetStmtAttr** and **SQLGetStmtAttr** serve as concise functions in some cases (see “[Descriptor Fields](#)” earlier in this chapter).

## Chapter 14: Overview of Transactions

A *transaction* is a unit of work that is done as a single, atomic operation; that is, the operation succeeds or fails as a whole. For example, consider transferring money from one bank account to another. This involves two steps: withdrawing the money from the first account and depositing it in the second. It is important that both steps succeed; it is not acceptable for one step to succeed and the other to fail. A database that supports transactions is able to guarantee this.

Transactions can be completed by either being *committed* or being *rolled back*. When a transaction is committed, the changes made in that transaction are made permanent. When a transaction is rolled back, the affected rows are returned to the state they were in before the transaction was started. To extend the account transfer example, an application executes one SQL statement to debit the first account and a different SQL statement to credit the second account. If both statements succeed, the application then commits the transaction. But if either statement fails for any reason, the application rolls back the transaction. In either case, the application guarantees a consistent state at the end of the transaction.

A single transaction can encompass multiple database operations that occur at different times. If other transactions had complete access to the intermediate results, the transactions might interfere with one another. For example, suppose one transaction inserts a row, a second transaction reads that row, and the first transaction is rolled back. The second transaction now has data for a row that does not exist.

To solve this problem, there are various schemes to isolate transactions from each other. *Transaction isolation* is generally implemented by locking rows, which precludes more than one transaction from using the same row at the same time. In some databases, locking a row may also lock other rows.

With increased transaction isolation comes reduced *concurrency*, or the ability of two transactions to use the same data at the same time. For more information, see “[Setting the Transaction Isolation Level](#)” later in this chapter.

A full discussion of transactions is well beyond the scope of this book.

### Transactions in ODBC

Transactions in ODBC are completed at the connection level; that is, when an application completes a transaction, it commits or rolls back all work done through all statement handles on that connection.

### Transaction Support

The degree of support for transactions is driver-defined. ODBC is designed to be implemented on a single-user or desktop database that has no need to manage multiple updates to its data. Moreover, some databases that support transactions do so only for the Data Manipulation Language (DML) statements of SQL; there are restrictions or special transaction semantics regarding the use of Data Definition Language (DDL) when a transaction is active. That is, there may be transaction support for multiple simultaneous updates to tables, but not for changing the number and definition of tables during a transaction.

An application determines whether transactions are supported, whether DDL can be included in a transaction, and any special effects of including DDL in a transaction, by calling **SQLGetInfo** with the

SQL\_TXN\_CAPABLE option. For more information, see the SQLGetInfo function description contained on the Microsoft Web site (ODBC Programmer's Reference).

If the driver does not support transactions, but the application has the ability (using an API other than ODBC) to lock and unlock data, applications can achieve transaction support by locking and unlocking records and tables as needed. To implement the account-transfer example, the application would lock the records for both accounts, copy the current values, debit the first account, credit the second account, and unlock the records. If any steps failed, the application would reset the accounts using the copies.

Even data sources that support transactions might not be able to support more than one transaction at a time within a particular environment. Applications call **SQLGetInfo** with the SQL\_MULTIPLE\_ACTIVE\_TXN option to determine whether a data source can support simultaneous active transactions on more than one connection in the same environment. Because there is one transaction per connection, this is only interesting to applications that have multiple connections to the same data source.

## **Commit Mode**

Transactions in ODBC can be in one of two modes: auto-commit mode or manual-commit mode, as described in the following sections.

### ***Auto-Commit Mode***

*In auto-commit mode*, every database operation is a transaction that is committed when performed. This mode is suitable for many real-world transactions that consist of a single SQL statement. It is unnecessary to delimit or specify completion of these transactions. In databases without transaction support, auto-commit mode is the only supported mode. In such databases, statements are committed when they are executed and there is no way to roll them back; they are therefore always in auto-commit mode.

If the underlying DBMS does not support auto-commit mode transactions, the driver can emulate them by manually committing each SQL statement as it is executed.

If a batch of SQL statements is executed in auto-commit mode, it is data source – specific when the statements in the batch are committed. They can be committed as they are executed or as a whole after the entire batch has been executed. Some data sources may support both of these behaviors and may provide a way of selecting one or the others. In particular, if an error occurs in the middle of the batch, it is data source – specific whether the already-executed statements are committed or rolled back. Thus, interoperable applications that use batches and require them to be committed or rolled back as a whole should only execute batches in manual-commit mode.

### ***Manual-Commit Mode***

*In manual-commit mode*, applications must explicitly complete transactions by calling **SQLEndTran** to commit them or roll them back. This is the normal transaction mode for most relational databases.

Transactions in ODBC do not have to be explicitly initiated. Instead, a transaction begins implicitly whenever the application starts operating on the database. If the data source requires explicit transaction initiation, the driver must provide it whenever the application executes a statement requiring a transaction and there is no current transaction.

### *Setting the Commit Mode*

Applications specify the transaction mode with the `SQL_ATTR_AUTOCOMMIT` connection attribute. By default, ODBC transactions are in auto-commit mode (unless **SQLSetConnectAttr** and **SQLSetConnectOption** are not supported, which is unlikely). Switching from manual-commit mode to auto-commit mode automatically commits any open transaction on the connection.

### **Committing and Rolling Back Transactions**

To commit or roll back a transaction in manual-commit mode, an application calls **SQLEndTran**. Drivers for DBMSs that support transactions typically implement this function by executing a **COMMIT** or **ROLLBACK** statement. The Driver Manager does not call **SQLEndTran** when the connection is in auto-commit mode; it simply returns `SQL_SUCCESS`, even if the application attempts to roll back the transaction. Because drivers for DBMSs that do not support transactions are always in auto-commit mode, they can either implement **SQLEndTran** to return `SQL_SUCCESS` without doing anything or not implement it at all.

**Note** Applications should not commit or roll back transactions by executing **COMMIT** or **ROLLBACK** statements with **SQLExecute** or **SQLExecDirect**. The effects of doing this are undefined. Possible problems include the driver no longer knowing when a transaction is active and these statements failing against data sources that do not support transactions. These applications should call **SQLEndTran** instead.

If an application passes the environment handle to **SQLEndTran** but does not pass a connection handle, the Driver Manager conceptually calls **SQLEndTran** with the environment handle for each driver that has one or more active connections in the environment. The driver then commits the transactions on each connection in the environment. However, it is important to realize that neither the driver nor the Driver Manager performs a two-phase commit on the connections in the environment; this is merely a programming convenience to simultaneously call **SQLEndTran** for all connections in the environment.

(A *two-phase commit* is generally used to commit transactions that are spread across multiple data sources. In its first phase, the data sources are polled as to whether they can commit their part of the transaction. In the second phase, the transaction is actually committed on all data sources. If any data sources reply in the first phase that they cannot commit the transaction, the second phase does not occur.)

### **Effect of Transactions on Cursors and Prepared Statements**

Committing or rolling back a transaction has the following effect on cursors and access plans:

- All cursors are closed and access plans for prepared statements on that connection are deleted.
- All cursors are closed and access plans for prepared statements on that connection remain intact.
- All cursors remain open and access plans for prepared statements on that connection remain intact.

For example, suppose a data source exhibits the first behavior in this list, the most restrictive of these behaviors. Now suppose an application does the following:

1. Sets the commit mode to manual commit.

2. Creates a result set of sales orders on statement 1.
3. Creates a result set of the lines in a sales order on statement 2 when the user highlights that order.
4. Calls **SQLExecute** to execute a positioned update statement that has been prepared on statement 3 when the user updates a line.
5. Calls **SQLEndTran** to commit the positioned update statement.

Because of the data source's behavior, the call to **SQLEndTran** in step 5 causes it to close the cursors on statements 1 and 2 and to delete the access plan on all statements. The application must reexecute statements 1 and 2 to re-create the result sets and reprepare the statement on statement 3.

In auto-commit mode, functions other than **SQLEndTran** commit transactions:

- **SQLExecute** or **SQLExecDirect**. In the previous example, the call to **SQLExecute** in step 4 commits a transaction. This causes the data source to close the cursors on statements 1 and 2 and delete the access plan on all statements on that connection.
- **SQLBulkOperations** or **SQLSetPos**. In the previous example, suppose that in step 4 the application calls **SQLSetPos** with the `SQL_UPDATE` option on statement 2 instead of executing a positioned update statement on statement 3. This commits a transaction and causes the data source to close the cursors on statements 1 and 2, and discards all access plans on that connection.
- **SQLCloseCursor**. In the previous example, suppose that, when the user highlights a different sales order, the application calls **SQLCloseCursor** on statement 2 before creating a result of the lines for the new sales order. The call to **SQLCloseCursor** commits the **SELECT** statement that created the result set of lines and causes the data source to close the cursor on statement 1, and then discards all access plans on that connection.

Applications, especially screen-based applications in which the user scrolls around the result set and updates or deletes rows, must be careful to code around this behavior.

To determine how a data source behaves when a transaction is committed or rolled back, an application calls **SQLGetInfo** with the `SQL_CURSOR_COMMIT_BEHAVIOR` and `SQL_CURSOR_ROLLBACK_BEHAVIOR` options.

## Transaction Isolation

*Transaction isolation* refers to the degree of interaction between multiple concurrent transactions. To see why this is important, you'll need to first look at the idea of serializability.

### *Serializability*

Ideally, transactions should be *serializable*. Transactions are said to be serializable if the results of running transactions simultaneously are the same as the results of running them serially, that is, one after the other.

It is not important which transaction executes first, only that the result does not reflect any mixing of the transactions.

For example, suppose transaction A multiplies data values by 2 and transaction B adds 1 to data values. Now suppose that there are two data values: 0 and 10. If these transactions are run one after another, the new values will be 1 and 21 if transaction A is run first, or 2 and 22 if transaction B is run first. But what if the order in which the two transactions are run is different for each value? If transaction A is run first on the first value and transaction B is run first on the second value, the new values will be 1 and 22. If this order is reversed, the new values are 2 and 21. The transactions are serializable if 1, 21 and 2, 22 are the only possible results. The transactions are not serializable if 1, 22 or 2, 21 is a possible result.

So why is serializability desirable? In other words, why is it important that it appears that one transaction finishes before the next transaction starts? Consider the following problem. A salesman is entering orders at the same time a clerk is sending out bills. Suppose the salesman enters an order from Company X but does not commit it; the salesman is still talking to the representative from Company X. The clerk requests a list of all open orders and discovers the order for Company X and sends them a bill. Now the representative from Company X decides they want to change their order, so the salesman changes it before committing the transaction. Company X gets an incorrect bill.

If the salesman's and clerk's transactions were serializable, this problem would never have occurred. Either the salesman's transaction would have finished before the clerk's transaction started, in which case the clerk would have sent out the correct bill, or the clerk's transaction would have finished before the salesman's transaction started, in which case the clerk would not have sent a bill to Company X at all.

### ***Transaction Isolation Levels***

*Transaction isolation levels* are a measure of the extent to which transaction isolation succeeds. In particular, transaction isolation levels are defined by the presence or absence of the following phenomena:

- **Dirty Reads.** A *dirty read* occurs when a transaction reads data that has not yet been committed. For example, suppose transaction 1 updates a row. Transaction 2 reads the updated row before transaction 1 commits the update. If transaction 1 rolls back the change, transaction 2 will have read data that is considered never to have existed.
- **Nonrepeatable Reads.** A *nonrepeatable read* occurs when a transaction reads the same row twice but gets different data each time. For example, suppose transaction 1 reads a row. Transaction 2 updates or deletes that row and commits the update or delete. If transaction 1 rereads the row, it retrieves different row values or discovers that the row has been deleted.
- **Phantoms.** A *phantom* is a row that matches the search criteria but is not initially seen. For example, suppose transaction 1 reads a set of rows that satisfy some search criteria. Transaction 2 generates a new row (through either an update or an insert) that matches the search criteria for transaction 1. If transaction 1 reexecutes the statement that reads the rows, it gets a different set of rows.

The four transaction isolation levels (as defined by SQL-92) are defined in terms of these phenomena. In the following table, an "X" marks each phenomenon that can occur.



| Transaction isolation level | Dirty reads | Nonrepeatable reads | Phantoms |
|-----------------------------|-------------|---------------------|----------|
| Read uncommitted            | X           | X                   | X        |
| Read committed              | --          | X                   | X        |
| Repeatable read             | --          | --                  | X        |
| Serializable                | --          | --                  | --       |

The following table describes simple ways that a DBMS might implement the transaction isolation levels.

**Important** Most DBMSs use more complex schemes than these to increase concurrency. These examples are provided for illustration purposes only. In particular, ODBC does not prescribe how particular DBMSs isolate transactions from each other.

| Transaction isolation | Possible implementation  |
|-----------------------|--|
| Read uncommitted      | Transactions are not isolated from each other. If the DBMS supports other transaction isolation levels, it ignores whatever mechanism it uses to implement those levels. So that they do not adversely affect other transactions, transactions running at the Read Uncommitted level are usually read-only.  |
| Read committed        | <p>The transaction waits until rows write-locked by other transactions are unlocked; this prevents it from reading any "dirty" data.</p> <p>The transaction holds a read lock (if it only reads the row) or write lock (if it updates or deletes the row) on the current row to prevent other transactions from updating or deleting it. The transaction releases read locks when it moves off the current row. It holds write locks until it is committed or rolled back.</p>   |
| Repeatable read       | <p>The transaction waits until rows write-locked by other transactions are unlocked; this prevents it from reading any "dirty" data.</p> <p>The transaction holds read locks on all rows it returns to the application and write locks on all rows it inserts, updates, or deletes. For example, if the transaction includes the SQL statement <b>SELECT * FROM Orders</b>, the transaction read-locks rows as the application fetches them. If the transaction includes the SQL statement <b>DELETE FROM Orders WHERE Status = 'CLOSED'</b>, the transaction write-locks rows as it deletes them.</p> <p>Because other transactions cannot update or delete these rows, the current transaction avoids any nonrepeatable reads. The transaction releases its locks when it is committed or rolled back.</p> |
| Serializable          | <p>The transaction waits until rows write-locked by other transactions are unlocked; this prevents it from reading any "dirty" data.</p> <p>The transaction holds a read lock (if it only reads rows) or write lock (if it can update or delete rows) on the range of rows it affects. For example, if the transaction includes the SQL statement <b>SELECT * FROM Orders</b>, the range is the entire Orders table; the transaction read-locks the table and does not allow any new rows to be inserted into it. If the transaction includes the SQL statement <b>DELETE FROM Orders WHERE Status = 'CLOSED'</b>, the range is all rows</p>   |

|  |   |
|--|---|
|  | <p>with a Status of "CLOSED"; the transaction write-locks all rows in the Orders table with a Status of "CLOSED" and does not allow any rows to be inserted or updated such that the resulting row has a Status of "CLOSED".</p> <p>Because other transactions cannot update or delete the rows in the range, the current transaction avoids any nonrepeatable reads. Because other transactions cannot insert any rows in the range, the current transaction avoids any phantoms. The transaction releases its lock when it is committed or rolled back.</p> |
|--|---|

It is important to note that the transaction isolation level does not affect a transaction's ability to see its own changes; transactions can always see any changes they make. For example, a transaction might consist of two **UPDATE** statements, the first of which raises the pay of all employees by 10 percent and the second of which sets the pay of any employees over some maximum amount to that amount. This succeeds as a single transaction only because the second **UPDATE** statement can see the results of the first.

### ***Setting the Transaction Isolation Level***

To set the transaction isolation level, an application uses the `SQL_ATTR_TXN_ISOLATION` connection attribute. If the data source does not support the requested isolation level, the driver or data source can set a higher level. To determine what transaction isolation levels a data source supports and what the default isolation level is, an application calls **SQLGetInfo** with the `SQL_TXN_ISOLATION_OPTION` and `SQL_DEFAULT_TXN_ISOLATION` options, respectively.

Higher levels of transaction isolation offer the most protection for the integrity of database data. Serializable transactions are guaranteed to be unaffected by other transactions and therefore guaranteed to maintain database integrity.

However, a higher level of transaction isolation can cause slower performance because it increases the chances that the application will have to wait for locks on data to be released. An application may specify a lower level of isolation to increase performance in the following cases:

- When it can be guaranteed that no other transactions exist that might interfere with an application's transactions. This situation occurs only in limited circumstances, such as when one person in a small company maintains dBASE files containing personnel data on one computer and does not share these files.
- When speed is more critical than accuracy and any errors are likely to be small. For example, suppose that a company makes many small sales and that large sales are rare. A transaction that estimates the total value of all open sales might safely use the Read Uncommitted isolation level. Although the transaction would include orders in the process of being opened or closed that are subsequently rolled back, these would tend to cancel each other out and the transaction would be much faster because it is not blocked each time it encounters such an order.

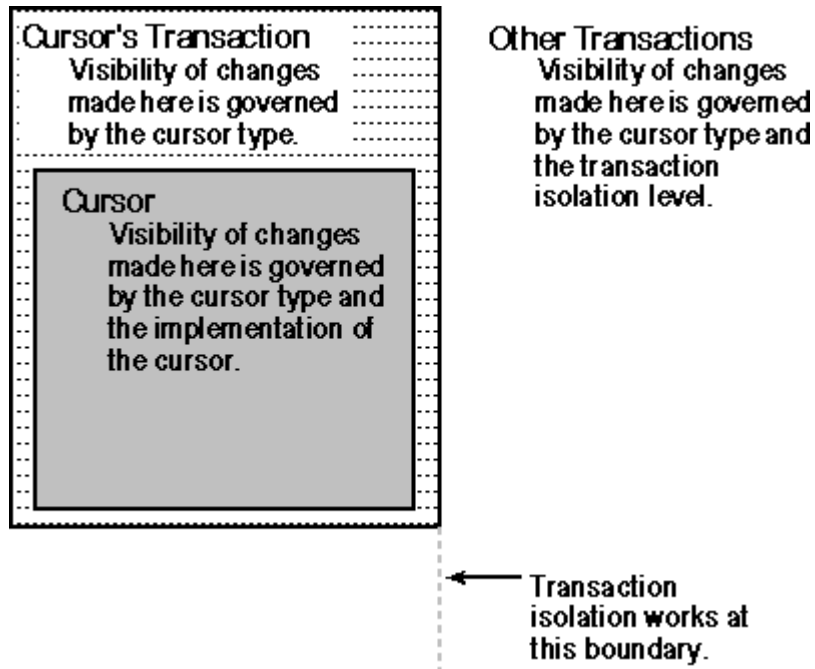
For more information, see the "[Optimistic Concurrency](#)" section later in this chapter.

### ***Scrollable Cursors and Transaction Isolation***

The following table lists the factors governing the visibility of changes.

| Changes made by:                     | Visibility depends on:                   |
|--------------------------------------|--|
| Cursor                               | Cursor type, cursor implementation       |
| Other statements in same transaction | Cursor type                              |
| Statements in other transactions     | Cursor type, transaction isolation level |

These factors are illustrated here.



### Visibility of changes

The following table summarizes the ability of each cursor type to detect changes made by itself, by other operations in its own transaction, and by other transactions. The visibility of the latter changes depends on the cursor type and the isolation level of the transaction containing the cursor.

|               | Self      | Own Txn | Othr Txn (RU [a]) | Othr Txn (RC [a]) | Othr Txn (RR [a]) | Othr Txn (S [a]) |
|---------------|-----------|---------|-------------------|-------------------|-------------------|------------------|
| Static        |           |         |                   |                   |                   |                  |
| Insert        | Maybe [b] | No      | No                | No                | No                | No               |
| Update        | Maybe [b] | No      | No                | No                | No                | No               |
| Delete        | Maybe [b] | No      | No                | No                | No                | No               |
| Keyset-driven |           |         |                   |                   |                   |                  |
| Insert        | Maybe [b] | No      | No                | No                | No                | No               |

|         |           |     |     |     |     |    |
|---------|-----------|-----|-----|-----|-----|----|
| Update  | Yes       | Yes | Yes | Yes | No  | No |
| Delete  | Maybe [b] | Yes | Yes | Yes | No  | No |
| Dynamic |           |     |     |     |     |    |
| Insert  | Yes       | Yes | Yes | Yes | Yes | No |
| Update  | Yes       | Yes | Yes | Yes | No  | No |
| Delete  | Yes       | Yes | Yes | Yes | No  | No |

[a] The letters in parentheses indicate the isolation level of the transaction containing the cursor; the isolation level of the other transaction (in which the change was made) is irrelevant.

RU: Read uncommitted

RC: Read committed

RR: Repeatable read

S: Serializable

[b] Depends on how the cursor is implemented. Whether the cursor can detect such changes is reported through the `SQL_STATIC_SENSITIVITY` option in **SQLGetInfo**.

## Concurrency Control

With increased transaction isolation usually comes reduced *concurrency*, or the ability of two transactions to use the same data at the same time. The reason for this is that transaction isolation is usually implemented by locking rows and, as more rows are locked, fewer transactions can be completed without being blocked at least temporarily by a locked row. While reduced concurrency is generally accepted as a trade-off for the higher transaction isolation levels necessary to maintain database integrity, it can become a problem in interactive applications with high read/write activity that use cursors.

For example, suppose an application executes the SQL statement **SELECT \* FROM Orders**. It calls **SQLFetchScroll** to scroll around the result set and allows the user to update, delete, or insert orders. After the user updates, deletes, or inserts an order, the application commits the transaction.

If the isolation level is Repeatable Read, the transaction might—depending on how it is implemented—lock each row returned by **SQLFetchScroll**. If the isolation level is Serializable, the transaction might lock the entire Orders table. In either case, the transaction releases its locks only when it is committed or rolled back. Thus, if the user spends a lot of time reading orders and very little time updating, deleting, or inserting them, the transaction could easily lock a large number of rows, making them unavailable to other users.

Note that this is a problem even if the cursor is read-only and the application only allows the user to read existing orders. In this case, the application commits the transaction—and releases locks—when it calls **SQLCloseCursor** (in auto-commit mode) or **SQLEndTran** (in manual-commit mode).

## Concurrency Types

To solve the problem of reduced concurrency in cursors, ODBC exposes four different types of cursor concurrency:

- **Read-only.** The cursor can read data but cannot update or delete data. This is the default concurrency type. Although the DBMS might lock rows to enforce the Repeatable Read and Serializable isolation levels, it can use read locks instead of write locks. This results in higher concurrency because other transactions can at least read the data.
- **Locking.** The cursor uses the lowest level of locking necessary to make sure it can update or delete rows in the result set. This usually results in very low concurrency levels, especially at the Repeatable Read and Serializable transaction isolation levels.
- **Optimistic concurrency using row versions and optimistic concurrency using values.** The cursor uses optimistic concurrency: It updates or deletes rows only if they have not changed since they were last read. To detect changes, it compares row versions or values. There is no guarantee that the cursor will be able to update or delete a row, but concurrency is much higher than when locking is used. For more information, see the following section, "[Optimistic Concurrency](#)."

