1 *Direct Review*

2 **Data Management:**

3 **X/Open Database Connectivity (XDBC), Version 2**

4 *X/Open Company Ltd.*

/

5 1996, X/Open Company Limited

6 All rights reserved. No part of this publication may be reproduced, stored in a retrieval system,
7 or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or
8 otherwise, without the prior permission of the copyright owners.

9 Direct Review

10 Data Management: X/Open Database Connectivity (XDBC), Version 2

11 X/Open Document Number: To be defined

12 This is a working draft that was typeset in Palatino by **Spike, 76 Scrabble Road, Brentwood,**
13 **New Hampshire 03833 USA** for the benefit of X/Open Company Ltd. The typesetting
14 technology differs from that used by X/Open; this will cause minor stylistic differences that do
15 not warrant comments by readers. Published by X/Open.

16 Any comments relating to the material contained in this document may be submitted to X/Open
17 at:

18 X/Open Company Limited
19 Apex Plaza
20 Forbury Road
21 Reading
22 Berkshire, RG1 1AX
23 United Kingdom

24 or by Electronic Mail to:

25 XoSpecs@xopen.co.uk

Contents

Chapter 1 Introduction 1

1.1 Development of XDBC 1
1.2 This Issue 2
1.3 Relation to Other X/Open Documents 7
1.3.1 Conceptual Differences from Embedded SQL 7
1.4 Relation to Standards 8
1.5 Compliance Policy 9
1.5.1 Language Binding 9
1.5.2 SQL Statement Text 9
1.5.3 Distributed Transaction Delimitation 10
1.6 Compliance Terminology 11
1.7 XDBC Compliance Levels 13
1.8 SQL Registry 21

Chapter 3 XDBC Architecture 23

3.1 XDBC Implementation 24
3.2 Implementation Architecture 25
3.3 Applications 26
3.4 Data Sources 27
3.5 Client and Server 28
3.6 System Information 30
3.7 Tables and Views 31

Chapter 4 Fundamentals 33

4.1 Handles 34
4.1.1 Environment Handles 35
4.1.2 Connection Handles 35
4.1.3 Statement Handles 36
4.1.4 Descriptor Handles 36
4.2 State Transitions 37
4.3 Buffers 38
4.3.1 Deferred Buffers 39
4.3.2 Allocating and Freeing Buffers 39
4.3.3 Using Data Buffers 40
4.3.4 Data Buffer Type 41
4.3.5 Using Length/Indicator Values 42
4.3.6 Data Length, Buffer Length, and Truncation 43
4.3.7 Character Data and C Strings 44
4.4 Data Types in XDBC 46
4.4.1 Type Identifiers 46
4.4.2 SQL Data Types in XDBC 46
4.4.3 C Data Types in XDBC 47
4.4.4 Data Type Conversions 48

68 4.5 Environment, Connection, and Statement Attributes 49

69 **Chapter 5 Basic Application Steps 51**
70 5.1 Basic Control Flow 52
71 5.2 Example Control Flow for SQL Statement Processing 54

72 **Chapter 6 Connecting to a Data Source 57**
73 6.1 Allocating the Environment Handle 58
74 6.2 Allocating a Connection Handle 59
75 6.3 Connection Attributes60
76 6.4 Establishing a Connection 61
77 6.4.1 Default Data Source61
78 6.4.2 Connecting with SQLConnect() 61
79 6.4.3 Connection String 61
80 6.4.4 Connecting with SQLDriverConnect() 62
81 6.4.5 Connecting with SQLBrowseConnect() 62
82 6.5 Disconnecting from a Data Source 64

83 **Chapter 7 Catalog Functions 65**
84 7.1 Uses of Catalog Data 66
85 7.2 Catalog Functions 67
86 7.3 Data Returned by Catalog Functions 68
87 7.3.1 COLUMN_DEF Column 68
88 7.4 Arguments in Catalog Functions 69

89 **Chapter 8 SQL Statements 75**
90 8.1 Building SQL Statements 76
91 8.1.1 Hard-Coded SQL Statements76
92 8.1.2 SQL Statements Built at Run Time 78
93 8.1.3 SQL Statements Entered by the User79
94 8.2 Interoperability of SQL Statements 80
95 8.2.1 Constructing Interoperable SQL Statements 81
96 8.3 Escape Clauses 84
97 8.3.1 Date, Time and Timestamp Literals 84
98 8.3.2 Interval Literals 86
99 8.3.3 Scalar Function Calls86
100 8.3.4 LIKE Predicate Escape Character 86
101 8.3.5 Outer Joins 87
102 8.3.6 Procedure Calls 88

103 **Chapter 9 