An application specifies what type of concurrency it wants the cursor to use with the `SQL_ATTR_CONCURRENCY` statement attribute. To determine what types are supported, it calls `SQLGetInfo` with the `SQL_SCROLL_CONCURRENCY` option.

### ***Optimistic Concurrency***

*Optimistic concurrency* derives its name from the optimistic assumption that collisions between transactions will rarely occur; a collision is said to have occurred when another transaction updates or deletes a row of data between the time it is read by the current transaction and it is updated or deleted. It is the opposite of *pessimistic concurrency*, or locking, in which the application developer believes that such collisions are commonplace.

In optimistic concurrency, a row is left unlocked until the time comes to update or delete it. At that point, the row is reread and checked to see if it has been changed since it was last read. If the row has changed, the update or delete fails and must be tried again.

To determine whether a row has been changed, its new version is checked against a cached version of the row. This checking can be based on the row version, such as the timestamp column in SQL Server or the ROWID column in Oracle, or the values of each column in the row. Note that many DBMSs do not support row versions.

Optimistic concurrency can be implemented by the data source or the application. In either case, the application should use a low transaction isolation level such as Read Committed; using a higher level negates the increased concurrency gained by using optimistic concurrency.

If optimistic concurrency is implemented by the data source, the application sets the `SQL_ATTR_CONCURRENCY` statement attribute to `SQL_CONCUR_ROWVER` or `SQL_CONCUR_VALUES`. To update or delete a row, it executes a positioned update or delete statement or calls `SQLSetPos` just as it would with pessimistic concurrency; the driver or data source returns `SQLSTATE 01001` (Cursor operation conflict) if the update or delete fails due to a collision.

If the application implements optimistic concurrency itself, then it sets the `SQL_ATTR_CONCURRENCY` statement attribute to `SQL_CONCUR_READ_ONLY` to read a row. If it will compare row versions and

does not know the row version column, it calls **SQLSpecialColumns** with the **SQL\_ROWVER** option to determine the name of this column.

The application updates or deletes the row by increasing the concurrency to **SQL\_CONCUR\_LOCK** (to gain write access to the row) and executing an **UPDATE** or **DELETE** statement with a **WHERE** clause that specifies the version or values the row had when the application read it. If the row has changed since then, the statement will fail. If the **WHERE** clause does not uniquely identify the row, the statement might also update or delete other rows; row versions always uniquely identify rows, but row values uniquely identify rows only if they include the primary key.

## Chapter 15: Overview of Diagnostics

Functions in ODBC return diagnostic information in two ways. The return code indicates the overall success or failure of the function, while diagnostic records provide detailed information about the function. At least one diagnostic record — the header record — is returned even if the function succeeds.

Diagnostic information is used at development time to catch programming errors such as invalid handles and syntax errors in hard-coded SQL statements. It is used at run time to catch run-time errors and warnings such as data truncation, access violations, and syntax errors in SQL statements entered by the user.

### Return Codes

Each function in ODBC returns a code, known as its *return code*, which indicates the overall success or failure of the function. Program logic is generally based on return codes.

For example, the following code calls **SQLFetch** to retrieve the rows in a result set. It checks the return code of the function to determine if the end of the result set was reached (**SQL\_NO\_DATA**), if any warning information was returned (**SQL\_SUCCESS\_WITH\_INFO**), or if an error occurred (**SQL\_ERROR**).

```
SQLRETURN rc;
SQLHSTMT hstmt;

while ((rc=SQLFetch(hstmt)) != SQL_NO_DATA) {
    if (rc == SQL_SUCCESS_WITH_INFO) {
        // Call function to display warning information.
    } else if (rc == SQL_ERROR) {
        // Call function to display error information.
        break;
    }
    // Process row.
}
```

The return code **SQL\_INVALID\_HANDLE** always indicates a programming error and should never be encountered at run time. All other return codes provide run-time information, although **SQL\_ERROR** may indicate a programming error.

The following table defines the return codes.

| Return code           | Description   |
|-----------------------|---|
| SQL_SUCCESS           | Function completed successfully. The application calls <b>SQLGetDiagField</b> to retrieve additional information from the header record.  |
| SQL_SUCCESS_WITH_INFO | Function completed successfully, possibly with a nonfatal error (warning). The application calls <b>SQLGetDiagRec</b> or <b>SQLGetDiagField</b> to retrieve additional information. |
| SQL_ERROR             | Function failed. The application calls <b>SQLGetDiagRec</b> or <b>SQLGetDiagField</b> to retrieve additional information. The   |

|                     |  |
|---------------------|--|
|                     | contents of any output arguments to the function are undefined.  |
| SQL_INVALID_HANDLE  | Function failed due to an invalid environment, connection, statement, or descriptor handle. This indicates a programming error. No additional information is available from <b>SQLGetDiagRec</b> or <b>SQLGetDiagField</b> . This code is returned only when the handle is a null pointer or is the wrong type, such as when a statement handle is passed for an argument that requires a connection handle. |
| SQL_NO_DATA         | No more data was available. The application calls <b>SQLGetDiagRec</b> or <b>SQLGetDiagField</b> to retrieve additional information. One or more driver-defined status records in class 02xxx may be returned.<br><br><b>Note</b> In ODBC 2.x, this return code was named SQL_NO_DATA_FOUND.   |
| SQL_NEED_DATA       | More data is needed, such as when parameter data is sent at execution time or additional connection information is required. The application calls <b>SQLGetDiagRec</b> or <b>SQLGetDiagField</b> to retrieve additional information, if any.  |
| SQL_STILL_EXECUTING | A function that was started asynchronously is still executing. The application calls <b>SQLGetDiagRec</b> or <b>SQLGetDiagField</b> to retrieve additional information, if any.  |

## Diagnostic Records

Associated with each environment, connection, statement, and descriptor handle are *diagnostic records*. These records contain diagnostic information about the last function called that used a particular handle. The records are replaced only when another function is called using that handle. There is no limit to the number of diagnostic records that can be stored at any one time.

There are two types of diagnostic records: a *header record* and zero or more *status records*. The header record is record 0; the status records are records 1 and above. Diagnostic records are composed of a number of separate fields, which are different for the header record and the status records. In addition, ODBC components can define their own diagnostic record fields.

Although diagnostic records can be thought of as structures, there is no requirement for them to actually be structures; how a driver stores the diagnostic information is driver-specific.

Fields in diagnostic records are retrieved with **SQLGetDiagField**. The SQLSTATE, native error number, and diagnostic message fields of status records can be retrieved in a single call with **SQLGetDiagRec**.



## Header Record

The fields in the header record contain general information about a function's execution, including the return code, row count, number of status records, and type of statement executed. The header record is always created unless the function returns `SQL_INVALID_HANDLE`. For a complete list of fields in the header record, see the `SQLGetDiagField` in the Part II PDF file, "ODBC API Reference" available on the Solid Web site.

## Status Records

The fields in the status records contain information about specific errors or warnings returned by the Driver Manager, driver, or data source, including the `SQLSTATE`, native error number, diagnostic message, column number, and row number. Status records can be created only if the function returns `SQL_ERROR`, `SQL_SUCCESS_WITH_INFO`, `SQL_NO_DATA`, `SQL_NEED_DATA`, or `SQL_STILL_EXECUTING`. For a complete list of fields in the status records, see the `SQLGetDiagField` in the Part II PDF file, "ODBC API Reference" available on the Solid Web site.

## Sequence of Status Records

If two or more status records are returned, the Driver Manager and driver rank them according to the following rules. The record with the highest rank is the first record. The source of a record (Driver Manager, driver, gateway, and so on) is not considered when ranking records.

- **Errors.** Status records that describe errors have the highest rank. Among error records, records that indicate a transaction failure or possible transaction failure outrank all other records. If two or more records describe the same error condition, then `SQLSTATE`s defined by the X/Open CLI specification (classes 03 through HZ) outrank ODBC- and driver-defined `SQLSTATE`s.
- **Implementation-defined No Data values.** Status records that describe driver-defined No Data values (class 02) have the second highest rank.
- **Warnings.** Status records that describe warnings (class 01) have the lowest rank. If two or more records describe the same warning condition, then warning `SQLSTATE`s defined by the X/Open CLI specification outrank ODBC- and driver-defined `SQLSTATE`s.

If there are two or more records with the highest rank, it is undefined which record is the first record. The order of all other records is undefined. In particular, because warnings may appear before errors, applications should check all status records when a function returns a value other than `SQL_SUCCESS`.

## SQLSTATEs

`SQLSTATE`s provide detailed information about the cause of a warning or error. The `SQLSTATE`s in this manual are based on those found in the ISO/IEF CLI specification, although those `SQLSTATE`s that start with IM are specific to ODBC.

Unlike return codes, the SQLSTATes in this manual are guidelines, and drivers are not required to return them. Thus, while drivers should return the proper SQLSTATE for any error or warning they are capable of detecting, applications should not count on this always occurring. The reasons for this situation are two-fold:

- **Incompleteness.** Although this manual lists a large number of errors and warnings and possible causes for those errors and warnings, it is not complete and probably never will be; driver implementations simply vary too much. Any given driver probably will not return all of the SQLSTATes listed in this manual and might return SQLSTATes not listed in this manual.
- **Complexity.** Some database engines — particularly relational database engines — return literally thousands of errors and warnings. The drivers for such engines are unlikely to map all of these errors and warnings to SQLSTATes because of the effort involved, the inexactness of the mappings, the large size of the resulting code, and the low value of the resulting code, which often returns programming errors that should never be encountered at run time. Thus, drivers should map as many errors and warnings as seems reasonable and be sure to map those errors and warnings on which application logic might be based, such as SQLSTATE 01004 (Data truncated).

Because SQLSTATes are not returned reliably, most applications just display them to the user along with their associated diagnostic message, which is often tailored to the specific error or warning that occurred, and native error code. There is rarely any loss of functionality in doing this, because applications cannot base programming logic on most SQLSTATes anyway. For example, suppose **SQLExecDirect** returns SQLSTATE 42000 (Syntax error or access violation). If the SQL statement that caused this error is hard-coded or built by the application, this is a programming error and the code needs to be fixed. If the SQL statement is entered by the user, this is a user error and the application has done all that is possible by informing the user of the problem.

When applications do base programming logic on SQLSTATes, they should be prepared for the SQLSTATE not to be returned or for a different SQLSTATE to be returned. Exactly which SQLSTATes are returned reliably can be based only on experience with numerous drivers. However, a general guideline is that SQLSTATes for errors that occur in the driver or Driver Manager, as opposed to the data source, are more likely to be returned reliably. For example, most drivers probably return SQLSTATE HYC00 (Optional feature not implemented) while fewer drivers probably return SQLSTATE 42021 (Column already exists).

The following SQLSTATes indicate run-time errors or warnings and are good candidates on which to base programming logic. However, there is no guarantee that all drivers return them.

- 01004 (Data truncated)
- 01S02 (Option value changed)
- HY008 (Operation canceled)
- HYC00 (Optional feature not implemented)

- HYT00 (Timeout expired)

SQLSTATE HYC00 (Optional feature not implemented) is particularly significant, because it is the only way in which an application can determine whether a driver supports a particular statement or connection attribute.

For a complete list of SQLSTATEs and what functions return them, see Appendix B, “Error Codes” in the **SOLID Programmer Guide**. For a detailed explanation of the conditions under which each function might return a particular SQLSTATE, see that function.

## Diagnostic Messages

A diagnostic message is returned with each SQLSTATE. The same SQLSTATE is often returned with a number of different messages. For example, SQLSTATE 42000 (Syntax error or access violation) is returned for most errors in SQL syntax. However, each syntax error is likely to be described by a different message.

Sample diagnostic messages are listed in the Error column in the table of SQLSTATEs in Appendix B, “Error Codes” of the **SOLID Programmer Guide** and in each function. Although drivers can return these messages, they are more likely to return whatever message is passed to them by the data source.

Applications generally display diagnostic messages to the user, along with the SQLSTATE and native error code. This helps the user and support personnel determine the cause of any problems. The component information embedded in the message is particularly helpful in doing this.

Diagnostic messages come from data sources and components in an ODBC connection, such as drivers, gateways, and the Driver Manager. Typically, data sources do not directly support ODBC. Consequently, if a component in an ODBC connection receives a message from a data source, it must identify the data source as the source of the message. It must also identify itself as the component that received the message.

If the source of an error or warning is a component itself, the diagnostic message must explain this. Therefore, the text of messages has two different formats. For errors and warnings that do not occur in a data source, the diagnostic message must use this format:

[vendor-identifier][ODBC-component-identifier]component-supplied-text

For errors and warnings that occur in a data source, the diagnostic message must use this format:

[vendor-identifier][ODBC-component-identifier][data-source-identifier]data-source-supplied-text

The following table shows the meaning of each element.

| Element                          | Meaning  |
|----------------------------------|--|
| <i>vendor-identifier</i>         | Identifies the vendor of the component in which the error or warning occurred or that received the error or warning directly from the data source. |
| <i>ODBC-component-identifier</i> | Identifies the component in which the error or warning occurred or that received the error or warning directly from the data source.               |
| <i>data-source-identifier</i>    | Identifies the data source. For file-based drivers, this is typically a  |

|                                  |   |
|----------------------------------|---|
|                                  | file format, such as Xbase. [1] For DBMS-based drivers, this is the DBMS product. |
| <i>component-supplied-text</i>   | Generated by the ODBC component.  |
| <i>data-source-supplied-text</i> | Generated by the data source.   |

[1] In this case, the driver is acting as both the driver and the data source.

Brackets ([ ]) must be included in the message and do not indicate optional items.

## Using SQLGetDiagRec and SQLGetDiagField

Applications call **SQLGetDiagRec** or **SQLGetDiagField** to retrieve diagnostic information. These functions accept an environment, connection, statement, or descriptor handle and return diagnostics from the function that last used that handle. The diagnostics logged on a particular handle are discarded when a new function is called using that handle. If the function returned multiple diagnostic records, the application calls these functions multiple times; the total number of status records is retrieved by calling **SQLGetDiagField** for the header record (record 0) with the SQL\_DIAG\_NUMBER option.

Applications retrieve individual diagnostic fields by calling **SQLGetDiagField** and specifying the field to retrieve. Certain diagnostic fields do not have any meaning for certain types of handles. For a list of diagnostic fields and their meanings, see the **SQLGetDiagField** in the Part II PDF file, “ODBC API Reference” available on the Solid Web site.

Applications retrieve the SQLSTATE, native error code, and diagnostic message in a single call by calling **SQLGetDiagRec**; **SQLGetDiagRec** cannot be used to retrieve information from the header record.

For example, the following code prompts the user for an SQL statement and executes it. If any diagnostic information was returned, it calls **SQLGetDiagField** to get the number of status records and **SQLGetDiagRec** to get the SQLSTATE, native error code, and diagnostic message from those records.

```
SQLCHAR  SqlState[6], SQLStmt[100], Msg[SQL_MAX_MESSAGE_LENGTH];
SQLINTEGER NativeError;
SQLSMALLINT i, MsgLen;
SQLRETURN rc1, rc2;
SQLHSTMT hstmt;

// Prompt the user for an SQL statement.
GetSQLStmt(SQLStmt);

// Execute the SQL statement and return any errors or warnings.
rc1 = SQLExecDirect(hstmt, SQLStmt, SQL_NTS);
if ((rc1 == SQL_SUCCESS_WITH_INFO) || (rc1 == SQL_ERROR)) {
    // Get the status records.
    i = 1;
    while ((rc2 = SQLGetDiagRec(SQL_HANDLE_STMT, hstmt, i, SqlState, &NativeError,
        Msg, sizeof(Msg), &MsgLen)) != SQL_NO_DATA) {
        DisplayError(SqlState, NativeError, Msg, MsgLen);
        i++;
    }
}
```

```

}

if ((rc1 == SQL_SUCCESS) || (rc1 == SQL_SUCCESS_WITH_INFO)) {
    // Process statement results, if any.
}

```

## Implementing SQLGetDiagRec and SQLGetDiagField

**SQLGetDiagRec** and **SQLGetDiagField** are implemented by the Driver Manager and each driver. The Driver Manager and each driver maintain diagnostic records for each environment, connection, statement, and descriptor handle, and free those records only when another function is called with that handle or the handle is freed.

Although both the Driver Manager and each driver must determine the first status record according to the rankings in “[Sequence of Status Records](#),” earlier in this chapter, the Driver Manager determines the final sequence of records.

SQLGetDiagRec and SQLGetDiagField do not post diagnostic records about themselves.

### *Diagnostic Handling Rules*

The following rules govern diagnostic handling in **SQLGetDiagRec** and **SQLGetDiagField**.

All ODBC components:

- Must not replace, alter, or mask errors or warnings received from another ODBC component.
- May add an additional status record when they receive a diagnostic message from another ODBC component. The added record must add real information value to the original message.

The ODBC component that directly interfaces a data source:

- Must prefix its vendor identifier, its component identifier, and the data source’s identifier to the diagnostic message it receives from the data source.
- Must preserve the data source’s native error code.
- Must preserve the data source’s diagnostic message.

Any ODBC component that generates an error or warning independent of the data source:

- Must supply the correct SQLSTATE for the error or warning.
- Must generate the text of the diagnostic message.
- Must prefix its vendor identifier and its component identifier to the diagnostic message.
- Must return a native error code, if one is available and meaningful.
- The ODBC component that interfaces with the Driver Manager:
- Must initialize the output arguments of **SQLGetDiagRec** and **SQLGetDiagField**.

- Must format and return the diagnostic information as output arguments of **SQLGetDiagRec** and **SQLGetDiagField** when that function is called.

One ODBC component other than the Driver Manager:

Must set the SQLSTATE based on the native error. For file-based drivers and DBMS-based drivers that do not use a gateway, the driver must set the SQLSTATE. For DBMS-based drivers that use a gateway, either the driver or a gateway that supports ODBC may set the SQLSTATE.

### ***Role of the Driver***

The driver checks for all errors and warnings not checked by the Driver Manager and orders status records that it generates. (An ODBC 2.x driver does not order status records.) This includes errors and warnings in data truncation, data conversion, syntax, and some state transitions. The driver might also check errors and warnings partially checked by the Driver Manager. For example, although the Driver Manager checks whether the value of *Operation* in **SQLSetPos** is legal, the driver must check whether it is supported.

The driver also maps *native errors*, or errors returned by the data source, to SQLSTATEs. For example, the driver might map a number of different native errors for illegal SQL syntax to SQLSTATE 42000 (Syntax error or access violation). The driver returns the native error number in the SQL\_DIAG\_NATIVE field of the status record. Driver documentation should show how errors and warnings are mapped from the data source to arguments in **SQLGetDiagRec** and **SQLGetDiagField**.

### **Argument Value Checks**

The Driver Manager checks the following types of arguments. Unless otherwise noted, the Driver Manager returns SQL\_ERROR for errors in argument values.

- Environment, connection, and statement handles usually cannot be null pointers. The Driver Manager returns SQL\_INVALID\_HANDLE when it finds a null handle.
- Required pointer arguments, such as *OutputHandlePtr* in **SQLAllocHandle** and *CursorName* in **SQLSetCursorName**, cannot be null pointers.
- Option flags that do not support driver-specific values must be a legal value. For example, *Operation* in **SQLSetPos** must be SQL\_POSITION, SQL\_REFRESH, SQL\_UPDATE, SQL\_DELETE, or SQL\_ADD.
- Option flags must be supported in the version of ODBC supported by the driver. For example, *InfoType* in **SQLGetInfo** cannot be SQL\_ASYNC\_MODE (introduced in ODBC 3.0) when calling an ODBC 2.0 driver.
- Column and parameter numbers must be greater than 0 or greater than or equal to 0, depending on the function. The driver must check the upper limit of these argument values based on the current result set or SQL statement.
- Length/indicator arguments and data buffer length arguments must contain appropriate values. For example, the argument that specifies the length of a table name in **SQLColumns** (*NameLength3*)

must be `SQL_NTS` or a value greater than 0; *BufferLength* in **SQLDescribeCol** must be greater than or equal to 0. The driver might also need to check these arguments. For example, it might check that *NameLength3* is less than or equal to the maximum length of a table name in the data source.

### State Transition Checks

The Driver Manager checks that the state of the environment, connection, or statement is appropriate for the function being called. For example, a connection must be in an allocated state when **SQLConnect** is called; a statement must be in a prepared state when **SQLExecute** is called. The Driver Manager returns `SQL_ERROR` for state transition errors.

### General Error Checks

The Driver Manager checks one general error. It always returns `SQL_ERROR` when it encounters this error:

The function must be supported by the driver.

### Driver Manager Error and Warning Checks

The Driver Manager completely or partially implements a number of functions and therefore checks for all or some of the errors and warnings in those functions.

- The Driver Manager implements **SQLDataSources** and **SQLDrivers**, and checks for all errors and warnings in these functions.
- The Driver Manager checks whether a driver implements **SQLGetFunctions**. If the driver does not implement **SQLGetFunctions**, the Driver Manager implements and checks for all errors and warnings in it.
- The Driver Manager partially implements **SQLAllocHandle**, **SQLConnect**, **SQLDriverConnect**, **SQLBrowseConnect**, **SQLFreeHandle**, **SQLGetDiagRec**, and **SQLGetDiagField** and checks for some errors in these functions. It may return the same errors as the driver for some of these functions because both perform similar operations. For example, the Driver Manager or driver may return `SQLSTATE IM008` (Dialog failed) if either one is unable to display a login dialog box for **SQLDriverConnect**.

## Diagnostic Handling Examples

The following examples show how various components in an ODBC connection might generate diagnostic messages and how various drivers might return diagnostics to the application with **SQLGetDiagRec**.

### File-Based Driver Diagnostic Example

A file-based driver acts both as an ODBC driver and as a data source. It can therefore generate errors and warnings both as a component in an ODBC connection and as a data source. Because it also is the component that interfaces with the Driver Manager, it formats and returns arguments for **SQLGetDiagRec**.

For example, if a Microsoft driver for dBASE could not allocate sufficient memory, it might return the following values from **SQLGetDiagRec**:

```
SQLSTATE:      "HY001"
Native Error:  42052
Diagnostic Msg: "[Microsoft][ODBC dBASE Driver]Unable to allocate sufficient
memory."
```

Because this error was not related to the data source, the driver only added prefixes to the diagnostic message for the vendor ([Microsoft]) and the driver ([ODBC dBASE Driver]).

If the driver could not find the file EMPLOYEE.DBF, it might return the following values from **SQLGetDiagRec**:

```
SQLSTATE:      "42S02"
Native Error:   -1305
Diagnostic Msg: "[Microsoft][ODBC dBASE Driver][dBASE]No such table or object"
```

Because this error was related to the data source, the driver added the file format of the data source ([dBASE]) as a prefix to the diagnostic message. Because the driver was also the component that interfaced with the data source, it added prefixes for the vendor ([Microsoft]) and the driver ([ODBC dBASE Driver]).

## DBMS-Based Driver Diagnostic Example

A DBMS-based driver sends requests to a DBMS and returns information to the application through the Driver Manager. Because the driver is the component that interfaces with the Driver Manager, it formats and returns arguments for **SQLGetDiagRec**.

For example, if a Microsoft driver for DEC's Rdb using SQL/Services encountered an invalid cursor name, it might return the following values from **SQLGetDiagRec**:

```
SQLSTATE:      "34000"
Native Error:   0
Diagnostic Msg: "[Microsoft][ODBC Rdb Driver]Invalid cursor name:
EMPLOYEE_CURSOR."
```

Because the error occurred in the driver, it added prefixes to the diagnostic message for the vendor ([Microsoft]) and the driver ([ODBC Rdb Driver]).

If the DBMS could not find the table EMPLOYEE, the driver might format and return the following values from **SQLGetDiagRec**:

```
SQLSTATE:      "42S02"
Native Error:   -1
Diagnostic Msg: "[Microsoft][ODBC Rdb Driver][Rdb] %SQL-F-RELNOTDEF, Table
EMPLOYEE "
               "is not defined in schema."
```

Because the error occurred in the data source, the driver added a prefix for the data source identifier ([Rdb]) to the diagnostic message. Because the driver was the component that interfaced with the data source, it added prefixes for its vendor ([Microsoft]) and identifier ([ODBC Rdb Driver]) to the diagnostic message.



## Gateways Diagnostic Example

In a gateway architecture, a driver sends requests to a gateway that supports ODBC. The gateway sends the requests to a DBMS. Because it is the component that interfaces with the Driver Manager, the driver formats and returns arguments for **SQLGetDiagRec**.

For example, if DEC based a gateway to Rdb on Microsoft Open Data Services, and Rdb could not find the table EMPLOYEE, the gateway might generate this diagnostic message:

```
"[42S02][-1][DEC][ODS Gateway][Rdb]%SQL-F-RELNOTDEF, Table EMPLOYEE is not
defined "
  "in schema."
```

Because the error occurred in the data source, the gateway added a prefix for the data source identifier ([Rdb]) to the diagnostic message. Because the gateway was the component that interfaced with the data source, it added prefixes for its vendor ([DEC]) and identifier ([ODS Gateway]) to the diagnostic message. Note that it also added the SQLSTATE value and the Rdb error code to the beginning of the diagnostic message. This permitted it to preserve the semantics of its own message structure and still supply the ODBC diagnostic information to the driver. The driver parses the error information attached to the error statement by the gateway.

Because the gateway driver is the component that interfaces with the Driver Manager, it would use the preceding diagnostic message to format and return the following values from **SQLGetDiagRec**:

```
SQLSTATE:      "42S02"
Native Error:  -1
Diagnostic Msg: "[DEC][ODS Gateway][Rdb]%SQL-F-RELNOTDEF, Table EMPLOYEE is not
defined in schema."
```

## Driver Manager Diagnostic Example

The Driver Manager can also generate diagnostic messages. For example, if an application passed an invalid direction option to **SQLDataSources**, the Driver Manager might format and return the following values from **SQLGetDiagRec**:

```
SQLSTATE:      "HY103"
Native Error:  0
Diagnostic Msg: "[Microsoft][ODBC Driver Manager]Direction option out of range"
```

Because the error occurred in the Driver Manager, it added prefixes to the diagnostic message for its vendor ([Microsoft]) and its identifier ([ODBC Driver Manager]).

## Chapter 16: Overview of Interoperability

*Interoperability* is the ability of a single application to operate with many different DBMSs. The need to write generic, interoperable applications was one of the major factors leading to the development of ODBC. However, interoperability is not a simple path followed from “not interoperable” to “completely interoperable.” The path has many branches and each requires tradeoffs among features, speed, code complexity, and development time.

The process of writing an interoperable application follows several steps:

1. Deciding whether the application will use ODBC.
2. Choosing a level of interoperability and deciding which tradeoffs are necessary to reach that level.
3. Writing interoperable code and testing it as fully as possible.

Before continuing, it should be noted that interoperability is primarily the domain of the application writer. Drivers are designed to work with a single DBMS and, by definition, are not interoperable. They play a role in interoperability by correctly implementing and exposing ODBC over a single DBMS.

### Is ODBC the Answer?

Before delving into the question of interoperability, consider the following question: Should the application use ODBC at all? This might seem a strange question to ask in a guide to ODBC, but it is, in fact, a legitimate one. ODBC was not designed to completely replace native database APIs, nor was it designed to provide database access in all circumstances. It was designed to provide a common interface to databases and was intended to free application programmers from having to learn about and maintain links to multiple databases.

Custom applications are prime candidates for native database APIs. The main reason is that custom applications often work with a single DBMS and have no need to be interoperable. Native database APIs might do a better job than ODBC of exposing the capabilities of a particular DBMS and might expose capabilities not exposed by ODBC. Furthermore, because the developers of custom applications are usually familiar with the native database API for their DBMS, there is little reason to learn ODBC. However, it is interesting to note that for some DBMSs, ODBC is the native database API.

So which applications are candidates for ODBC? The best candidates are applications that work with more than one DBMS. This includes virtually all generic and vertical applications. It also includes a number of custom applications. For example, custom applications that use several different DBMSs are much easier and cleaner to write with ODBC than with multiple native APIs. And custom applications written with ODBC are much easier to migrate as a company moves from one DBMS to another or deploys the same application against different DBMSs.

### Choosing a Level of Interoperability

Assuming the application will use ODBC, the next step is to determine what level of interoperability is required. The basic level of interoperability is generally a function of the application type: Custom

applications tend not to be interoperable, vertical applications tend to be interoperable among a limited number of DBMSs, and generic applications tend to be interoperable among all DBMSs.

## **Custom Applications**

Custom applications typically perform a specific task for a few DBMSs. For example, an application might retrieve data from a single DBMS and generate a report, or it might transfer data among several DBMSs. What these applications have in common is that these DBMSs are known before the application is written and are unlikely to change over the life of the application.

The custom application thus requires little or no interoperability. The application developer can choose a single driver for each DBMS and code directly to those drivers. The application can safely contain driver-specific code to exploit the capabilities of those drivers and might even make calls to the native database API to use functionality not supported by ODBC.

The major interoperability concern of most custom applications is whether the target DBMSs will change in the future. If so, this process can be simplified by writing more interoperable code to start with. However, such changing of DBMSs is rare and generally entails a large amount of work. Because of this, developers of custom applications rarely choose to increase interoperability at the expense of functionality, because they can recode that functionality when they change DBMSs.

## **Vertical Applications**

Vertical applications typically perform a well-defined task against a single DBMS. For example, an order entry application tracks the orders in a company. What these types of applications have in common is that the database schema is usually designed by the application developer and, while the application might work with a number of different DBMSs, it works with a single DBMS for a single customer.

Because vertical applications usually require certain functionality, such as scrollable cursors or transactions, they rarely support all DBMSs. Instead, they tend to be highly interoperable among a limited set of DBMSs. Typically, vertical application developers choose to support those DBMSs that represent a large fraction of the market and ignore the rest. They might even choose to support specific drivers for those DBMSs to reduce their testing and product support costs.

Because vertical applications can support a known set of DBMSs, they sometimes contain driver- or DBMS-specific code. However, such code is best kept to a minimum, because it requires extra time to maintain.

## **Generic Applications**

Generic applications sometimes perform a hard-coded task, such as a spreadsheet retrieving data from a database. They might also perform a variety of user-defined tasks, such as a generic query application allowing the user to enter and execute an SQL statement. What generic applications have in common is that they must work with a variety of different DBMSs and that the developer does not know beforehand what these DBMSs will be.

Therefore, generic applications need to be highly interoperable. The developer must make many choices, trading off interoperability for features, and must write code that expects drivers to support a wide range of functionality. While generic applications might be tuned to work with popular DBMSs, they rarely contain driver- or DBMS-specific code.

## Determining the Target DBMSs and Drivers

The next question to consider is, what are the target DBMSs for the application, and what drivers are available that support those DBMSs? Because generic applications tend to be highly interoperable, the question of target DBMSs is most applicable to custom and vertical applications. However, the question of target drivers applies to all applications, because drivers vary widely in speed, quality, feature support, and availability. Also, if drivers are to be redistributed with the application, the cost and availability of licensing plans needs to be considered.

For many custom applications, the target DBMSs are obvious: They are existing DBMSs that the application is designed to access. DBMSs to which future migration is planned should also be considered. However, the major question for these applications is which driver or drivers to use with them. For other custom applications — those which are not designed to access an existing DBMS — the target DBMSs can be chosen based on feature support, concurrent user support, driver availability, and affordability.

For vertical applications, the target DBMSs are usually chosen based on feature support, driver availability, and market. For example, a vertical application designed for small businesses must target DBMSs that are affordable to those businesses; a vertical application designed as an add-on to existing DBMSs must target widely used DBMSs.

When choosing target DBMSs, the differences between desktop and server databases should be considered. Desktop databases such as dBASE, Paradox, and Btrieve are less powerful than server databases. Because they are generally accessed through the less-powerful SQL engines found in most file-based drivers, they often lack full transaction support, support fewer concurrent users, and have limited SQL. However, they are inexpensive and have a large installed base.

Server databases such as Oracle, DB2, and SQL Server provide full transaction support, support many concurrent users, and have rich SQL. They are much more expensive and have a smaller installed base. On the other hand, software prices tend to be higher, somewhat offsetting a smaller potential market.

Thus, target DBMSs sometimes can be chosen based on the features required by the application and the application's target market. For example, an order entry system for large corporations might not target desktop databases, because these lack adequate transaction support. A similar system designed for small businesses might exclude most server databases on the basis of cost. And developers of generic applications might target both, but avoid using the advanced features found in server databases.

## Considering Database Features to Use

After the basic level of interoperability is known, the database features used by the application must be considered. For example, what SQL statements will the application execute? Will the application use scrollable cursors? Transactions? Procedures? Long data? For ideas about what features might not be supported by all DBMSs, see the `SQLGetInfo`, `SQLSetConnectAttr`, and `SQLSetStmtAttr` function descriptions in the Part II PDF file, "ODBC API Reference" available on the Solid Web site, and Appendix C, "SQL Minimum Grammar" in the **SOLID Programmer Guide**. The features required by an application might eliminate some DBMSs from the list of target DBMSs. They might also show that the application can easily target many DBMSs.

For example, if the required features are simple, they can usually be implemented with a high degree of interoperability. An application that executes a simple **SELECT** statement and retrieves results with a

forward-only cursor is likely to be highly interoperable by virtue of its simplicity: Almost all drivers and DBMSs support the functionality it needs.

On the other hand, if the required features are more complex, such as scrollable cursors, positioned update and delete statements, and procedures, tradeoffs must often be made. There are several possibilities:

- **Lower interoperability, more features.** The application includes the features, but works only with DBMSs that support them.
- **Higher interoperability, fewer features.** The application drops the features, but works with more DBMSs.
- **Higher interoperability, optional features.** The application includes the features, but makes them available only with those DBMSs that support them.
- **Higher interoperability, more features.** The application uses the features with DBMSs that support them and emulates them for DBMSs that do not.

The first two cases are relatively simple to implement, because the features are used either with all supported DBMSs or with none. The latter two cases, on the other hand, are more complex. It is necessary in both cases to check whether the DBMS supports the features, and in the last case to write a potentially large amount of code to emulate these features. Thus, these schemes are likely to require more development time and may be slower at run time.

Consider a generic query application that can connect to a single data source. The application accepts a query from the user and displays the results in a window. Now suppose this application has one feature that allows users to display the results of multiple queries simultaneously; that is, they can execute a query and look at some of the results, execute a different query and look at some of its results, and then return to the first query. This presents an interoperability problem because some drivers support only a single active statement.

The application has a number of choices, based on what the driver returns for the `SQL_MAX_CONCURRENT_ACTIVITIES` option in **SQLGetInfo**:

- **Always support multiple queries.** After connecting to a driver, the application checks the number of active statements. If the driver supports only one active statement, the application closes the connection and informs the user that the driver does not support required functionality. The application is easy to implement and has full functionality, but has lower interoperability.
- **Never support multiple queries.** The application drops the feature altogether. It is easy to implement and has high interoperability, but has less functionality.
- **Support multiple queries only if the driver does.** After connecting to a driver, the application checks the number of active statements. The application allows the user to start a new statement when one is already active only if the driver supports multiple active statements. The application has higher functionality and interoperability, but is harder to implement.

- **Always support multiple queries and emulate them when necessary.** After connecting to a driver, the application checks the number of active statements. The application always allows the user to start a new statement when one is already active; if the driver supports only one active statement, the application opens an additional connection to that driver and executes the new statement on that connection. The application has full functionality and high interoperability, but is harder to implement.

## Length of the Product Cycle

The final question about interoperability is time. Developing an interoperable application usually takes longer than developing a noninteroperable one. The reason is that the application must check DBMS capabilities, perform the same tasks differently for different DBMSs, work around functionality supported by some DBMSs but not others, and so on.

In addition to development time, product lifetime must be considered. If the application is designed to be used once, such as an application that transfers data when migrating from one DBMS to another, there is no point in making it interoperable. The application will be used once and discarded.

On the other hand, if the application will exist for a long time, it might be easier to maintain as an interoperable application. This is true even for custom applications that have a single DBMS as a target. The reason is that interoperable code uses a limited subset of database features. The driver is required to keep those features available, even in the face of changes to the underlying DBMS. Thus, interoperable code can shift the burden of coping with changes to the DBMS from the application developer to the driver developer.

## Writing an Interoperable Application

Whenever an application uses the same code against more than one driver, that code must be interoperable among those drivers. In most cases, this is an easy task. For example, the code to fetch rows with a forward-only cursor is the same for all drivers. In some cases, this can be more difficult. For example, the code to construct identifiers for use in SQL statements needs to consider identifier case, quoting, and one-, two-, and three-part naming conventions.

In general, interoperable code must cope with problems of feature support and feature variability. *Feature support* refers to whether or not a particular feature is supported. For example, not all DBMSs support transactions, and interoperable code must work correctly regardless of transaction support. *Feature variability* refers to variation in the manner in which a particular feature is supported. For example, catalog names are placed at the start of identifiers in some DBMSs and at the end of identifiers in others.

Applications can deal with feature support and feature variability at design time or at run time. To deal with feature support and variability at design time, a developer looks at the target DBMSs and drivers and makes sure that the same code will be interoperable among them. This is generally the way in which applications with low or limited interoperability deal with these problems.

For example, if the developer guarantees that a vertical application will work only with four particular DBMSs, and each of those DBMSs supports transactions, the application does not need code to check for transaction support at run time. It can always assume transactions are available because of the design-time decision to use only four DBMSs, each of which supports transactions.

To deal with feature support and variability at run time, the application must test for different capabilities at run time and act accordingly. This is generally the way in which highly interoperable applications deal with these problems. For feature support problems, this means writing code that makes the feature optional or writing code that emulates the feature when it is not available. For feature variability problems, this means writing code that supports all possible variations.

### ***Checking Feature Support and Variability***

To check feature support and variability, applications generally call **SQLGetInfo**, **SQLGetFunctions**, and **SQLGetTypeInfo**. A good starting place is the driver's API and SQL grammar conformance levels. These describe broad levels of feature support. The application can then call **SQLGetInfo** with other options to determine the support or variability of features it needs, **SQLGetFunctions** to determine whether functions it needs beyond the returned conformance level are supported, and **SQLGetTypeInfo** to determine what SQL data types are supported.

An application can determine whether a statement or connection attribute is supported by calling **SQLSetStmtAttr** or **SQLSetConnectAttr** with that attribute. If the function returns **SQL\_SUCCESS** or **SQL\_SUCCESS\_WITH\_INFO**, the attribute is supported; if it returns **SQL\_ERROR** and **SQLSTATE HYC00** (Optional feature not implemented), the attribute is not supported.

Applications can also determine a limited amount of information before connecting to the driver by calling **SQLDrivers**.

## **Features to Watch For**

This section describes a number of features that application developers often take for granted. In fact, these features vary widely in support and manner of support among DBMSs; failure to code for them is likely to cause problems in interoperable applications.

This section does not list all features that application developers need to consider. For that information, see the **SQLGetInfo**, **SQLSetStmtAttr**, and **SQLSetConnectAttr** in the Part II PDF file, "ODBC API Reference" available on the Solid Web site, and Appendix C, "SQL Minimum Grammar" in the **SOLID Programmer Guide** and the sections of this manual that discuss each feature.

### ***Version Number***

There are several versions of ODBC, each with different features. An application determines which ODBC version the Driver Manager and a particular driver support by calling **SQLGetInfo** with the **SQL\_ODBC\_VER** and **SQL\_DRIVER\_ODBC\_VER** options.

### ***Multiple Active Statements and Connections***

Some drivers and DBMSs limit the number of statements and connections that can be active at one time. These numbers can be as small as one. For more information, see the **SQL\_MAX\_CONCURRENT\_ACTIVITIES** and **SQL\_MAX\_DRIVER\_CONNECTIONS** options in the **SQLGetInfo** in the Part II PDF file, "ODBC API Reference" available on the Solid Web site, and "[Statement Handles](#)" and "[Connection Handles](#)" in Chapter 4, "ODBC Fundamentals."

### ***Transaction Support in DBMSs***

Some databases, especially desktop databases such as dBASE, Paradox, and Btrieve, do not support transactions. Even among databases that support transactions, there is variation in what kinds of SQL statements can be in a transaction. For more information, see the **SQL\_TXN\_CAPABLE** option in the

SQLGetInfo in the Part II PDF file, “ODBC API Reference” available on the Solid Web site function description.

### ***Commit and Rollback Behavior***

A common behavior among server DBMSs is to close cursors and discard prepared statements when a statement is committed or rolled back. Desktop databases are more likely to keep cursors open and keep prepared statements. For more information, see the SQL\_CURSOR\_COMMIT\_BEHAVIOR and SQL\_CURSOR\_ROLLBACK\_BEHAVIOR options in the SQLGetInfo in the Part II PDF file, “ODBC API Reference” available on the Solid Web site [“Effect of Transactions on Cursors and Prepared Statements”](#) in Chapter 14, “Transactions.”

### ***NOT NULL in CREATE TABLE Statements***

Some databases, and especially desktop databases, do not support the **NOT NULL** column constraint in **CREATE TABLE** statements. For more information, see the SQL\_NON\_NULLABLE\_COLUMNS option in the SQLGetInfo function description.

### ***Supported Data Types***

The data types supported by DBMSs vary considerably. An application can determine the names and characteristics of supported data types by calling **SQLGetTypeInfo**. Because of wide variation in data type names, the application must use the data type names returned by **SQLGetTypeInfo** in **CREATE TABLE** statements. For more information, see [“Data Types in ODBC”](#) in Chapter 4, “ODBC Fundamentals.”

### ***ODBC SQL Grammar***

Interoperable applications should always use the ODBC SQL grammar in SQL statements. However, considerable variation is possible even within this grammar. For more information, see [“Interoperability of SQL Statements”](#) in Chapter 8, “SQL Statements.”

### ***Batch Processing***

Support for batches of SQL statements is not widespread, so interoperable applications should use them conditionally, or not at all. For more information, see [“Executing Batches”](#) in Chapter 9, “Executing Statements.”

## **Testing Interoperable Applications**

Testing interoperable applications is at best a time-consuming business, and at worst impossible because new drivers continually appear on the market. However, a reasonable degree of testing is possible. Applications with limited or low interoperability need only be tested against those drivers they are guaranteed to support. However, they must be fully tested against these drivers.

Highly interoperable applications cannot be tested practically against all drivers. The best that most application developers can do is to test them fully against a small number of drivers and cursorily against several more. Tested drivers should include the most popular drivers for the most popular DBMSs in the application’s market; if the market covers all DBMSs, then drivers for both desktop and server DBMSs should be tested.

One of the problems in testing ODBC applications is the number of components involved: the application itself, the Driver Manager, the driver, the DBMS, and possibly network software or gateways. Applications can make it easier to track errors by posting the error messages returned by ODBC functions through



**SQLGetDiagField** and **SQLGetDiagRec**. These messages identify the manufacturer and component in which errors occur. For more information, see “[Chapter 15: Overview of Diagnostics](#).”

## Chapter 17: Overview of Programming Considerations

This chapter briefly discusses a number of topics related to writing ODBC applications and drivers.

### Multithreading

On multithread operating systems, drivers must be thread safe. That is, it must be possible for applications to use the same handle on more than one thread. How this is achieved is driver-specific, and it is likely that drivers will serialize any attempts to concurrently use the same handle on two different threads.

Applications commonly use multiple threads instead of asynchronous processing. The application creates a separate thread and calls an ODBC function on it, and then continues processing on the main thread. Rather than having to continually poll the asynchronous function, as is the case when the `SQL_ATTR_ASYNC_ENABLE` statement attribute is used, the application can simply let the newly created thread finish.

Functions that accept a statement handle and are running on one thread can be canceled by calling **SQLCancel** with the same statement handle from another thread. Although drivers should not serialize the use of **SQLCancel** in this manner, there is no guarantee that calling **SQLCancel** will actually cancel the function running on the other thread.

### Alignment

The alignment issues in an ODBC application are generally no different than they are in any other application. That is, most ODBC applications have few or no problems with alignment. The penalties for not aligning addresses vary with the hardware and operating system, and may be as minor as a slight performance penalty or as major as a fatal run-time error. Thus, ODBC applications — and portable ODBC applications in particular — should be careful to align data properly.

One place where ODBC applications encounter alignment issues is when they allocate a large block of memory and bind different parts of that memory to the columns in a result set. This is most likely to occur when a generic application must determine the shape of a result set at run time and allocate and bind memory accordingly.

For example, suppose an application executes a **SELECT** statement entered by the user and fetches the results from this statement. Because the shape of this result set is not known when the program is written, the application must determine the type of each column after the result set is created and bind memory accordingly. The easiest way to do this is to allocate a large block of memory and bind different addresses in that block to each column. To access the data in a column, the application casts the memory bound to that column.

The following diagram shows a sample result set and how a block of memory might be bound to it using the default C data type for each SQL data type. Each "X" represents a single byte of memory. Note that this example only shows the data buffers that are bound to the columns. This is done for simplicity. In actual code, the length/indicator buffers must also be aligned.

Assuming the bound addresses are stored in the *Address* array, the application uses the following expressions to access the memory bound to each column:

```
(SQLCHAR *) Address[0]  
(SQLSMALLINT *) Address[1]  
(SQLINTEGER *) Address[2]
```

Notice that the addresses bound to the second and third columns start on odd-numbered bytes and that the address bound to the third column is not divisible by four, which is the size of an SDWORD. On some machines, this will not be a problem, on others, it will cause a slight performance penalty, on still others, it will cause a fatal run-time error. A better solution would be to align each bound address on its natural alignment boundary. Assuming this is 1 for a UCHAR, 2 for an SWORD, and 4 for an SDWORD, this would give the following, where an "X" represents a byte of memory that is used and an "O" represents a byte of memory that is unused:

While this solution does not use all of the application's memory, it does not encounter any alignment problems. Unfortunately, it takes a fair amount of code to implement this solution, as each column must be aligned individually according to its type. A simpler solution is to align all columns on the size of the largest alignment boundary, which is 4 in this case:

Although this solution leaves larger holes, the code to implement it is relatively simple and fast. In most cases, this offsets the penalty paid in unused memory. For an example that uses this method, see "[Using SQLBindCol](#)" in Chapter 10, "Retrieving Results (Basic)."

## Unicode

Unicode is a method of software character encoding that treats all characters as having a fixed width of two bytes. This method is used as an alternative to Windows ANSI character encoding, which because it represents characters in one byte, is limited to 256 characters. Because Unicode can represent over 65,000 characters, it accommodates many languages whose characters are not represented in ANSI encoding.

Unicode does not require the use of codepages, which ANSI uses to accommodate a limited set of languages. Unicode is an improvement upon the Double-Byte Character Set (DBCS), which uses a mixture of 8-bit and 16-bit characters and still requires codepages. For more information about the Unicode standard, see <http://www.cam.spyglass.com/unicode.html>.

The ODBC 3.5 (or higher) Driver Manager is Unicode-enabled. This affects two major areas: function calls and string data types. The Driver Manager maps function string arguments and string data as required by the application and driver, both of which can be either Unicode-enabled or ANSI-enabled. These two areas are discussed in detail in the sections, "[Unicode Function Arguments](#)" and "Unicode Data."

The ODBC 3.5 (or higher) Driver Manager supports the use of a Unicode driver with both a Unicode application and an ANSI application. It also supports the use of an ANSI driver with an ANSI application. The Driver Manager provides limited Unicode-to-ANSI mapping for a Unicode application working with an ANSI driver.

## Unicode Function Arguments

The ODBC 3.5 (or higher) Driver Manager supports both ANSI and Unicode versions of all functions that accept pointers to character strings or SQLPOINTER in their arguments. The Unicode functions are implemented as functions (with a suffix of "W"), not as macros. The ANSI functions (which can be called with or without a suffix of "A") are identical to the current ODBC API functions.

Unicode functions that always return or take strings or length arguments are passed as count-of-characters. For functions that return length information for server data, the display size and precision are described in number of characters. When a length (transfer size of the data) could refer to string or non-string data, the length is described in octet lengths. For example, **SQLGetInfoW** will still take the length as count-of-bytes, but **SQLExecDirectW** will use count-of-characters.

The following is a list of the ODBC API functions that support both Unicode (W) and ANSI (A) versions.

|                            |                            |
|----------------------------|----------------------------|
| <b>SQLBrowseConnect</b>    | <b>SQLGetDiagField</b>     |
| <b>SQLColAttribute</b>     | <b>SQLGetDiagRec</b>       |
| <b>SQLColAttributes</b>    | <b>SQLGetInfo</b>          |
| <b>SQLColumnPrivileges</b> | <b>SQLGetStmtAttr</b>      |
| <b>SQLColumns</b>          | <b>SQLNativeSQL</b>        |
| <b>SQLConnect</b>          | <b>SQLPrepare</b>          |
| <b>SQLDataSources</b>      | <b>SQLPrimaryKeys</b>      |
| <b>SQLDescribeCol</b>      | <b>SQLProcedureColumns</b> |
| <b>SQLDriverConnect</b>    | <b>SQLProcedures</b>       |
| <b>SQLDrivers</b>          | <b>SQLSetConnectAttr</b>   |
| <b>SQLError</b>            | <b>SQLSetConnectOption</b> |
| <b>SQLExecDirect</b>       | <b>SQLSetCursorName</b>    |
| <b>SQLForeignKeys</b>      | <b>SQLSetDescField</b>     |
| <b>SQLGetConnectAttr</b>   | <b>SQLSetStmtAttr</b>      |
| <b>SQLGetConnectOption</b> | <b>SQLSpecialColumns</b>   |
| <b>SQLGetCursorName</b>    | <b>SQLStatistics</b>       |
| <b>SQLGetDescField</b>     | <b>SQLTablePrivileges</b>  |
| <b>SQLGetDescRec</b>       | <b>SQLTables</b>           |

The following is a list of the ODBC Installer and ODBC Translator functions that require both Unicode (W) and ANSI (A) versions.

|                               |                                |
|-------------------------------|--------------------------------|
| <b>SQLConfigDataSource</b>    | <b>SQLInstallDriver</b>        |
| <b>SQLCreateDataSource</b>    | <b>SQLInstallDriverManager</b> |
| <b>SQLDataSourceToDriver</b>  | <b>SQLInstallODBC</b>          |
| <b>SQLDriverToDataSource</b>  | <b>SQLRemoveDSNFromINI</b>     |
| <b>SQLGetAvailableDrivers</b> | <b>SQLValidDSN</b>             |
| <b>SQLGetInstalledDrivers</b> | <b>SQLWriteDSNToINI</b>        |
| <b>SQLGetTranslator</b>       |                                |

**Note** Deprecated functions have Unicode-to-ANSI mapping support, because the ODBC 3.x Driver Manager supports recompiling ODBC 2.x application with the **UNICODE #define**.

## Unicode Applications

You can recompile an application as a Unicode application in one of two ways:

- Include the Unicode *#define* contained in the SQLUCODE.H header file in the application.
- Compile the application with the compiler's Unicode option (note that this option will be different for different compilers).

To convert an ANSI application to a Unicode application, write the application to store and pass Unicode data. In addition, calls to functions that support SQLPOINTER arguments must be converted to use count of bytes.

Once an application is compiled as a Unicode application, if the application calls an ODBC API function (without a suffix), the Driver Manager recognizes the application as a Unicode application and converts the function call to a Unicode function (with the "W" suffix) if the underlying driver supports Unicode. When an ANSI application makes a function call without a suffix, the Driver Manager converts it to ANSI if the underlying driver supports ANSI. If both the application and the driver support the same character encoding, the driver manager passes the calls through to the driver (with certain exceptions for ANSI applications.)

An application can call both Unicode functions (with the "W" suffix) and ANSI functions (with or without the "A" suffix). Unicode and ANSI function calls can be mixed. If the cursor library is to be used, however, Unicode and ANSI function calls cannot be mixed. The cursor library is either Unicode or ANSI, not a mixture.

An application can be written such that it can be compiled as either a Unicode application or an ANSI application. In this case, character data types can be declared as SQL\_C\_TCHAR. This is a macro that inserts SQL\_C\_WCHAR if the application is compiled as a Unicode application or SQL\_C\_CHAR if it is compiled as an ANSI application. The application programmer must be careful of functions that take SQLPOINTER as their argument, since the size of the length argument will change (for string data types) depending upon whether the application is ANSI or Unicode.

A function can be called in one of three ways: as a Unicode-only function call (with the "W" suffix), as an ANSI-only function call (with the "A" suffix), or as the ODBC function call with no suffix. The arguments to the three forms of a function are identical. Only those functions with SQLCHAR \* arguments or SQLPOINTER arguments that point to strings require Unicode and ANSI forms. For functions that have arguments that can be declared as a character type, such as **SQLBindCol** or **SQLGetData** (which do not have Unicode and ANSI forms), the argument can be declared as the Unicode type, the ANSI type, or in the case of a C type argument, the SQL\_C\_TCHAR macro. For more information, see "Unicode Data."

An application can be written as a Unicode application even if no Unicode drivers are available for it to work with. The Driver Manager will map Unicode functions and data types to ANSI. Note that there are some restrictions to the Unicode to ANSI mappings that can be performed. The existence of a Unicode driver for the Unicode application to work with will result in better performance and will remove the restrictions inherent in the Unicode to ANSI mappings.

## Unicode Drivers

Whether a driver should be a Unicode driver or an ANSI driver depends entirely on the nature of the data source. If the data source supports Unicode data, the driver should be a Unicode driver. If the data source only supports ANSI data, the driver should remain an ANSI driver.

A Unicode driver must export **SQLConnectW** in order to be recognized as a Unicode driver by the Driver Manager.

A Unicode driver must accept Unicode functions (with a suffix of "W") and store Unicode data. It can also accept ANSI functions, but is not required to. (The Driver Manager does not pass an ANSI function call with the "A" suffix to the driver, but converts it to an ANSI function call without the suffix, then passes it to the driver.)

A Unicode driver must be able to return result sets in either Unicode or ANSI, depending on the application's binding. If an application binds to `SQL_C_CHAR`, then the Unicode driver must convert `SQL_WCHAR` data to `SQL_CHAR`. Note that the driver manager will map `SQL_C_WCHAR` to `SQL_C_CHAR` for ANSI drivers, but does no mapping for Unicode drivers.

## Function Mapping in the Driver Manager

The driver manager supports two entry points for functions that take string arguments. The undecorated function (**SQLDriverConnect**) is the ANSI form of the function. The Unicode form is decorated with a "W" (**SQLDriverConnectW**.)

The ODBC header file also supports functions decorated with an "A" (**SQLDriverConnectA**) for the convenience of mixed ANSI/Unicode applications. Calls made to the "A" functions are actually calls into the undecorated entry point (**SQLDriverConnect**.)

If the application is compiled with the `_UNICODE #define`, then the ODBC header file will map undecorated function calls (**SQLDriverConnect**) to the Unicode version (**SQLDriverConnectW**.)

The Driver Manager recognizes a driver as a Unicode driver if **SQLConnectW** is supported by the driver.

If the driver is a Unicode driver, the Driver Manager makes function calls as follows:

- Passes a function without string arguments or parameters directly through to the driver.
- Passes Unicode functions (with the "W" suffix) directly through to the driver.
- Converts an ANSI function (with the "A" suffix) to a Unicode function (with the "W" suffix) by converting the string arguments into Unicode characters, and passes the Unicode function to the driver.

If the driver is an ANSI driver, the Driver Manager makes function calls as follows:

- Passes functions without string arguments or parameters directly through to the driver.

- Converts Unicode functions (with the "W" suffix) to an ANSI function call and passes it to the driver.
- Passes an ANSI function directly to the driver.

The Driver Manager is Unicode-enabled internally. As a result, the optimum performance is obtained by a Unicode application working with a Unicode driver, because the Driver Manager simply passes Unicode functions through to the driver. When an ANSI application is working with an ANSI driver, the Driver Manager must convert strings from ANSI to Unicode when processing some functions, such as **SQLDriverConnect**. After processing the function, the Driver Manager must then convert the Unicode string back to ANSI before sending the function to the ANSI driver.

An application should not modify or read its bound parameter buffers when the driver returns `SQL_STILL_EXECUTING` or `SQL_NEED_DATA`. The Driver Manager leaves the buffers bound to ANSI until the driver returns `SQL_SUCCESS`, `SQL_SUCCESS_WITH_INFO`, or `SQL_ERROR`. A multithreaded application should not gain access to any bound parameter values that another thread is executing an SQL statement on. The Driver Manager converts the data from Unicode to ANSI "in place", and the other thread may see ANSI data in these buffers while the driver is still processing the SQL statement. Applications that bind Unicode data to an ANSI driver must not bind two different columns to the same address.

## Unicode Data

SQL Unicode data types are provided to describe data that resides in Unicode natively on the DBMS. A C Unicode data type is provided to allow an application to bind data to a Unicode buffer. The Driver Manager can convert data from a Unicode C type (`SQL_C_WCHAR`) to make it function with an ANSI driver.

An ODBC 3.0, or 2.x application will always bind to the ANSI data types. For optimum performance, an ODBC 3.5 (or higher) application should bind to the ANSI data C type if the SQL column type is ANSI, and should bind to the Unicode C data type if the SQL column type is Unicode.

The SQL Unicode type indicators are `SQL_WCHAR`, `SQL_WVARCHAR`, and `SQL_WLONGVARCHAR`. `SQL_WCHAR` data has a fixed string length, while `SQL_WVARCHAR` has a variable length with a declared maximum and `SQL_WLONGVARCHAR` has a variable length with a maximum that depends on the data source.

The C Unicode type indicator is `SQL_C_WCHAR`. This is the default for each of the SQL Unicode type indicators. All of the SQL types can be converted to `SQL_C_WCHAR`, and `SQL_C_WCHAR` can be converted to all of the SQL types. An application can retrieve data in one of three ways:

- Retrieve the data as `SQL_C_CHAR`.
- Retrieve the data as `SQL_C_WCHAR`.
- Declare the data as `SQL_C_TCHAR`. This is a macro that inserts `SQL_C_WCHAR` if the application is compiled as a Unicode application or `SQL_C_CHAR` if it is compiled as an ANSI application.

SQL\_C\_TCHAR is declared in a function as follows:

```
SQLBindParameter(StatementHandle, 1, SQL_PARAM_INPUT, SQL_C_TCHAR, SQL_WCHAR,  
NameLen, 0, Name, 0, &Name)
```

When the application is compiled as a Unicode application, the *ValueType* argument would be changed from SQL\_C\_TCHAR to SQL\_C\_WCHAR. When the application is compiled as an ANSI application, the *ValueType* argument would be changed to SQL\_C\_CHAR.

Unicode drivers must still support ANSI data types, including SQL\_CHAR. If an application working with a non-Unicode driver binds to SQL\_WCHAR, the Driver Manager will map the SQL\_WCHAR data to SQL\_CHAR. If, on the other hand, an application working with a Unicode driver binds to SQL\_CHAR, the Driver Manager will not map the SQL\_CHAR data to SQL\_WCHAR. The Unicode driver must accept the SQL\_CHAR data.

The Driver Manager stores driver and DSN names in Unicode, and maps them to ANSI as needed. If a Unicode character cannot be mapped to an ANSI character (as can occur if characters from a codepage that is not the native codepage of the computer are used in driver and DSN names), the characters that could not be converted are represented by a default character supplied by the system.

## Translation DLLs

The application and data source often store data in different character sets. ODBC provides a generic mechanism that allows the driver to translate data from one character set to another. It consists of a DLL that implements the translation functions **SQLDriverToDataSource** and **SQLDataSourceToDriver** which is called by the driver to translate all data flowing between the data source and driver. This DLL can be written by the application developer, the driver developer, or a third party.

The translation DLL for a particular data source can be specified in the system information for that data source; for more information, see “Data Source Specification Key” in Chapter 20, “Configuring Data Sources” in the Part III PDF file, “Configuring and Installing ODBC software,” available on the Solid Web site. It can also be set at run time with the SQL\_ATTR\_TRANSLATE\_DLL and SQL\_ATTR\_TRANSLATE\_OPTION connection attributes.

The translation option is a value that can be interpreted only by a particular translation DLL. For example, if the translation DLL translates between different code pages, the option might give the numbers of the code pages used by the application and the data source. There is no requirement for a translation DLL to use a translation option.

After a translation DLL has been specified, the driver loads it and calls it to translate all data flowing between the application and data source. This includes all SQL statements and character parameters being sent to the data source and all character results, character metadata such as column names, and error messages retrieved from the data source. Note that connection data is not translated, as the translation DLL is not loaded until after the application has connected to the data source.

## Tracing

The ODBC Driver Manager has a trace facility that allows the sequence of function calls made by an ODBC application to be recorded and transcribed into a log file. Tracing is performed by a trace DLL that captures calls between the application and the Driver Manager, and the Driver Manager and the driver. This



method of tracing replaces the tracing performed by the ODBC 2.x Driver Manager and the tracing performed in ODBC 2.x by ODBC Spy.

### ***Trace DLL***

The DLL that performs tracing is one of the ODBC core components. The trace DLL is provided as a sample DLL in the ODBC component of the Microsoft Data Access SDK, so the registry entry, interface, and source code for the trace DLL are available. This DLL can be replaced by a trace DLL produced by either an ODBC user or a third-party vendor. A custom trace DLL should be given a different name than the original sample trace DLL. Trace DLLs must be installed in the system directory, or they will fail to load. The connection strings will not be passed to the trace DLL by the Driver Manager.

The trace DLL traces input arguments, output arguments, deferred arguments, return codes, and SQLSTATEs. When tracing is enabled, the Driver Manager calls the trace DLL at two points: once upon function entry (before argument validation) and again just before the function returns.

When an application calls a function, the Driver Manager calls a trace function in the trace DLL before calling the function in the driver or processing the call itself. Each ODBC function has a corresponding trace function (prefixed with "TRACE") that is identical to the ODBC function with the exception of the name. When the trace function is called, the trace DLL captures the input arguments and returns a return code. Because the trace DLL is called before the Driver Manager validates arguments, invalid function calls are traced, so state transition errors and invalid arguments are logged.

After calling the trace function in the trace DLL, the Driver Manager calls the ODBC function in the driver. It then calls **TraceReturn** in the trace DLL. This function takes two arguments: the value returned by the trace DLL for the trace function, and the return code returned by the driver to the Driver Manager for the ODBC function (or the value returned by the Driver Manager itself if it processed the function). The function uses the value returned for the trace function to manipulate captured input argument values. It writes the code returned for the ODBC function to the log file (or displays it dynamically, if that is enabled). It dereferences the output argument pointers, and logs the output argument values.

### ***Trace File***

An application specifies the trace file either by setting the **TraceFile** keyword in the ODBC.INI registry entry, or by calling **SQLSetConnectAttr** with the SQL\_ATTR\_TRACEFILE connection attribute. If the file does not exist when tracing is enabled, the Driver Manager will create the file. Each application should have its own dedicated trace file to avoid contention. An application can use more than one trace file; an application's setup program can provide the user with a choice of trace files. If tracing is enabled dynamically, an application can also display trace results, rather than logging to the trace file.

The trace file provides a log of each ODBC function call with the data types and values of all arguments. It logs all input functions, and all returned functions with return codes and error states.

In ODBC 3.x, parameters to connection functions are not provided to the trace DLL.

## **Enabling Tracing**

Tracing can be enabled in the following three ways:

- Set the Trace and TraceFile keywords in the ODBC.INI registry entry. This enables or disables tracing when SQLAllocHandle with a HandleType of SQL\_HANDLE\_ENV is called. These

options are set in the Tracing tab of the ODBC Data Source Administrator dialog box displayed during data source setup. For more information, see "Registry Entries for Data Sources" in Chapter 20, "Configuring Data Sources" in the Part III PDF file, "Configuring and Installing ODBC Software," available on the Solid Web site.

- Call **SQLSetConnectAttr** to set the **SQL\_ATTR\_TRACE** connection attribute to **SQL\_OPT\_TRACE\_ON**. This enables or disables tracing for the duration of the connection. For more information, see the **SQLSetConnectAttr** function description.
- Use **ODBCSharedTraceFlag** to turn tracing on or off dynamically. (For more information, see the next section, "Dynamic Tracing.")

## Dynamic Tracing

Tracing can be enabled or disabled at any point in an application run. This allows an application to trace any number of function calls.

The variable **ODBCSharedTraceFlag** is set to enable tracing dynamically. This variable is shared among all running copies of the Driver Manager. If any application sets this variable, tracing is enabled for all ODBC applications currently running. To turn tracing off when dynamic tracing is enabled, an application calls **SQLSetConnectAttr** to set **SQL\_ATTR\_TRACE** to **SQL\_TRACE\_OFF**. This call will turn tracing off for that application only. Applications that are linked with **ODBC32.LIB** can modify use of this variable. Trace data can be displayed in a real-time window, instead of the trace file, which must be opened after the ODBC session. Controls can be added to an application's screen to turn tracing on or off at will.

The trace DLL shipped with the ODBC 3.x SDK is not thread-safe. It is not guaranteed that the log file is written correctly if global tracing is enabled (the variable **ODBCSharedTraceFlag** is set) and more than one application writes to the trace file at the same time. This condition does not return an error.

## Driver-Specific Data Types, Descriptor Types, Information Types, Diagnostic Types, and Attributes

Drivers can allocate driver-specific values for the following:

- **SQL data type indicators.** These are used in *ParameterType* in **SQLBindParameter** and *DataType* in **SQLGetTypeInfo** and returned by **SQLColAttribute**, **SQLColumns**, **SQLDescribeCol**, **SQLGetTypeInfo**, **SQLDescribeParam**, **SQLProcedureColumns**, and **SQLSpecialColumns**.
- **Descriptor fields.** These are used in *FieldIdentifier* in **SQLColAttribute**, **SQLGetDescField**, and **SQLSetDescField**.
- **Diagnostic fields.** These are used in *DiagIdentifier* in **SQLGetDiagField** and **SQLGetDiagRec**.
- **Information types.** These are used in *InfoType* in **SQLGetInfo**.

- **Connection and statement attributes.** These are used in *Attribute* in **SQLGetConnectAttr**, **SQLGetStmtAttr**, **SQLSetConnectAttr**, and **SQLSetStmtAttr**.

For each of these items, there are two sets of values: values reserved for use by ODBC, and values reserved for use by drivers. Before implementing driver-specific values, a driver writer must request a value for each driver-specific type, field, or attribute from X/Open. Driver-specific data types, descriptor fields, diagnostic fields, information types, statement attributes, and connection attributes must be described in the driver documentation.

When any of these values is passed to an ODBC function, the driver must check whether the value is valid. Note that drivers return SQLSTATE HYC00 (Optional feature not implemented) for driver-specific values that apply to other drivers.

## Backward Compatibility and Standards Compliance

Backward compatibility is the ability of newer ODBC components to work with old ODBC components. The following sections discuss how these components are affected by the changes in ODBC 3.x. The information contained in them primarily addresses the writing of an ODBC 3.x application and how backward compatibility issues are handled by ODBC drivers. For specific guidelines about how backward compatibility issues affect the writing of an ODBC 3.x driver, see Appendix G, “Driver Guidelines for Backward Compatibility” contained in the Microsoft Web site (ODBC Programmer’s Guide).

### *Affected ODBC Components*

Backward compatibility describes how applications, the Driver Manager, and drivers are affected by the introduction of a new version of the Driver Manager. This affects applications and driver when either or both of them remain in the old version. There are, therefore, three types of backward compatibility to consider:

| Type                                     | Version of DM | Version of application | Version of driver |
|--|---------------|------------------------|-------------------|
| Backward Compatibility of Driver Manager | 3.x           | 2.x                    | 2.x               |
| Backward Compatibility of Driver [1]     | 3.x           | 2.x                    | 3.x               |
| Backward Compatibility of Application    | 3.x           | 3.x                    | 2.x               |

[1]The backward compatibility of drivers is primarily discussed Appendix G, “Driver Guidelines for Backward Compatibility” contained on the Microsoft Web site (ODBC Programmer’s Reference).

**Note** A standards-compliant application, for example, an application that has been written in accordance with the X/Open or ISO CLI standards, is guaranteed to work with an ODBC 3.x driver through the ODBC 3.x Driver Manager. It is assumed that the functionality that the application is using is available in the driver. It is also assumed that the standards-compliant application has been compiled with the ODBC 3.x header files.

## Types of Changes

Three types of changes are made in ODBC 3.x (and any version of ODBC). Each of these affects backward compatibility differently and is handled in a different way. These changes are as follows:

| Type of change      | Description   |
|---------------------|---|
| New features        | These are features that are new to ODBC 3.x, such as out-of-line binding or descriptors. These are only implemented when the application and driver, as well as the Driver Manager, are of version 3.x, so there is no attempt to make these backward compatible.   |
| Duplicated features | These are features that exist in both ODBC 2.x and ODBC 3.x, but are implemented in different ways in both. The functions <b>SQLAllocHandle</b> and <b>SQLAllocStmt</b> are an example. Backward compatibility issues for these and other duplicated features are mostly handled by mappings in the Driver Manager. |
| Behavioral changes  | These are features that are handled differently in ODBC 2.x and ODBC 3.x. A datetime <b>#define</b> is an example. These features are handled by the ODBC 3.x driver based on an environment attribute setting (see the " <a href="#">Behavioral Changes</a> " section later in this chapter).                      |

## Application/Driver Compatibility

ODBC applications and driver fall into a number of categories in addition to their version. Some of these applications are incompatible with some drivers; in other cases, the type of the application or driver may have a bearing on the backward compatibility issues between them.

## Types of Applications

ODBC applications can be classified as follows:

- **Pure ODBC 2.x application.** A 32-bit application that:
  - Calls only ODBC 2.x functions (including the ODBC 1.0 function **SQLSetParam**). These include ODBC 1.x applications that have been ported to 32-bit.
  - Expects ODBC 2.x behavior for features that have had behavioral changes (see "Behavioral Changes" later in this chapter).
  - Has not been recompiled with ODBC 3.5 headers.
- **Pure ODBC 2.x Recompiled application.** A pure ODBC 2.x application that has been recompiled using the ODBC 3.5 header files, by setting **ODBCVER=0x0250**.
- **Pure ODBC 2.x Unicode application.** A pure ODBC 2.x recompiled application that is Unicode-compliant and uses the **SQL\_WCHAR** data type.

- **Pure X/Open and ISO – compliant ODBC application.** A 32-bit application that:
  - Calls functions defined in the X/Open or ISO CLI standards. (These functions may include deprecated 3.0 functions.)
  - Does not use the Unicode data types.
  - Expects ODBC 3.0 behavior for features that have had.
- **Pure ODBC 3.0 application.** A 32-bit application that:
  - Is compiled with 3.0 headers.
  - Calls any ODBC 3.0 function, possibly including those that are deprecated.
  - Expects ODBC 3.0 behavior for features that have had behavioral changes.
- **Pure ODBC 3.5 (or higher) application.** A 32-bit application that:
  - May use Unicode data types.
  - Calls any ODBC 3.5 function, possibly including those that are deprecated.
  - Expects ODBC 3.5 behavior for features that have had behavioral changes.
- **Replaced application.** A 32-bit application that:
  - Implements ODBC 3.x behavior for duplicated functionality.
  - Uses any new features in ODBC 3.x only within conditional code.
  - Has limited conditional code to handle behavioral changes or has registered itself to be an ODBC 2.x application.

## Types of Drivers

ODBC drivers can be classified as follows:

- **32-bit ODBC 2.x driver.** A 32-bit driver that:
  - Exports only ODBC 2.x functions.
  - Exhibits ODBC 2.x behavior for behavioral changes.

- **ISO and X/Open – compliant driver.** A 32-bit driver that:
  - Exports all functions that are documented in the X/Open or ISO CLI documents. This will include some of functions that are deprecated in ODBC.
  - Exhibits ODBC 3.0 behavior for behavioral changes.
  - Does not necessarily go through the ODBC 3.0 Driver Manager.
- **ODBC 3.0 driver.** A 32-bit driver that:
  - Exports only functions that are in ODBC 3.0 minus deprecated functions.
  - Is capable of exhibiting ODBC 2.x behavior or ODBC 3.0 behavior with respect to behavioral changes, based on the SQL\_ATTR\_APP\_ODBC\_VERSION environment attribute.
- **ODBC 3.5 (or higher) ANSI driver.** A 32-bit driver that:
  - Exports only functions that are in ODBC 3.5 minus deprecated functions.
  - Is capable of exhibiting ODBC 2.x behavior or ODBC 3.0 behavior, or ODBC 3.5 behavior with respect to behavioral changes, based on the SQL\_ATTR\_APP\_ODBC\_VERSION environment attribute.
- **ODBC 3.5 (or higher) Unicode driver. A 32-bit driver that:**
  - Supports all the features of an ODBC 3.5 ANSI driver.
  - Exports Unicode versions of all ODBC string APIs.
  - Can store and process Unicode data on the data source.

**Note** 16-bit ODBC drivers will not work directly with the ODBC 3.x Driver Manager. However, it is possible for 16-bit drivers to work with the 2.0 ODBC Driver Manager, which subsequently thunks up to the 3.x Driver Manager.

## Compatibility Matrix

The following table describes the compatibility of the types of applications and drivers defined previously in this chapter.

| Application type and version    | 32-bit ODBC 2.x driver | ODBC 3.x driver | ISO and X/Open–compliant driver |
|---------------------------------|------------------------|-----------------|---------------------------------|
| 16-bit application, any version | Compatible             | Compatible      | Compatible                      |

|   |                |                |                    |
|---|----------------|----------------|--------------------|
| Pure 2.x application                        | Compatible     | Compatible     | Not compatible [3] |
| Pure 2.x recompiled application             | Compatible     | Compatible [1] | Not compatible [3] |
| Pure 2.x Unicode application                | Compatible     | Compatible [1] | Not Compatible [3] |
| Pure X/Open and ISO – compliant application | Not compatible | Compatible [2] | Compatible [2]     |
| Pure 3.0 application                        | Not compatible | Compatible     | Not compatible [4] |
| Pure 3.5 (or higher) application            | Not compatible | Compatible     | Not compatible [4] |
| Replaced application                        | Compatible     | Compatible     | Not compatible [3] |

[1] The application must recompile using ODBC 3.5 (or higher) headers with the UNICODE option (if it is a Unicode application) and must set ODBCVER to 0x0250.

[2] The application must compile using ODBC 3.5 (or higher) headers, and link with the ODBC Driver Manager. It must also set the header flag ODBC\_STD.

[3] This configuration can potentially fail to work because there are features in ODBC 2.x that are not in the standards, such as bookmarks.

[4] This configuration can potentially fail to work because there are features in ODBC 3.x that are not in the standards, such as bookmarks.

## New Features

The following new functionality has been introduced in ODBC 3.x. An ODBC 3.x application working with an ODBC 2.x driver will not be able to use this functionality. The ODBC 3.x Driver Manager does not map these features when working with an ODBC 2.x driver.

- Functions that take a descriptor handle as an argument: **SQLSetDescField**, **SQLGetDescField**, **SQLSetDescRec**, **SQLGetDescRec**, and **SQLCopyDesc**.
- The functions **SQLSetEnvAttr** and **SQLGetEnvAttr**.
- The use of **SQLAllocHandle** to allocate a descriptor handle. (The use of **SQLAllocHandle** to allocate environment, connection, and statement handles is duplicated, not new, functionality.)
- The use of **SQLGetConnectAttr** to get the SQL\_ATTR\_AUTO\_IPD connection attributes. (The use of **SQLSetConnectAttr** to set, and **SQLGetConnectAttr** to get, other connection attributes is duplicated, not new, functionality.)
- The use of **SQLSetStmtAttr** to set, and **SQLGetStmtAttr** to get, the following statement attributes. (The use of **SQLSetStmtAttr** to set, and **SQLGetStmtAttr** to get, other statement attributes is duplicated, not new, functionality.):

SQL\_ATTR\_APP\_ROW\_DESC  
 SQL\_ATTR\_APP\_PARAM\_DESC  
 SQL\_ATTR\_ENABLE\_AUTO\_IPD  
 SQL\_ATTR\_FETCH\_BOOKMARK\_PTR  
 SQL\_ATTR\_BIND\_OFFSET  
 SQL\_ATTR\_METADATA\_ID  
 SQL\_ATTR\_PARAM\_BIND\_OFFSET\_PTR  
 SQL\_ATTR\_PARAM\_BIND\_TYPE  
 SQL\_ATTR\_PARAM\_OPERATION\_PTR  
 SQL\_DESC\_PARAM\_STATUS\_PTR  
 SQL\_ATTR\_PARAMS\_PROCESSED\_PTR  
 SQL\_ATTR\_PARAMSET\_SIZE  
 SQL\_ATTR\_ROW\_BIND\_OFFSET\_PTR  
 SQL\_ATTR\_ROW\_OPERATION\_PTR  
 SQL\_ATTR\_ROW\_ARRAY\_SIZE

- The use of **SQLGetStmtAttr** to get the following statement attributes. (The use of **SQLGetStmtAttr** to get other statement attributes is duplicated functionality, not new functionality.):
 

SQL\_ATTR\_IMP\_ROW\_DESC  
 SQL\_ATTR\_IMP\_PARAM\_DESC
- Use of the interval C data type, the interval SQL data types, the BIGINT C data types, and the SQL\_C\_NUMERIC data structure.
- Row-wise binding of parameters.
- Offset-based bookmark fetches, such as calling **SQLFetchScroll** with an *FetchOrientation* of SQL\_FETCH\_BOOKMARK, and specifying an offset other than 0.
- **SQLFetch** returning the row status array, number of rows fetched, fetching multiple rows, intermixing calls with **SQLFetchScroll**, and intermixing calls with **SQLBulkOperations** or **SQLSetPos**. For more information, see the next section, "[Block Cursors, Scrollable Cursors, and Backward Compatibility for ODBC 3.x Applications](#)."
- Named parameters.
- Any of the ODBC 3.x-specific **SQLGetInfo** options. (If an ODBC 3.x application working with an ODBC 2.x driver calls the SQL\_XXX\_CURSOR\_ATTRIBUTES1 information types, which have replaced several ODBC 2.x information types, some of the information may be reliable, but some may be unreliable. For more information, see SQLGetInfo in the Part II PDF file, "ODBC API Reference" available on the Solid Web site.)
- Bind offsets.



- Updating, refreshing, and deleting by bookmarks (through a call to **SQLBulkOperations**).
- Calling **SQLBulkOperations** or **SQLSetPos** in the S5 state.
- The ROW\_NUMBER and COLUMN\_NUMBER fields in the diagnostic record (which have to be retrieved by the replacement functions **SQLGetDiagField** or **SQLGetDiagRec**).
- Approximate row counts.
- Warning information (SQL\_ROW\_SUCCESS\_WITH\_INFO from **SQLFetchScroll**).
- Variable-length bookmarks.
- Extended error information for arrays of parameters.
- All of the new columns in the result sets returned by the catalog functions.
- Use of **SQLDescribeCol** and **SQLColAttribute** on column 0.
- Use of any ODBC 3.x-specific column attributes in a call to **SQLColAttribute**.
- Use of multiple environment handles.

## Block Cursors, Scrollable Cursors, and Backward Compatibility for ODBC 3.x Applications

The existence of both **SQLFetchScroll** and **SQLExtendedFetch** represents the first clear split in ODBC between the Application Programming Interface (API), which is the set of functions the application calls, and the Service Provider Interface (SPI), which is the set of functions the driver implements. This split is required to balance the requirement in ODBC 3.x to align with the standards, which uses **SQLFetchScroll**, and be compatible with ODBC 2.x, which uses **SQLExtendedFetch**.

The ODBC 3.x API, which is the set of functions the application calls, includes **SQLFetchScroll** and related statement attributes. The ODBC 3.xSPI, which is the set of functions the driver implements, includes **SQLFetchScroll**, **SQLExtendedFetch**, and related statement attributes. Note that because ODBC does not formally enforce this split between the API and the SPI, it is possible for ODBC 3.x applications to call **SQLExtendedFetch** and related statement attributes. However, there is no reason for ODBC 3.x application to do this. For more information about APIs and SPIs, see the introduction to "[Chapter 3: ODBC Architecture](#)."

For information about how the ODBC 3.x Driver Manager maps calls to ODBC 2.x and ODBC 3.x drivers, and what functions and statement attributes an ODBC 3.x driver should implement for block and scrollable cursors, see “What the Driver Manager Does” in Appendix G, “Driver Guidelines for Backward Compatibility” contained on the Microsoft Web site (ODBC Programmer’s Guide).

The following table summarizes what functions and statement attributes an ODBC 3.x application should use with block and scrollable cursors. It also lists changes between ODBC 2.x and ODBC 3.x in this area that ODBC 3.x applications should be aware of to be compatible with ODBC 2.x drivers.

| Function or statement attribute | Comments  |
|---------------------------------|---|
| SQL_ATTR_FETCH_BOOKMARK_PTR     | Points to the bookmark to use with <b>SQLFetchScroll</b> . The following are implementation details: <ul style="list-style-type: none"> <li>When an application sets this in an ODBC 2.x driver, this must point to a fixed-length bookmark.</li> </ul>   |
| SQL_ATTR_ROW_STATUS_PTR         | Points to the row status array filled by <b>SQLFetch</b> , <b>SQLFetchScroll</b> , <b>SQLBulkOperations</b> , and <b>SQLSetPos</b> . The following are implementation details: <ul style="list-style-type: none"> <li>If an application sets this in an ODBC 2.x driver and calls <b>SQLBulkOperation</b> with an <i>Operation</i> of SQL_ADD before calling <b>SQLFetchScroll</b>, <b>SQLFetch</b>, or <b>SQLExtendedFetch</b>, <b>SQLSTATE HY011</b> (Attribute cannot be set now) is returned.</li> </ul> <p>Note that when an application calls <b>SQLFetch</b> in an ODBC 2.x driver, <b>SQLFetch</b> is mapped to <b>SQLExtendedFetch</b>, so returns values in this array.</p> |
| SQL_ATTR_ROWS_FETCHED_PTR       | Points to the buffer in which <b>SQLFetch</b> and <b>SQLFetchScroll</b> return the number of rows fetched. <p>Note that when an application calls <b>SQLFetch</b> in an ODBC 2.x driver, <b>SQLFetch</b> is mapped to <b>SQLExtendedFetch</b>, so returns a value in this buffer.</p>   |
| SQL_ATTR_ROW_ARRAY_SIZE         | Sets the rowset size. <p>If an application calls <b>SQLBulkOperations</b> with an <i>Operation</i> of SQL_ADD in an ODBC 2.x driver, SQL_ROWSET_SIZE will be used for the call, not SQL_ATTR_ROW_ARRAY_SIZE, because the call is mapped to <b>SQLSetPos</b> with an <i>Operation</i> of SQL_ADD, which uses SQL_ROWSET_SIZE.</p> <p>Calling <b>SQLSetPos</b> with an <i>Operation</i> of SQL_ADD or <b>SQLExtendedFetch</b> in an ODBC 2.x driver uses SQL_ROWSET_SIZE.</p> <p>Calling <b>SQLFetch</b> or <b>SQLFetchScroll</b> in an ODBC 2.x driver uses SQL_ATTR_ROW_ARRAY_SIZE.</p>   |
| <b>SQLBulkOperations</b>        | Performs insert and bookmark operations. When <b>SQLBulkOperations</b>  |

|                       |  |
|-----------------------|--|
|                       | <p>with an <i>Operation</i> of SQL_ADD is called in an ODBC 2.x driver, it is mapped to <b>SQLSetPos</b> with an <i>Operation</i> of SQL_ADD. The following are implementation details:</p> <ul style="list-style-type: none"> <li>• When working with an ODBC 2.x driver, an application must use only the implicitly allocated ARD associated with the <i>StatementHandle</i>; it cannot allocate another ARD for adding rows, because explicit descriptor operations are not supported in an ODBC 2.x driver. An application must use <b>SQLBindCol</b> to bind to the ARD, not <b>SQLSetDescField</b> or <b>SQLSetDescRec</b>.</li> <li>• When calling an ODBC 3.x driver, an application can call <b>SQLBulkOperations</b> with an <i>Operation</i> of SQL_ADD before calling <b>SQLFetch</b> or <b>SQLFetchScroll</b>. When calling an ODBC 2.x driver, an application must call <b>SQLFetchScroll</b> before calling <b>SQLBulkOperations</b> with an <i>Operation</i> of SQL_ADD.</li> </ul> |
| <b>SQLFetch</b>       | <p>Returns the next rowset. The following are implementation details:</p> <ul style="list-style-type: none"> <li>• When an application calls <b>SQLFetch</b> in an ODBC 2.x driver, it is mapped to <b>SQLExtendedFetch</b>.</li> <li>• When an application calls <b>SQLFetch</b> in an ODBC 3.x driver, it returns the number of rows specified with the SQL_ATTR_ROW_ARRAY_SIZE statement attribute.</li> </ul>  |
| <b>SQLFetchScroll</b> | <p>Returns the specified rowset. The following are implementation details:</p> <ul style="list-style-type: none"> <li>• When an application calls <b>SQLFetchScroll</b> in an ODBC 2.x driver, it returns SQLSTATE 01S01 (Error in row) before each error that applies to a single row. It does this only because the ODBC 3.x Driver Manager maps this to <b>SQLExtendedFetch</b> and <b>SQLExtendedFetch</b> returns this SQLSTATE. When an application calls <b>SQLFetchScroll</b> in an ODBC 3.x driver, it never returns SQLSTATE 01S01 (Error in row).</li> <li>• When an application calls <b>SQLFetchScroll</b> in an ODBC 2.x driver with <i>FetchOrientation</i> set to SQL_FETCH_BOOKMARK, the <i>FetchOffset</i> argument must be set to 0. SQLSTATE HYC00 (Optional feature not implemented) is returned if offset-based bookmark fetching is attempted with an ODBC 2.x driver.</li> </ul>   |

**Note** ODBC 3.x applications should not use **SQLExtendedFetch** or the SQL\_ROWSET\_SIZE statement attribute. Instead, they should use **SQLFetchScroll** and the SQL\_ATTR\_ROW\_ARRAY\_SIZE statement attribute. ODBC 3.x applications should not use

**SQLSetPos** with an *Operation* of **SQL\_ADD**, but should use **SQLBulkOperations** with an *Operation* of **SQL\_ADD**.

## Duplicated Features

The following ODBC 2.x functions have been duplicated by ODBC 3.x functions. As a result, the ODBC 2.x functions are deprecated in ODBC 3.x. The ODBC 3.x functions are referred to as replacement functions.

When an application uses a deprecated ODBC 2.x function, and the underlying driver is an ODBC 3.x driver, the Driver Manager maps the function call to the corresponding replacement function. The only exception to this rule is **SQLExtendedFetch** (see footnote in the following table). For more information on these mappings, see Appendix G, “Driver Guidelines for Backward Compatibility” contained on the Microsoft Web site (ODBC Programmer’s Reference).

When an application uses a replacement function, and the underlying driver is an ODBC 2.x driver, the Driver Manager maps the function call to the corresponding deprecated function.

| ODBC 2.x function           | ODBC 3.x function                     |
|-----------------------------|---------------------------------------|
| <b>SQLAllocConnect</b>      | <b>SQLAllocHandle</b>                 |
| <b>SQLAllocEnv</b>          | <b>SQLAllocHandle</b>                 |
| <b>SQLAllocStmt</b>         | <b>SQLAllocHandle</b>                 |
| <b>SQLColAttributes</b>     | <b>SQLColAttribute</b>                |
| <b>SQLError</b>             | <b>SQLGetDiagRec</b>                  |
| <b>SQLExtendedFetch</b> [1] | <b>SQLFetchScroll</b>                 |
| <b>SQLFreeConnect</b>       | <b>SQLFreeHandle</b>                  |
| <b>SQLFreeEnv</b>           | <b>SQLFreeHandle</b>                  |
| <b>SQLGetConnectOption</b>  | <b>SQLGetConnectAttr</b>              |
| <b>SQLGetStmtOption</b>     | <b>SQLGetStmtAttr</b>                 |
| <b>SQLParamOptions</b>      | <b>SQLSetStmtAttr, SQLGetStmtAttr</b> |
| <b>SQLSetConnectOption</b>  | <b>SQLSetConnectAttr</b>              |
| <b>SQLSetParam</b>          | <b>SQLBindParameter</b>               |
| <b>SQLSetStmtOption</b>     | <b>SQLSetStmtAttr</b>                 |
| <b>SQLTransact</b>          | <b>SQLEndTran</b>                     |

[1] The function **SQLExtendedFetch** is duplicated functionality; **SQLFetchScroll** provides the same functionality in ODBC 3.x. However, the Driver Manager does not map **SQLExtendedFetch** to **SQLFetchScroll** when going against an ODBC 3.x driver. For more details, see “What the Driver Manager Does” in the Part I PDF file, “Introducing ODBC” available on the Solid Web site and Appendix G, “Driver Guidelines for Backward Compatibility” contained on the Microsoft Web site (ODBC Programmer’s Guide). The Driver Manager maps **SQLFetchScroll** to **SQLExtendedFetch** when going against an ODBC 2.x driver.

**Note** The function **SQLBindParam** is a special case. **SQLBindParam** is duplicated functionality. This is not an ODBC 2.x function, but a function that is present in the X/Open and ISO standards. The functionality provided by this function is completely subsumed by that of **SQLBindParameter**. As a result, the Driver Manager maps a call to **SQLBindParam** to **SQLBindParameter** when the underlying driver is an ODBC 3.x driver. When the underlying driver is an ODBC 2.x driver, however, the Driver Manager does not perform this mapping.

## Behavioral Changes

Behavioral changes are those changes for which the *syntax* of the interface remains the same, but the *semantics* have changed. For these changes, functionality used in ODBC 2.x behaves differently than the same functionality in ODBC 3.x.

Whether an application exhibits ODBC 2.x behavior or ODBC 3.x behavior is determined by the `SQL_ATTR_ODBC_VERSION` environment attribute. This 32-bit value is set to `SQL_OV_ODBC2` to exhibit ODBC 2.x behavior, and `SQL_OV_ODBC3` to exhibit ODBC 3.x behavior.

The `SQL_ATTR_ODBC_VERSION` environment attribute is set by a call to **SQLSetEnvAttr**. When an application calls **SQLAllocHandle** to allocate an environment handle, then it must call **SQLSetEnvAttr** immediately to set the behavior it exhibits. (As a result, there is a new environment state to describe the environment handle in an allocated, but versionless state.) For more information, see “ODBC Transition State Tables” contained in the Part II Microsoft SDK PDF file available on the Solid Web site.

An application states what behavior it exhibits with the `SQL_ATTR_ODBC_VERSION` environment attribute, but the attribute has no effect on the application's connection with an ODBC 2.x or ODBC 3.x driver. An ODBC 3.x application will be able to connect to either an ODBC 2.x or 3.x driver, no matter what the setting of the environment attribute.

ODBC 3.x applications should never call **SQLAllocEnv**. As a result, if the Driver Manager receives a call to **SQLAllocEnv**, it recognizes the application as an ODBC 2.x application.

The `SQL_ATTR_ODBC_VERSION` attribute affects three different aspects of an ODBC 3.x driver's behavior:

- `SQLSTATEs`
- Data types for date, time, and timestamp
- The *CatalogName* argument in **SQLTables** accepts search patterns in ODBC 3.x, but not in ODBC 2.x

The setting of the `SQL_ATTR_ODBC_VERSION` environment attribute does not affect **SQLSetParam** or **SQLBindParam**. **SQLColAttribute** is also not affected by this bit. Although **SQLColAttribute** returns attributes that are affected by the version of ODBC (date type, precision, scale and length), the intended behavior is determined by the value of the *FieldIdentifier* argument. When *FieldIdentifier* is equal to `SQL_DESC_TYPE`, **SQLColAttribute** returns the ODBC 3.x codes for date, time, and timestamp; when *FieldIdentifier* is equal to `SQL_COLUMN_TYPE`, **SQLColAttribute** returns the ODBC 2.x codes for date, time, and timestamp.

### ***SQLSTATE Mappings***

In ODBC 3.x, HYxxx SQLSTATES are returned instead of S1xxx, and 42Sxx SQLSTATES are returned instead of S00XX. This was done to align with X/Open and ISO standards. In many cases, the mapping is not one-to-one because the standards have redefined the interpretation of several SQLSTATES.

When an ODBC 2.x application is upgraded to an ODBC 3.x application, the application has to be changed to expect ODBC 3.x SQLSTATES instead of ODBC 2.x SQLSTATES. The following table lists the ODBC 3.x SQLSTATES that each ODBC 2.x SQLSTATE is mapped to.

When the SQL\_ATTR\_ODBC\_VERSION environment attribute is set to SQL\_OV\_ODBC2, the driver posts ODBC 2.x SQLSTATES instead of ODBC 3.x SQLSTATES when **SQLGetDiagField** or **SQLGetDiagRec** is called. A specific mapping can be determined by noting the ODBC 2.x SQLSTATE in column 1 of the following table that corresponds to the ODBC 3.x SQLSTATE in column 2.

| ODBC 2.x SQLSTATE | ODBC 3.x SQLSTATE | Comments   |
|-------------------|-------------------|--|
| 01S03             | 01001             |  |
| 01S04             | 01001             |  |
| 22003             | HY019             |  |
| 22008             | 22007             |  |
| 22005             | 22018             |  |
| 24000             | 07005             |  |
| 37000             | 42000             |  |
| 70100             | HY018             |  |
| S0001             | 42S01             |  |
| S0002             | 42S02             |  |
| S0011             | 42S11             |  |
| S0012             | 42S12             |  |
| S0021             | 42S21             |  |
| S0022             | 42S22             |  |
| S0023             | 42S23             |  |
| S1000             | HY000             |  |
| S1001             | HY001             |  |
| S1002             | 07009             | ODBC 2.x SQLSTATE S1002 is mapped to ODBC 3.x SQLSTATE 07009 if the underlying function is <b>SQLBindCol</b> , <b>SQLColAttribute</b> , <b>SQLExtendedFetch</b> , <b>SQLFetch</b> , <b>SQLFetchScroll</b> , or <b>SQLGetData</b> . |
| S1003             | HY003             |  |
| S1004             | HY004             |  |
| S1008             | HY008             |  |

|       |                |   |
|-------|----------------|---|
| S1009 | HY009          | Returned for an invalid use of a null pointer.  |
| S1009 | HY024          | Returned for an invalid attribute value.  |
| S1009 | HY092          | Returned for updating or deleting data by a call to <b>SQLSetPos</b> , or adding, updating, or deleting data by a call to <b>SQLBulkOperations</b> , when the concurrency is read-only.   |
| S1010 | HY007<br>HY010 | SQLSTATE S1010 is mapped to SQLSTATE HY007 when <b>SQLDescribeCol</b> is called prior to calling <b>SQLPrepare</b> , <b>SQLExecDirect</b> , or a catalog function for the <i>StatementHandle</i> . Otherwise, SQLSTATE S1010 is mapped to SQLSTATE HY010. |
| S1011 | HY011          |   |
| S1012 | HY012          |   |
| S1090 | HY090          |   |
| S1091 | HY091          |   |
| S1092 | HY092          |   |
| S1093 | 07009          | ODBC 3.x SQLSTATE 07009 is mapped to ODBC 2.x SQLSTATE S1093 if the underlying function is <b>SQLBindParameter</b> or <b>SQLDescribeParam</b> .   |
| S1096 | HY096          |   |
| S1097 | HY097          |   |
| S1098 | HY098          |   |
| S1099 | HY099          |   |
| S1100 | HY100          |   |
| S1101 | HY101          |   |
| S1103 | HY103          |   |
| S1104 | HY104          |   |
| S1105 | HY105          |   |
| S1106 | HY106          |   |
| S1107 | HY107          |   |
| S1108 | HY108          |   |
| S1109 | HY109          |   |
| S1110 | HY110          |   |

|       |       |  |
|-------|-------|--|
| S1111 | HY111 |  |
| S1C00 | HYC00 |  |
| S1T00 | HYT00 |  |

Note ODBC 3.x SQLSTATE 07008 is mapped to ODBC 2.x SQLSTATE S1000.

### ***Datetime Data Type Changes***

In ODBC 3.x, the identifiers for date, time, and timestamp SQL data types have changed from SQL\_DATE, SQL\_TIME, and SQL\_TIMESTAMP (with instances of **#define** in the header file of 9, 10, and 11) to SQL\_TYPE\_DATE, SQL\_TYPE\_TIME, and SQL\_TYPE\_TIMESTAMP (with instances of **#define** in the header file of 91, 92, and 93), respectively. The corresponding C type identifiers have changed from SQL\_C\_DATE, SQL\_C\_TIME, and SQL\_C\_TIMESTAMP to SQL\_C\_TYPE\_DATE, SQL\_C\_TYPE\_TIME, and SQL\_C\_TYPE\_TIMESTAMP, respectively.

The column size and decimal digits returned for the SQL datetime data types in ODBC 3.x are the same as the precision and scale returned for them in ODBC 2.x. These values are different than the values in the SQL\_DESC\_PRECISION and SQL\_DESC\_SCALE descriptor fields. (For more information, see Appendix D, “Data Types” in the **SOLID Programmer Guide**.)

These changes affect SQLDescribeCol, SQLDescribeParam, and SQLColAttribute; SQLBindCol, SQLBindParameter, and SQLGetData; and SQLColumns, SQLGetTypeInfo, SQLProcedureColumns, SQLStatistics, and SQLSpecialColumns.

The following table shows how the ODBC 3.x Driver Manager performs mapping of the date, time, and timestamp C data types entered in the *TargetType* arguments of **SQLBindCol** and **SQLGetData** or the *ValueType* argument of **SQLBindParameter**.

| <b>Data type code entered</b> | <b>2.x app to 2.x driver</b> | <b>2.x app to 3.x driver</b> | <b>3.x app to 2.x driver</b> | <b>3.x app to 3.x driver</b> |
|-------------------------------|------------------------------|------------------------------|------------------------------|------------------------------|
| SQL_C_DATE (9)                | No mapping                   | SQL_C_TYPE_DATE (91)         | No mapping [1]               | SQL_C_TYPE_DATE (91)         |
| SQL_C_TYPE_DATE (91)          | Error (from DM)              | Error (from DM)              | SQL_C_DATE (9)               | No mapping [2]               |
| SQL_C_TIME (10)               | No mapping                   | SQL_C_TYPE_TIME (92)         | No mapping [1]               | SQL_C_TYPE_TIME (92)         |
| SQL_C_TYPE_TIME (92)          | Error (from DM)              | Error (from DM)              | SQL_C_TIME (10)              | No mapping [2]               |
| SQL_C_TIMESTAMP (11)          | No mapping                   | SQL_C_TYPE_TIMESTAMP (93)    | No mapping [1]               | SQL_C_TYPE_TIMESTAMP (93)    |
| SQL_C_TYPE_TIMESTAMP (93)     | Error (from DM)              | Error (from DM)              | SQL_C_TIMESTAMP (11)         | No mapping [2]               |



[1] As a result of this, an ODBC 3.x application working with an ODBC 2.x driver can use the date, time, or timestamp codes returned in the result sets that are returned by the catalog functions.

[2] As a result of this, an ODBC 3.x application working with an ODBC 3.x driver can use the date, time, or timestamp codes returned in the result sets that are returned by the catalog functions.

The following table shows how the ODBC 3.x Driver Manager performs mapping of the date, time, and timestamp SQL data types entered in the *ParameterType* argument of **SQLBindParameter** or the *DataType* argument of **SQLGetTypeInfo**.

| Data type code entered  | 2.x app to 2.x driver | 2.x app to 3.x driver   | 3.x app to 2.x driver | 3.x app to 3.x driver   |
|-------------------------|-----------------------|-------------------------|-----------------------|-------------------------|
| SQL_DATE (9)            | No mapping            | SQL_TYPE_DATE (91)      | No mapping [1]        | SQL_TYPE_DATE (91)      |
| SQL_TYPE_DATE (91)      | Error (from DM)       | Error (from DM)         | SQL_DATE (9)          | No mapping [2]          |
| SQL_TIME (10)           | No mapping            | SQL_TYPE_TIME (92)      | No mapping [1]        | SQL_TYPE_TIME (92)      |
| SQL_TYPE_TIME (92)      | Error (from DM)       | Error (from DM)         | SQL_TIME (10)         | No mapping [2]          |
| SQL_TIMESTAMP (11)      | No mapping            | SQL_TYPE_TIMESTAMP (93) | No mapping [1]        | SQL_TYPE_TIMESTAMP (93) |
| SQL_TYPE_TIMESTAMP (93) | Error (from DM)       | Error (from DM)         | SQL_TIMESTAMP (11)    | No mapping [2]          |

<sup>1</sup>As a result of this, an ODBC 3.x application working with an ODBC 2.x driver can use the date, time, or timestamp codes returned in the result sets that are returned by the catalog functions.

<sup>2</sup>As a result of this, an ODBC 3.x application working with an ODBC 3.x driver can use the date, time, or timestamp codes returned in the result sets that are returned by the catalog functions.

## Writing ODBC 3.x Applications

When an ODBC 2.x application is upgraded to ODBC 3.x, it should be written such that it works with both ODBC 2.x and 3.x drivers. The application should incorporate conditional code to take full advantage of the ODBC 3.x features.

The `SQL_ATTR_ODBC_VERSION` environment attribute should be set to `SQL_OV_ODBC2`. This will ensure that the driver behaves like an ODBC 2.x driver with respect to the changes described in the "[Behavioral Changes](#)" section earlier in this chapter are concerned.

If the application will use any of the features described in the "New Features" section earlier in this chapter, conditional code should be used to determine whether the driver is an ODBC 3.x or ODBC 2.x driver. The application uses **SQLGetDiagField** and **SQLGetDiagRec** to obtain ODBC 3.x SQLSTATEs while doing

error processing on these conditional code fragments. The following points about the new functionality should be considered:

- An application affected by the change in rowset size behavior should be careful not to call **SQLFetch** when the array size is greater than 1. These applications should replace calls to **SQLExtendedFetch** with calls to **SQLSetStmtAttr** to set the **SQL\_ATTR\_ARRAY\_STATUS\_PTR** statement attribute and **SQLFetchScroll**, so they have common code that works with both ODBC 3.x and ODBC 2.x drivers. Because **SQLSetStmtAttr** with **SQL\_ATTR\_ROW\_ARRAY\_SIZE** will be mapped to **SQLSetStmtAttr** with **SQL\_ROWSET\_SIZE** for ODBC 2.x drivers, applications can just set **SQL\_ATTR\_ROW\_ARRAY\_SIZE** for their multirow fetch operations.
- Most applications that are upgrading are not actually affected by changes in **SQLSTATE** codes. For those applications that are affected, they can do a mechanical search and replace in most cases using the error conversion table in the "SQLSTATE Mapping" section to convert ODBC 3.x error codes to ODBC 2.x codes. Since the ODBC 3.x Driver Manager will perform mapping from ODBC 2.x **SQLSTATE**s to ODBC 3.x **SQLSTATE**s, these application writers need only check for the ODBC 3.x **SQLSTATE**s and not worry about including ODBC 2.x **SQLSTATE**s in conditional code.
- If an application makes great use of date, time, and timestamp data types, the application can declare itself to be an ODBC 2.x application and use its existing code, instead of using conditioning code.

The upgrade should also include the following steps:

- Call **SQLSetEnvAttr** before allocating a connection to set the **SQL\_ATTR\_ODBC\_VERSION** environment attribute to **SQL\_OV\_ODBC2**.
- Replace all calls to **SQLAllocEnv**, **SQLAllocConnect**, or **SQLAllocStmt** with calls to **SQLAllocHandle** with the appropriate *HandleType* argument of **SQL\_HANDLE\_ENV**, **SQL\_HANDLE\_DBC**, or **SQL\_HANDLE\_STMT**.
- Replace all calls to **SQLFreeEnv** or **SQLFreeConnect** with calls to **SQLFreeHandle** with the appropriate *HandleType* argument of **SQL\_HANDLE\_DBC** or **SQL\_HANDLE\_STMT**.
- Replace all calls to **SQLSetConnectOption** with calls to **SQLSetConnectAttr**. If setting an attribute whose value is a string, set the *StringLength* argument appropriately. Change *Attribute* argument from **SQL\_XXXX** to **SQL\_ATTR\_XXXX**.
- Replace all calls to **SQLGetConnectOption** with calls to **SQLGetConnectAttr**. If getting a string or binary attribute, set *BufferLength* to the appropriate value and pass in a *StringLength* argument. Change *Attribute* argument from **SQL\_XXXX** to **SQL\_ATTR\_XXXX**.

- Replace all calls to **SQLSetStmtOption** with calls to **SQLSetStmtAttr**. If setting an attribute whose value is a string, set the *StringLength* argument appropriately. Change *Attribute* argument from SQL\_XXXX to SQL\_ATTR\_XXXX.
- Replace all calls to **SQLGetStmtOption** with calls to **SQLGetStmtAttr**. If getting a string or binary attribute, set *BufferLength* to the appropriate value and pass in a *StringLength* argument. Change *Attribute* argument from SQL\_XXXX to SQL\_ATTR\_XXXX.
- Replace all calls to **SQLTransact** with calls to **SQLEndTran**. If the rightmost valid handle in the **SQLTransact** call is an environment handle, then a *HandleType* argument of SQL\_HANDLE\_ENV should be used in the **SQLEndTran** call with the appropriate *Handle* argument. If the rightmost valid handle in your **SQLTransact** call is a connection handle, then a *HandleType* argument of SQL\_HANDLE\_DBC should be used in the **SQLEndTran** call with the appropriate *Handle* argument.
- Replace all calls to **SQLColAttributes** with calls to **SQLColAttribute**. If the *FieldIdentifier* argument is either SQL\_COLUMN\_PRECISION, SQL\_COLUMN\_SCALE, or SQL\_COLUMN\_LENGTH, then do not change anything other than the name of the function. If not, change *FieldIdentifier* from SQL\_COLUMN\_XXXX to SQL\_DESC\_XXXX. If *FieldIdentifier* is SQL\_DESC\_CONCISE\_TYPE and the data type is a datetime data type, change to the corresponding ODBC 3.x data type.
- If using block cursors, scrollable cursors, or both then the application:
  - Sets the rowset size, cursor type, and cursor concurrency using **SQLSetStmtAttr**.
  - Calls **SQLSetStmtAttr** to set SQL\_ATTR\_ROW\_STATUS\_PTR to point to an array of status records.
  - Calls **SQLSetStmtAttr** to set SQL\_ATTR\_ROWS\_FETCHED\_PTR to point to an SQLINTEGER.
  - Performs the required bindings and executes the SQL statement.
  - Calls **SQLFetchScroll** in a loop to fetch rows and move around in the result set.
  - If it wants to fetch by bookmark, then the application calls **SQLSetStmtAttr** to set SQL\_ATTR\_FETCH\_BOOKMARK\_PTR to a variable that will contain the bookmark for the row that it wants to fetch, and calls **SQLFetchScroll** with a *FetchOrientation* argument of SQL\_FETCH\_BOOKMARK.
- If using arrays of parameters, then the application:

- Calls **SQLSetStmtAttr** to set the SQL\_ATTR\_PARAMSET\_SIZE attribute to the size of the parameter array.
- Calls **SQLSetStmtAttr** to set SQL\_ATTR\_ROWS\_PROCESSED\_PTR to point to an internal UDWORD variable.
- Performs prepare, bind, and execute operations as appropriate.
- If execution halts for some reason (such as SQL\_NEED\_DATA), it can find the "current" row of parameters by inspecting the location pointed to by SQL\_ATTR\_ROWS\_PROCESSED\_PTR.

## Mapping Replacement Functions for Backward Compatibility of Applications

An ODBC 3.x application working through the ODBC 3.x Driver Manager will work against an ODBC 2.x driver as long as no new features are used. Both duplicated functionality and behavioral changes do, however, affect the way that the ODBC 3.x application works on an ODBC 2.x driver. When working with an ODBC 2.x driver, the Driver Manager maps the following ODBC 3.x functions, which have replaced one or more ODBC 2.x functions, into the corresponding ODBC 2.x functions.

| ODBC 3.x function | ODBC 2.x function                                   |
|-------------------|---|
| SQLAllocHandle    | SQLAllocEnv,<br>SQLAllocConnect, or<br>SQLAllocStmt |
| SQLBulkOperations | SQLSetPos   |
| SQLColAttribute   | SQLColAttributes                                    |
| SQLEndTran        | SQLTransact   |
| SQLFetch          | SQLExtendedFetch                                    |
| SQLFetchScroll    | SQLExtendedFetch                                    |
| SQLFreeHandle     | SQLFreeEnv,<br>SQLFreeConnect, or<br>SQLFreeStmt    |
| SQLGetConnectAttr | SQLGetConnectOption                                 |
| SQLGetDiagRec     | SQLError  |
| SQLGetStmtAttr    | SQLGetStmtOption [1]                                |
| SQLSetConnectAttr | SQLSetConnectOption                                 |
| SQLSetStmtAttr    | SQLSetStmtOption [1]                                |

[1] Other actions might also be taken, depending on the attribute being requested.

## SQLAllocHandle

The Driver Manager maps this to **SQLAllocEnv**, **SQLAllocConnect**, or **SQLAllocStmt**, as appropriate. The following call to **SQLAllocHandle**:

```
SQLAllocHandle(HandleType, InputHandle, OutputHandlePtr);
```

will result in the Driver Manager performing the following (conceptual, no error checking) mapping:

```
switch (HandleType) {  
    case SQL_HANDLE_ENV: return (SQLAllocEnv(OutputHandlePtr));  
    case SQL_HANDLE_DBC: return (SQLAllocConnect (InputHandle, OutputHandlePtr));  
    case SQL_HANDLE_STMT: return (SQLAllocStmt (InputHandle, OutputHandlePtr));  
    default: // return SQL_ERROR, SQLSTATE HY092 ("Invalid attribute/option  
identifier")  
}
```

## SQLBulkOperations

The Driver Manager maps this to **SQLSetPos**. The following call to **SQLBulkOperations**:

```
SQLBulkOperations(hstmt, Operation);
```

will result in the following sequence of steps:

1. If the Operation argument is **SQL\_ADD**, the Driver Manager calls **SQLSetPos** as follows:
2. **SQLSetPos** (hstmt, 0, **SQL\_ADD**, **SQL\_LOCK\_NO\_CHANGE**);
3. If the Operation argument is not **SQL\_ADD**, the driver returns SQLSTATE HY092 (Invalid attribute/option identifier).
4. If the application attempts to change the **SQL\_ATTR\_ROW\_STATUS\_PTR** between calls to **SQLFetch** or **SQLFetchScroll** and **SQLBulkOperations**, the Driver Manager will return SQLSTATE HY011 (Attribute cannot be set now).
5. If the Operation argument is **SQL\_ADD**, the application must call **SQLBindCol** to bind the data to be inserted. It cannot call **SQLSetDescField** or **SQLSetDescRec** to bind the data to be inserted.
6. If the Operation argument is **SQL\_ADD**, and the number of rows to be inserted is not the same as the current rowset size, then **SQLSetStmtAttr** must be called to set the **SQL\_ATTR\_ROW\_ARRAY\_SIZE** statement attribute to the number of rows to be inserted before calling **SQLBulkOperations**. To revert back to the previous rowset size, the application must set the **SQL\_ATTR\_ROW\_ARRAY\_SIZE** statement attribute before **SQLFetch**, **SQLFetchScroll**, or **SQLSetPos** is called.

## SQLColAttribute

The Driver Manager maps this to **SQLColAttributes**. The following call to **SQLColAttribute**:

```
SQLColAttribute(StatementHandle, ColumnNumber, FieldIdentifier, CharacterAttributePtr, BufferLength,  
StringLengthPtr, NumericAttributePtr);
```

will result in the following sequence of steps:

1. If *FieldIdentifier* is one of the following:

SQL\_DESC\_PRECISION, SQL\_DESC\_SCALE, SQL\_DESC\_LENGTH,  
SQL\_DESC\_OCTET\_LENGTH, SQL\_DESC\_UNNAMED,  
SQL\_DESC\_BASE\_COLUMN\_NAME, SQL\_DESC\_LITERAL\_PREFIX,  
SQL\_DESC\_LITERAL\_SUFFIX, or SQL\_DESC\_LOCAL\_TYPE\_NAME

then the Driver Manager returns SQL\_ERROR with SQLSTATE HY091 (Invalid descriptor field identifier). No further rules of this section apply.

2. The Driver Manager maps SQL\_COLUMN\_COUNT, SQL\_COLUMN\_NAME, or SQL\_COLUMN\_NULLABLE to SQL\_DESC\_COUNT, SQL\_DESC\_NAME, or SQL\_DESC\_NULLABLE, respectively. (An ODBC 2.x driver need only support SQL\_COLUMN\_COUNT, SQL\_COLUMN\_NAME, and SQL\_COLUMN\_NULLABLE, not SQL\_DESC\_COUNT, SQL\_DESC\_NAME, and SQL\_DESC\_NULLABLE.) The call to SQLColAttribute is mapped to:
3. SQLColAttributes(StatementHandle, ColumnNumber, FieldIdentifier, CharacterAttributePtr, BufferLength, StringLengthPtr, NumericAttributePtr);
4. All other *FieldIdentifier* values are passed through to the driver, with **SQLColAttribute** mapped to **SQLColAttributes** as shown previously.
5. If *BufferLength* is less than 0, then the Driver Manager returns SQL\_ERROR with SQLSTATE HY090 (Invalid string or buffer length). No further rules of this section apply.
6. If *FieldIdentifier* is SQL\_DESC\_CONCISE\_TYPE, and the returned type is a concise datetime data type, the Driver Manager maps the return values for date, time, and timestamp codes.
7. The Driver Manager performs necessary checks to see if SQLSTATE HY010 (Function sequence error) needs to be raised. If so, the Driver Manager returns SQL\_ERROR and SQLSTATE HY010 (Function sequence error). No further rules of this section apply.

## SQLEndTran

The Driver Manager maps this to **SQLTransact**. The following call to **SQLEndTran**:

```
SQLEndTran(HandleType, Handle, CompletionType);
```

will result in the Driver Manager performing the following (conceptual, no error checking) mapping:

```
switch (HandleType) {  
    case SQL_HANDLE_ENV: return(SQLTransact(Handle, SQL_NULL_HDBC,  
CompletionType));  
    case SQL_HANDLE_DBC: return(SQLTransact(SQL_NULL_HENV, Handle,  
CompletionType));  
    default: // return SQL_ERROR, SQLSTATE HY092 ("Invalid attribute/option  
identifier")  
}
```

## SQLFetch

The Driver Manager maps this to **SQLExtendedFetch** with a *FetchOrientation* of `SQL_FETCH_NEXT`. The following call to **SQLFetch**:

```
SQLFetch (StatementHandle);
```

will result in the Driver Manager calling **SQLExtendedFetch**, as follows:

```
rc = SQLExtendedFetch(StatementHandle, FetchOrientation, FetchOffset, &RowCount, RowStatusArray);
```

In this call, the *pcRow* argument is set to the value that the application sets the `SQL_ATTR_ROWS_FETCHED_PTR` statement attribute to through a call to **SQLSetStmtAttr**.

Recall that when the application calls **SQLSetStmtAttr** to set `SQL_ATTR_ROW_STATUS_PTR` to point to a status array, the Driver Manager caches the pointer. *RowStatusArray* can be equal to a null pointer.

If the driver does not support **SQLExtendedFetch** and the cursor library is loaded, then the Driver Manager uses the cursor library's **SQLExtendedFetch** to map **SQLFetch** to **SQLExtendedFetch**. If the driver does not support **SQLExtendedFetch** and the cursor library is not loaded, the Driver Manager passes the call to **SQLFetch** through to the driver. If the application calls **SQLSetStmtAttr** to set `SQL_ATTR_ROW_STATUS_PTR`, then the Driver Manager ensures that the array is populated. If the application calls **SQLSetStmtAttr** to set `SQL_ATTR_ROWS_FETCHED_PTR`, then the Driver Manager sets this field to 1.

## SQLFetchScroll

The Driver Manager maps this to **SQLExtendedFetch**. The following call to **SQLFetchScroll**:

```
SQLFetchScroll(StatementHandle, FetchOrientation, FetchOffset);
```

will result in the following sequence of steps:

1. Recall that when the application calls **SQLSetStmtAttr** to set `SQL_ATTR_ROW_STATUS_PTR` (which sets the `SQL_DESC_ARRAY_STATUS_PTR` field in the IRD) to point to a status array, the Driver Manager caches this pointer. Let this pointer be *RowStatusArray*; otherwise, let *RowStatusArray* be equal to a null pointer. If the *RowStatusArray* arguments is set to a null pointer, the Driver Manager generates a row-status array.
2. If *FetchOrientation* is not one of `SQL_FETCH_NEXT`, `SQL_FETCH_PRIOR`, `SQL_FETCH_ABSOLUTE`, `SQL_FETCH_RELATIVE`, `SQL_FETCH_FIRST`, `SQL_FETCH_LAST`, or `SQL_FETCH_BOOKMARK`, then the Driver Manager returns with `SQL_ERROR` and `SQLSTATE HY106` (Fetch type out of range). No further rules of this section apply.
3. Case:
  - If *FetchOrientation* is equal to `SQL_FETCH_BOOKMARK`, then:

- If **SQLSetStmtAttr** was called earlier to set the value of `SQL_ATTR_FETCH_BOOKMARK_PTR`, then let *Bmk* be the value obtained by dereferencing the pointer `SQL_DESC_FETCH_BOOKMARK_PTR`.
- Otherwise, return `SQL_ERROR` with `SQLSTATE HY111` (Invalid bookmark value). No further rules of this section apply.

The Driver Manager now calls **SQLExtendedFetch**, as follows:

```
rc = SQLExtendedFetch(StatementHandle, FetchOrientation, Bmk, pcRow, RowStatusArray);
```

- Otherwise, the Driver Manager calls **SQLExtendedFetch**, as follows:
- `rc = SQLExtendedFetch(StatementHandle, FetchOrientation, FetchOffset, pcRow, RowStatusArray);`

In these calls, the *pcRow* argument is set to the value that the application sets the `SQL_ATTR_ROWS_FETCHED_PTR` statement attribute to through a call to **SQLSetStmtAttr**.

4. `SQL_ATTR_ROW_ARRAY_SIZE` is mapped to `SQL_ROWSET_SIZE`.
5. If *rc* is equal to `SQL_SUCCESS` or `SQL_SUCCESS_WITH_INFO`, and if *FetchOrientation* is equal to `SQL_FETCH_BOOKMARK` and *FetchOffset* is not equal to 0, then the Driver Manager posts a warning, `SQLSTATE 01S10` (Attempt to fetch by a bookmark offset, offset value ignored), and returns `SQL_SUCCESS_WITH_INFO`.

## SQLFreeHandle

The Driver Manager maps this to **SQLFreeEnv**, **SQLFreeConnect**, or **SQLFreeStmt** as appropriate. The following call to **SQLFreeHandle**:

```
SQLFreeHandle(HandleType, Handle);
```

will result in the Driver Manager performing the following (conceptual, no error checking) mapping:

```
switch (HandleType) {
    case SQL_HANDLE_ENV: return (SQLFreeEnv(Handle));
    case SQL_HANDLE_DBC: return (SQLFreeConnect(Handle));
    case SQL_HANDLE_STMT: return (SQLFreeStmt(Handle, SQL_DROP));
    default: // return SQL_ERROR, SQLSTATE HY092 ("Invalid attribute/option
identifier")
}
```

## SQLGetConnectAttr

The Driver Manager maps this to **SQLGetConnectOption**. The following call to **SQLGetConnectAttr**:

```
SQLGetConnectAttr(ConnectionHandle, Attribute, ValuePtr, BufferLength,
StringLengthPtr);
```

will result in the following sequence of steps:

1. If *Attribute* is not a driver-defined connection or statement attribute, and is not an attribute defined in ODBC 2.x, then the Driver Manager returns `SQL_ERROR` with `SQLSTATE HY092` (Invalid attribute/option identifier). No further rules in this section apply.



2. If *Attribute* is equal to `SQL_ATTR_AUTO_IPD` or `SQL_ATTR_METADATA_ID`, then the Driver Manager returns `SQL_ERROR` with `SQLSTATE HY092` (Invalid attribute/option identifier).
3. The Driver Manager performs necessary checks to see if `SQLSTATE 08003` (Connection not open) or `SQLSTATE HY010` (Function sequence error) needs to be raised. If so, the Driver Manager returns `SQL_ERROR` and posts the appropriate error message. No further rules of this section apply.
4. The Driver Manager calls **SQLGetConnectOption** as follows:
5. `SQLGetConnectOption` (ConnectionHandle, Attribute, ValuePtr);  
Note that the *BufferLength* and *StringLengthPtr* are ignored.

## SQLGetData

When an ODBC 3.x application working with an ODBC 2.x driver calls **SQLGetData** with the *ColumnNumber* argument equal to 0, the ODBC 3.x Driver Manager maps this to a call to **SQLGetStmtOption** with the *Option* attribute set to `SQL_GET_BOOKMARK`.

## SQLGetStmtAttr

The Driver Manager maps this to **SQLGetStmtOption**. The following call to **SQLGetStmtAttr**:

```
SQLGetStmtAttr(StatementHandle, Attribute, ValuePtr, BufferLength,
StringLengthPtr);
```

will result in the following sequence of steps:

1. If *Attribute* is not a driver-defined connection or statement attribute, and is not an attribute defined in ODBC 2.x, then the Driver Manager returns `SQL_ERROR` with `SQLSTATE HY092` (Invalid attribute/option identifier). No further rules in this section apply.
2. If *Attribute* is one of the following:

`SQL_ATTR_APP_ROW_DESC`, `SQL_ATTR_APP_PARAM_DESC`, `SQL_ATTR_AUTO_IPD`,  
`SQL_ATTR_ROW_BIND_TYPE`, `SQL_ATTR_IMP_ROW_DESC`,  
`SQL_ATTR_IMP_PARAM_DESC`, `SQL_ATTR_METADATA_ID`,  
`SQL_ATTR_PARAM_BIND_TYPE`, `SQL_ATTR_PREDICATE_PTR`,  
`SQL_ATTR_PREDICATE_OCTET_LENGTH_PTR`,  
`SQL_ATTR_PARAM_BIND_OFFSET_PTR`, `SQL_ATTR_ROW_BIND_OFFSET_PTR`,  
`SQL_ATTR_ROW_OPERATION_PTR`, `SQL_ATTR_PARAM_OPERATION_PTR`

Then the Driver Manager returns `SQL_ERROR` with `SQLSTATE HY092` (Invalid attribute/option identifier). No further rules of this section apply.

3. The Driver Manager performs necessary checks to see if `SQLSTATE HY010` (Function sequence error) needs to be raised. If so, the Driver Manager returns `SQL_ERROR` and `SQLSTATE HY010` (Function sequence error). No further rules of this section apply.

4. If *Attribute* is equal to `SQL_ATTR_ROWS_FETCHED_PTR`, the Driver Manager returns a pointer to the internal Driver Manager variable *cRow*, which it has used or will use in a call to **SQLExtendedFetch**. No further rules of this section apply.
5. If *Attribute* is equal to `SQL_DESC_FETCH_BOOKMARK_PTR`, the Driver Manager returns the appropriate pointer that it had cached during a call to **SQLSetStmtAttr**.
6. If *Attribute* is equal to `SQL_ATTR_ROW_STATUS_PTR`, the Driver Manager returns the appropriate pointer that it had cached during a call to **SQLSetStmtAttr**.
7. The Driver Manager calls **SQLGetStmtOption** as follows:
8. `SQLGetStmtOption (hstmt, fOption, pvParam);`  
 where *hstmt*, *fOption*, and *pvParam* will be set to the values of *StatementHandle*, *Attribute*, and *ValuePtr*, respectively. Note that the *BufferLength* and *StringLengthPtr* are ignored.

## SQLSetConnectAttr

The Driver Manager maps this to **SQLSetConnectOption**. The following call to **SQLSetConnectAttr**:

```
SQLSetConnectAttr(ConnectionHandle, Attribute, ValuePtr, StringLength);
```

will result in the following sequence of steps:

1. If *Attribute* is not a driver-defined connection or statement attribute, and is not an attribute defined in ODBC 2.x, then the Driver Manager returns `SQL_ERROR` with `SQLSTATE HY092` (Invalid attribute/option identifier). No further rules in this section apply.
2. If *Attribute* is equal to `SQL_ATTR_AUTO_IPD`, then the Driver Manager returns `SQL_ERROR` with `SQLSTATE HY092` (Invalid attribute/option identifier).
3. The Driver Manager performs necessary checks to see if `SQLSTATE 08003` (Connection not open) or `SQLSTATE HY010` (Function sequence error) need to be raised. If one of these errors needs to be raised, the Driver Manager returns `SQL_ERROR` and posts the appropriate error message. No further rules of this section apply.
4. The Driver Manager calls **SQLSetConnectOption** as follows:
5. `SQLSetConnectOption (hdbc, fOption, vParam);`  
 where *hdbc*, *fOption*, and *vParam* will be set to the values of *ConnectionHandle*, *Attribute*, and *ValuePtr*, respectively. Note that *StringLengthPtr* is ignored.

**Note** The ability to set statement attributes on the connection level has been deprecated. Statement attributes should never be set on the connection level by an ODBC 3.x application.

## SQLSetStmtAttr

The Driver Manager maps this to **SQLSetStmtOption**. The following call to **SQLSetStmtAttr**:

```
SQLSetStmtAttr(StatementHandle, Attribute, ValuePtr, StringLength);
```

will result in the following sequence of steps:

1. If *Attribute* is not a driver-defined connection or statement attribute, and is not an attribute defined in ODBC 2.x, then the Driver Manager returns `SQL_ERROR` with `SQLSTATE HY092` (Invalid attribute/option identifier). No further rules in this section apply.

2. If *Attribute* is one of the following:

`SQL_ATTR_APP_ROW_DESC`, `SQL_ATTR_APP_PARAM_DESC`, `SQL_ATTR_AUTO_IPD`,  
`SQL_ATTR_ROW_BIND_TYPE`, `SQL_ATTR_IMP_ROW_DESC`,  
`SQL_ATTR_IMP_PARAM_DESC`, `SQL_ATTR_METADATA_ID`,  
`SQL_ATTR_PARAM_BIND_TYPE`, `SQL_ATTR_PREDICATE_PTR`,  
`SQL_ATTR_PREDICATE_OCTET_LENGTH_PTR`,  
`SQL_ATTR_PARAM_BIND_OFFSET_PTR`, `SQL_ATTR_ROW_BIND_OFFSET_PTR`,  
`SQL_ATTR_ROW_OPERATION_PTR`, `SQL_ATTR_PARAM_OPERATION_PTR`.

Then the Driver Manager returns `SQL_ERROR` with `SQLSTATE HY092` (Invalid attribute/option identifier). No further rules of this section apply.

3. The Driver Manager performs the necessary checks to see if `SQLSTATE HY010` (Function sequence error) need to be raised. If so, the Driver Manager returns `SQL_ERROR` and `SQLSTATE HY010` (Function sequence error). No further rules of this section apply.
4. If *Attribute* is equal to `SQL_ATTR_PARAMSET_SIZE` or `SQL_ATTR_PARAMS_PROCESSED_PTR`, then see the section "Mappings for Handling Parameter Arrays" later in this topic. No further rules of this section apply.
5. If *Attribute* is equal to `SQL_ATTR_ROWS_FETCHED_PTR`, then the Driver Manager caches the pointer value, for later use with **SQLFetchScroll**.
6. If *Attribute* is equal to `SQL_ATTR_ROW_STATUS_PTR`, then the Driver Manager caches the pointer value, for later use with **SQLFetchScroll** or **SQLSetPos**. No further rules of this section apply.
7. If *Attribute* is equal to `SQL_ATTR_FETCH_BOOKMARK_PTR`, the Driver Manager caches *ValuePtr* and it will use the cached value later in a call to **SQLFetchScroll**. No further rules of this section apply.
8. The Driver Manager calls **SQLSetStmtOption** as follows:
9. `SQLSetStmtOption(hstmt, fOption, vParam);`  
where *hstmt*, *fOption*, and *vParam* will be set to the values of *StatementHandle*, *Attribute*, and *ValuePtr*, respectively. Note that the *StringLength* argument is ignored.

If an ODBC 2.x driver supports character-string, driver-specific statement options, an ODBC 3.x application should call **SQLSetStmtOption** to set those options.

## Mappings For Handling Parameter Arrays

When the application calls:

```
SQLSetStmtAttr (StatementHandle, SQL_ATTR_PARAMSET_SIZE, Size, StringLength);
```

the Driver Manager calls:

```
SQLParamOptions (StatementHandle, Size, &RowCount);
```

The Driver Manager later returns a pointer to this variable when the application calls **SQLGetStmtAttr** to retrieve `SQL_ATTR_PARAMS_PROCESSED_PTR`. Note that the Driver Manager cannot change this internal variable until the statement handle is returned to the prepared or allocated state.

An ODBC 3.x application can call **SQLGetStmtAttr** to obtain the value of `SQL_ATTR_PARAMS_PROCESSED_PTR` even though it has not explicitly set the `SQL_DESC_ARRAY_SIZE` field in the APD. This situation could arise, for example, if the application has a generic routine that checks for the current "row" of parameters being processed when **SQLExecute** returns `SQL_NEED_DATA`. This routine is invoked regardless of whether the `SQL_DESC_ARRAY_SIZE` is 1 or is greater than 1. To account for this, the Driver Manager will need to define this internal variable regardless of whether or not the application has called **SQLSetStmtAttr** to set the `SQL_DESC_ARRAY_SIZE` field in APD. If `SQL_DESC_ARRAY_SIZE` has not been set, the Driver Manager has to make sure that this variable contains the value 1 prior to returning from **SQLExecDirect** or **SQLExecute**.

### Error Handling

In ODBC 3.x, calling **SQLFetch** or **SQLFetchScroll** populates the `SQL_DESC_ARRAY_STATUS_PTR` in the IRD, and the `SQL_DIAG_ROW_NUMBER` field of a given diagnostic record contains the number of the row in the rowset that this record pertains to. Using this, the application can correlate an error message with a given row position.

An ODBC 2.x driver will be unable to provide this functionality. However, it will provide error demarcation with `SQLSTATE 01S01` (Error in row). An ODBC 3.x application that is using **SQLFetch** or **SQLFetchScroll** while going against an ODBC 2.x driver needs to be aware of this fact. Note also that such an application will be unable to call **SQLGetDiagField** to actually get the `SQL_DIAG_ROW_NUMBER` field anyway. An ODBC 3.x application working with an ODBC 2.x driver will only be able to call **SQLGetDiagField** with a *DiagIdentifier* argument of `SQL_DIAG_MESSAGE_TEXT`, `SQL_DIAG_NATIVE`, `SQL_DIAG_RETURNCODE`, or `SQL_DIAG_SQLSTATE`. The ODBC 3.x Driver Manager maintains the diagnostic data structure when working with an ODBC 2.x driver, but the ODBC 2.x driver only returns these four fields.

When an ODBC 2.x application is working with an ODBC 2.x driver, if an operation can cause multiple errors to be returned by the Driver Manager, different errors may be returned by the ODBC 3.x Driver Manager than by the ODBC 2.x Driver Manager.

## Mappings For Bookmark Operations

The ODBC 3.x Driver Manager performs the following mappings when an ODBC 3.x application working with an ODBC 2.x driver performs bookmark operations.

## ***SQLBindCol***

When an ODBC 3.x application working with an ODBC 2.x driver calls **SQLBindCol** to bind to column 0 with *fCType* equal to SQL\_C\_VARBOOKMARK, the ODBC 3.x Driver Manager checks to see whether the *BufferLength* argument is less than 4, or greater than 4, and if so, returns SQLSTATE HY090 (Invalid string or buffer length). If the *BufferLength* argument is equal to 4, the Driver Manager calls **SQLBindCol** in the driver, after replacing *fCType* with SQL\_C\_BOOKMARK.

## ***SQLColAttribute***

When an ODBC 3.x application working with an ODBC 2.x driver calls **SQLColAttribute** with the *ColumnNumber* argument set to 0, the Driver Manager returns the following *FieldIdentifier* values:

| <b><i>FieldIdentifier</i></b>   | <b>Value</b>                                       |
|---------------------------------|--|
| SQL_DESC_AUTO_UNIQUE_VALUE      | SQL_FALSE  |
| SQL_DESC_CASE_SENSITIVE         | SQL_FALSE  |
| SQL_DESC_CATALOG_NAME           | "" (empty string)                                  |
| SQL_DESC_CONCISE_TYPE           | SQL_BINARY   |
| SQL_DESC_COUNT                  | The same value returned by <b>SQLNumResultCols</b> |
| SQL_DESC_DATETIME_INTERVAL_CODE | 0  |
| SQL_DESC_DISPLAY_SIZE           | 8  |
| SQL_DESC_FIXED_PREC_SCALE       | SQL_FALSE  |
| SQL_DESC_LABEL                  | "" (empty string)                                  |
| SQL_DESC_LENGTH                 | 0  |
| SQL_DESC_LITERAL_PREFIX         | "" (empty string)                                  |
| SQL_DESC_LITERAL_SUFFIX         | "" (empty string)                                  |
| SQL_DESC_LOCAL_TYPE_NAME        | "" (empty string)                                  |
| SQL_DESC_NAME                   | "" (empty string)                                  |
| SQL_DESC_NULLABLE               | SQL_NO_NULLS                                       |
| SQL_DESC_OCTET_LENGTH           | 4  |
| SQL_DESC_PRECISION              | 4  |
| SQL_DESC_SCALE                  | 0  |
| SQL_DESC_SCHEMA_NAME            | "" (empty string)                                  |
| SQL_DESC_SEARCHABLE             | SQL_PRED_NONE                                      |
| SQL_DESC_TABLE_NAME             | "" (empty string)                                  |
| SQL_DESC_TYPE                   | SQL_BINARY   |
| SQL_DESC_TYPE_NAME              | "" (empty string)                                  |
| SQL_DESC_UNNAMED                | SQL_UNNAMED  |
| SQL_DESC_UNSIGNED               | SQL_FALSE  |
| SQL_DESC_UPDATEABLE             | SQL_ATTR_READ_ONLY                                 |

## ***SQLDescribeCol***

When an ODBC 3.x application working with an ODBC 2.x driver calls **SQLDescribeCol** with the *ColumnNumber* argument set to 0, the Driver Manager returns the following values:

| Buffer            | Value             |
|-------------------|-------------------|
| ColumnName        | "" (empty string) |
| *NameLengthPtr    | 0                 |
| *DataTypePtr      | SQL_BINARY        |
| *ColumnSizePtr    | 4                 |
| *DecimalDigitsPtr | 0                 |
| *NullablePtr      | SQL_NO_NULLS      |

## ***SQLGetData***

When an ODBC 3.x application working with an ODBC 2.x driver makes the following call to **SQLGetData** to retrieve a bookmark:

```
SQLGetData(StatementHandle, 0, SQL_C_VARBOOKMARK, TargetValuePtr, BufferLength,  
StrLen_or_IndPtr)
```

the call is mapped to **SQLGetStmtOption** with an *fOption* of SQL\_GET\_BOOKMARK, as follows:

```
SQLGetStmtOption(hstmt, SQL_GET_BOOKMARK, TargetValuePtr)
```

where *hstmt* and *pvParam* are set to the values in *StatementHandle* and *TargetValuePtr*, respectively. The bookmark is returned in the buffer pointed to by the *pvParam* (*TargetValuePtr*) argument. The value in the buffer pointed to by the *StrLen\_or\_IndPtr* argument in the call to **SQLGetData** is set to 4.

This mapping is necessary to account for the case in which **SQLFetch** was called prior to the call to **SQLGetData**, and the ODBC 2.x driver did not support **SQLExtendedFetch**. In this case, **SQLFetch** would be passed through to the ODBC 2.x driver, in which case bookmark retrieval is not supported.

**SQLGetData** cannot be called multiple times in an ODBC 2.x driver to retrieve a bookmark in parts, so calling **SQLGetData** with the *BufferLength* argument set to a value less than 4 and the *ColumnNumber* argument set to 0 will return SQLSTATE HY090 (Invalid string or buffer length). **SQLGetData** can, however, be called multiple times to retrieve the same bookmark.

## ***SQLSetStmtAttr***

When an ODBC 3.x application working with an ODBC 2.x driver calls **SQLSetStmtAttr** to set the SQL\_ATTR\_USE\_BOOKMARKS attribute to SQL\_UB\_VARIABLE, the Driver Manager sets the attribute to SQL\_UB\_ON in the underlying ODBC 2.x driver

## **Calling SQLCloseCursor**

Because **SQLCloseCursor** is almost the same as **SQLFreeStmt** with SQL\_CLOSE, the Driver Manager does not map this function. Replacement functions are mapped so that existing ODBC 2.x applications can easily move to ODBC 3.x by using the new functions. Such a move makes it easier for such applications to begin using new ODBC 3.x functionality inside of conditional code in a modular fashion.

**SQLCloseCursor** does not represent any new functionality. An application does not gain any advantage by moving to **SQLCloseCursor** from **SQLFreeStmt** with **SQL\_CLOSE**.

## Calling SQLGetDiagField

When an ODBC 3.x application calls **SQLGetDiagField** in an ODBC 2.x driver, the driver will return **SQL\_SUCCESS** and the appropriate information in *\*DiagInfoPtr* if the *DiagIdentifier* argument is **SQL\_DIAG\_CLASS\_ORIGIN**, **SQL\_DIAG\_CLASS\_SUBCLASS\_ORIGIN**, **SQL\_DIAG\_CONNECTION\_NAME**, **SQL\_DIAG\_MESSAGE\_TEXT**, **SQL\_DIAG\_NATIVE**, **SQL\_DIAG\_NUMBER**, **SQL\_DIAG\_RETURNCODE**, **SQL\_DIAG\_SERVER\_NAME**, or **SQL\_DIAG\_SQLSTATE**. All other diagnostic fields will return **SQL\_ERROR**.

## Calling SQLSetPos

In ODBC 2.x, the pointer to the row status array was an argument to **SQLExtendedFetch**. The row status array was later updated by a call to **SQLSetPos**. Some drivers have relied on the fact that this array does not change between **SQLExtendedFetch** and **SQLSetPos**. In ODBC 3.x, the pointer to the status array is a descriptor field and so the application can easily change it to point to a different array. This can be a problem when an ODBC 3.x application is working with an ODBC 2.x driver, but is calling **SQLSetStmtAttr** to set the array status pointer and is calling **SQLFetchScroll** to fetch data. The Driver Manager maps it as a sequence of calls to **SQLExtendedFetch**. In the following code, an error would normally be raised when the Driver Manager maps the second **SQLSetStmtAttr** call when working with an ODBC 2.x driver:

```
SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_STATUS_PTR, rgfRowStatus, 0);
SQLFetchScroll(hstmt, fFetchType, iRow);
SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_STATUS_PTR, rgfRowStat1, 0);
SQLSetPos(hstmt, iRow, fOption, fLock);
```

The error would be raised if there were no way to change the row status pointer in ODBC 2.x between calls to **SQLExtendedFetch**. Instead, the Driver Manager performs the following steps when working with an ODBC 2.x driver:

1. Initializes an internal Driver Manager flag *fSetPosError* to TRUE.
2. When an application calls **SQLFetchScroll**, the Driver Manager sets *fSetPosError* to FALSE.
3. When the application calls **SQLSetStmtAttr** to set **SQL\_ATTR\_ROW\_STATUS\_PTR**, the Driver Manager sets *fSetPosError* equal to TRUE.
4. When the application calls **SQLSetPos**, with *fSetPosError* equal to TRUE, the Driver Manager raises **SQL\_ERROR** with **SQLSTATE** HY011 (Attribute cannot be set now) to indicate that the application attempted to call **SQLSetPos** after changing the row status pointer, but prior to calling **SQLFetchScroll**.

## Cursor Library Operations

If an application working with an ODBC 2.x driver makes calls to the ODBC 3.x cursor library, the application may be able to use ODBC 3.x features that are not supported by the ODBC 2.x driver. An

application writer should be careful how these features are used, however. Use of the ODBC 3.x cursor library does not make an ODBC 2.x driver into an ODBC 3.x driver.

## Mapping the Cursor Attributes1 Information Types

When an ODBC 3.x application calls **SQLGetInfo** in an ODBC 2.x driver with the **SQL\_XXXX\_CURSOR\_ATTRIBUTES1** information type (for dynamic, forward-only, keyset-driver, or static cursors), the setting of the bits returned by Driver Manager depends upon what the ODBC 2.x driver returns for the corresponding ODBC 2.x information types. The bits are set as follows:

| Bit in <b>SQL_XXXX_CURSOR_ATTRIBUTES1</b>   | Cursor type                    | ODBC 2.x information type        |
|---|--------------------------------|----------------------------------|
| <b>SQL_CA1_NEXT</b>   | All                            | <b>SQL_FETCH_DIRECTION</b>       |
| <b>SQL_CA1_ABSOLUTE</b><br><b>SQL_CA1_RELATIVE</b><br><b>SQL_CA1_BOOKMARK</b>   | Dynamic, keyset-driver, static | <b>SQL_FETCH_DIRECTION</b>       |
| <b>SQL_CA1_LOCK_NO_CHANGE</b><br><b>SQL_CA1_LOCK_UNLOCK</b><br><b>SQL_CA1_LOCK_EXCLUSIVE</b>                          | Dynamic, keyset-driver, static | <b>SQL_LOCK_TYPES</b>            |
| <b>SQL_CA1_POSITIONED_UPDATE</b><br><b>SQL_CA1_POSITIONED_DELETE</b><br><b>SQL_CA1_SELECT_FOR_UPDATE</b>              | All                            | <b>SQL_POSITIONED_STATEMENTS</b> |
| <b>SQL_CA1_POS_POSITION</b><br><b>SQL_CA1_POS_DELETE</b><br><b>SQL_CA1_POS_REFRESH</b><br><b>SQL_CA1_POS_BULK_ADD</b> | Dynamic, keyset-driver, static | <b>SQL_POS_OPERATIONS</b>        |



## SQL\_NO\_DATA

When an ODBC 3.x application calls **SQLExecDirect**, **SQLExecute**, or **SQLParamData** in an ODBC 2.x driver to execute a searched update or delete statement that does not affect any rows at the data source, the driver should return **SQL\_SUCCESS**, not **SQL\_NO\_DATA**. When an ODBC 2.x or ODBC 3.x application working with an ODBC 3.x driver calls **SQLExecDirect**, **SQLExecute**, or **SQLParamData** with the same result, the ODBC 3.x driver should return **SQL\_NO\_DATA**.

## Writing ODBC 3.x Drivers

The following table shows function support in an ODBC 3.x driver and an ODBC application, and the mapping performed by the Driver Manager when the functions are called against an ODBC 3.x driver.

| Function                   | Supported by an ODBC 3.x driver? | Supported by an ODBC 3.x application? | Mapped/supported by the ODBC 3.x Driver Manager to an ODBC 3.x driver? |
|----------------------------|----------------------------------|---------------------------------------|--|
| <b>SQLAllocConnect</b>     | No                               | No [1]                                | Yes  |
| <b>SQLAllocEnv</b>         | No                               | No [1]                                | Yes  |
| <b>SQLAllocHandle</b>      | Yes                              | Yes                                   | No   |
| <b>SQLAllocStmt</b>        | No                               | No [1]                                | Yes  |
| <b>SQLBindCol</b>          | Yes                              | Yes                                   | No   |
| <b>SQLBindParam</b>        | No                               | Yes [2]                               | Yes  |
| <b>SQLBindParameter</b>    | Yes                              | Yes                                   | No   |
| <b>SQLBrowseConnect</b>    | Yes                              | Yes                                   | No   |
| <b>SQLCancel</b>           | Yes                              | Yes                                   | No   |
| <b>SQLCloseCursor</b>      | Yes                              | Yes                                   | No   |
| <b>SQLColAttribute</b>     | Yes                              | Yes                                   | No   |
| <b>SQLColAttributes</b>    | No [3]                           | No                                    | Yes  |
| <b>SQLColumnPrivileges</b> | Yes                              | Yes                                   | No   |
| <b>SQLColumns</b>          | Yes                              | Yes                                   | No   |
| <b>SQLConnect</b>          | Yes                              | Yes                                   | No   |
| <b>SQLCopyDesc</b>         | Yes                              | Yes                                   | Yes [4]  |
| <b>SQLDataSources</b>      | No                               | Yes                                   | Yes  |
| <b>SQLDescribeCol</b>      | Yes                              | Yes                                   | No   |
| <b>SQLDescribeParam</b>    | Yes                              | Yes                                   | No   |
| <b>SQLDisconnect</b>       | Yes                              | Yes                                   | No   |
| <b>SQLDriverConnect</b>    | Yes                              | Yes                                   | No   |
| <b>SQLDrivers</b>          | No                               | Yes                                   | Yes  |
| <b>SQLEndTran</b>          | Yes                              | Yes                                   | No   |
| <b>SQLError</b>            | No                               | No [1]                                | Yes  |
| <b>SQLExecDirect</b>       | Yes                              | Yes                                   | No   |

|                            |        |         |     |
|----------------------------|--------|---------|-----|
| <b>SQLExecute</b>          | Yes    | Yes     | No  |
| <b>SQLExtendedFetch</b>    | Yes    | No      | No  |
| <b>SQLFetch</b>            | Yes    | Yes     | No  |
| <b>SQLFetchScroll</b>      | Yes    | Yes     | No  |
| <b>SQLForeignKeys</b>      | Yes    | Yes     | No  |
| <b>SQLFreeConnect</b>      | No     | Yes [1] | Yes |
| <b>SQLFreeEnv</b>          | No     | Yes [1] | Yes |
| <b>SQLFreeHandle</b>       | Yes    | Yes     | No  |
| <b>SQLFreeStmt</b>         | Yes    | Yes     | No  |
| <b>SQLGetConnectAttr</b>   | Yes    | Yes     | No  |
| <b>SQLGetConnectOption</b> | No [5] | No [1]  | Yes |
| <b>SQLGetCursorName</b>    | Yes    | Yes     | No  |
| <b>SQLGetData</b>          | Yes    | Yes     | No  |
| <b>SQLGetDescField</b>     | Yes    | Yes     | No  |
| <b>SQLGetDescRec</b>       | Yes    | Yes     | No  |
| <b>SQLGetDiagField</b>     | Yes    | Yes     | No  |
| <b>SQLGetDiagRec</b>       | Yes    | Yes     | No  |
| <b>SQLGetEnvAttr</b>       | Yes    | Yes     | No  |
| <b>SQLGetFunctions</b>     | No [6] | Yes     | Yes |
| <b>SQLGetInfo</b>          | Yes    | Yes     | No  |
| <b>SQLGetStmtAttr</b>      | Yes    | Yes     | No  |
| <b>SQLGetStmtOption</b>    | No [5] | No [1]  | Yes |
| <b>SQLGetTypeInfo</b>      | Yes    | Yes     | No  |
| <b>SQLMoreResults</b>      | Yes    | Yes     | No  |
| <b>SQLNativeSql</b>        | Yes    | Yes     | No  |
| <b>SQLNumParams</b>        | Yes    | Yes     | No  |
| <b>SQLNumResultCols</b>    | Yes    | Yes     | No  |
| <b>SQLParamData</b>        | Yes    | Yes     | No  |
| <b>SQLParamOptions</b>     | No     | No      | Yes |
| <b>SQLPrepare</b>          | Yes    | Yes     | No  |
| <b>SQLPrimaryKeys</b>      | Yes    | Yes     | No  |
| <b>SQLProcedureColumns</b> | Yes    | Yes     | No  |
| <b>SQLProcedures</b>       | Yes    | Yes     | No  |
| <b>SQLPutData</b>          | Yes    | Yes     | No  |
| <b>SQLRowCount</b>         | Yes    | Yes     | No  |
| <b>SQLSetConnectAttr</b>   | Yes    | Yes     | No  |
| <b>SQLSetConnectOption</b> | No [5] | No [1]  | Yes |
| <b>SQLSetCursorName</b>    | Yes    | Yes     | No  |

|                           |        |        |     |
|---------------------------|--------|--------|-----|
| <b>SQLSetDescField</b>    | Yes    | Yes    | No  |
| <b>SQLSetDescRec</b>      | Yes    | Yes    | No  |
| <b>SQLSetEnvAttr</b>      | Yes    | Yes    | No  |
| <b>SQLSetPos</b>          | Yes    | Yes    | No  |
| <b>SQLSetParam</b>        | No     | No     | Yes |
| <b>SQLSetScrollOption</b> | Yes    | Yes    | No  |
| <b>SQLSetStmtAttr</b>     | Yes    | Yes    | No  |
| <b>SQLSetStmtOption</b>   | No [5] | No [1] | Yes |
| <b>SQLSpecialColumns</b>  | Yes    | Yes    | No  |
| <b>SQLStatistics</b>      | Yes    | Yes    | No  |
| <b>SQLTablePrivileges</b> | Yes    | Yes    | No  |
| <b>SQLTables</b>          | Yes    | Yes    | No  |
| <b>SQLTransact</b>        | No     | No [1] | Yes |

[1] This function is deprecated in ODBC 3.x. ODBC 3.x applications should not use this function. However, an X/Open or ISO CLI – compliant application can call this function.

[2] ODBC 3.x applications should use **SQLBindParameter** instead of **SQLBindParam**. However, an X/Open or ISO CLI – compliant application can call this function.

[3] Driver writers note that the ODBC 2.x column attributes SQL\_COLUMN\_PRECISION, SQL\_COLUMN\_SCALE, and SQL\_COLUMN\_LENGTH must be supported with **SQLColAttribute**.

[4] Note that **SQLCopyDesc** is partially implemented by the Driver Manager when a descriptor is being copied across connections that belong to different drivers. Drivers are required to support **SQLCopyDesc** across two of their own connections. Functions such as **SQLDrivers**, which are implemented solely by the Driver Manager, do not show up on this list.

[5] Under certain circumstances, drivers may need to support this function. See the reference manual page for this function for more information.

[6] The driver may choose to support **SQLGetFunctions** if the functions that it supports varies from connection to connection.

## ODBC in Windows

The following items apply only to ODBC running in Windows NT and Windows 95 operating systems.

### Standards-Compliant Applications and Drivers

A standards-compliant application or driver is one that conforms to the X/Open CAE Specification "Data Management: SQL Call-Level Interface (CLI)," and the ISO/IEC 9075-3:1995 (E) Call-Level Interface (SQL/CLI).

ODBC 3.x guarantees that:

- An application written to the X/Open and ISO CLI specifications will work with an ODBC 3.x driver or a standards-compliant driver when it is compiled with the ODBC 3.x header files and linked with ODBC 3.x libraries, and when it gains access to the driver through the ODBC 3.x Driver Manager.
- A driver written to the X/Open and ISO CLI specifications will work with an ODBC 3.x application or a standards-compliant application when it is compiled with the ODBC 3.x header files and linked with ODBC 3.x libraries, and when the application gains access to the driver through the ODBC 3.x Driver Manager.
- Standards-compliant applications and drivers are compiled with the ODBC\_STD compile flag.
- Standards-compliant applications exhibit the following behavior:
- If a standards-compliant application calls **SQLAllocEnv** (which may occur because **SQLAllocEnv** is a valid function in the X/Open and ISO CLI), the call is mapped to **SQLAllocHandleStd** at compile time. As a result, at run time, the application calls **SQLAllocHandleStd**. During the course of processing this call, the Driver Manager sets the SQL\_ATTR\_ODBC\_VERSION environment attribute to SQL\_OV\_ODBC3. A call to **SQLAllocHandleStd** is equivalent to a call to **SQLAllocHandle** with a *HandleType* of SQL\_HANDLE\_ENV and a call to **SQLSetEnvAttr** to set SQL\_ATTR\_ODBC\_VERSION to SQL\_OV\_ODBC3.
- If a standards-compliant application calls **SQLBindParam** (which may occur because **SQLBindParam** is a valid function in the X/Open and ISO CLI), the ODBC 3.x Driver Manager maps the call to the equivalent call in **SQLBindParameter** (see “SQLBindParamMappings” in Appendix G, "Driver Guidelines for Backward Compatibility" contained on the Microsoft Web site (ODBC Programmer's Guide)).
- To align with the ISO CLI, the ODBC 3.x header files contain aliases for information types used in calls to **SQLGetInfo**. A standards-compliant application can use these aliases instead of the ODBC 3.x information types. For more information, see the next section, “Header Files.”
- A standards-compliant application must verify that all features it supports are supported in the driver it will work with. Setting the SQL\_ATTR\_CURSOR\_SCROLLABLE statement attribute to SQL\_SCROLLABLE, and setting the SQL\_ATTR\_CURSOR\_SENSITIVITY statement attribute to SQL\_INSENSITIVE or SQL\_SENSITIVE are capabilities that are available as optional features in the standards, but are not included in the ODBC 3.x Core level, so may not be supported by all ODBC 3.x drivers. If a standards-compliant application uses these capabilities, it should verify that the driver that it will work with supports them.

## Header Files

The `Sql.h` header file contains prototypes for the functions and features in the Core ODBC Interface conformance level. The `Sqlext.h` header file contains prototypes for the functions and features in the Level

1 and Level 2 API conformance levels. The `Sqotypes.h` header file contains type definitions and indicators for the SQL data types.

The header files all contain a **#define**, `ODBCVER`, that an application or driver can set to be compiled for different versions of ODBC.

To align with the ISO CLI and X/Open CLI, the header files contain aliases for the information types used in calls to **SQLGetInfo**. In the following table, the column "ODBC name" indicates the ODBC name for the information type in the Part II PDF file, "ODBC API Reference" available on the Solid Web site. The column "Alias in header file" indicates the name that is used in the ISO CLI and the X/Open CLI. The actual numeric value of these manifest names is the same in both ODBC and the standard CLIs. These aliases enable a standards-compliant application or driver to compile with the ODBC 3.x header files.

These aliases include expansions of abbreviations in the ODBC names so that the names are more understandable. "MAX" is expanded to "MAXIMUM", "LEN" to "LENGTH", "MULT" to "MULTIPLE", "OJ" to "OUTER\_JOIN", and "TXN" to "TRANSACTION."

| ODBC name                     | Alias in header file              |
|-------------------------------|-----------------------------------|
| SQL_MAX_CATALOG_NAME_LEN      | SQL_MAXIMUM_CATALOG_NAME_LENGTH   |
| SQL_MAX_COLUMN_NAME_LEN       | SQL_MAXIMUM_COLUMN_NAME_LENGTH    |
| SQL_MAX_COLUMNS_IN_GROUP_BY   | SQL_MAXIMUM_COLUMNS_IN_GROUP_BY   |
| SQL_MAX_COLUMNS_IN_ORDER_BY   | SQL_MAXIMUM_COLUMNS_IN_ORDER_BY   |
| SQL_MAX_COLUMNS_IN_SELECT     | SQL_MAXIMUM_COLUMNS_IN_SELECT     |
| SQL_MAX_COLUMNS_IN_TABLE      | SQL_MAXIMUM_COLUMNS_IN_TABLE      |
| SQL_MAX_CONCURRENT_ACTIVITIES | SQL_MAXIMUM_CONCURRENT_ACTIVITIES |
| SQL_MAX_CURSOR_NAME_LEN       | SQL_MAXIMUM_CURSOR_NAME_LENGTH    |
| SQL_MAX_DRIVER_CONNECTIONS    | SQL_MAXIMUM_DRIVER_CONNECTIONS    |
| SQL_MAX_IDENTIFIER_LEN        | SQL_MAXIMUM_IDENTIFIER_LENGTH     |
| SQL_MAX_SCHEMA_NAME_LEN       | SQL_MAXIMUM_SCHEMA_NAME_LENGTH    |
| SQL_MAX_STATEMENT_LEN         | SQL_MAXIMUM_STATEMENT_LENGTH      |
| SQL_MAX_TABLE_NAME_LEN        | SQL_MAXIMUM_TABLE_NAME_LENGTH     |
| SQL_MAX_TABLES_IN_SELECT      | SQL_MAXIMUM_TABLES_IN_SELECT      |
| SQL_MAX_USER_NAME_LEN         | SQL_MAXIMUM_USER_NAME_LENGTH      |
| SQL_MULT_RESULT_SETS          | SQL_MULTIPLE_RESULT_SETS          |
| SQL_OJ_CAPABILITIES           | SQL_OUTER_JOIN_CAPABILITIES       |
| SQL_TXN_CAPABLE               | SQL_TRANSACTION_CAPABLE           |
| SQL_TXN_ISOLATION_OPTION      | SQL_TRANSACTION_ISOLATION_OPTION  |

## CString Class

Because objects of the CString class in Microsoft<sup>®</sup> Visual C++<sup>®</sup> are signed and string arguments in ODBC functions are unsigned, applications that pass CString objects to ODBC functions without casting them will receive compiler warnings.

## Creating and Terminating Threads

Multithread applications that use ODBC should call the Microsoft Visual C++ Run-Time Library functions **\_beginthread** and **\_endthread** (or **\_beginthreadex** and **\_endthreadex**) to create and terminate threads that call the ODBC Driver Manager. If applications call the Windows NT<sup>®</sup> functions **CreateThread** and **EndThread** instead, memory leaks will occur because the Driver Manager and some ODBC drivers call C run-time functions that will not work on a thread created by calling **CreateThread**. For more information, see the Microsoft Windows documentation.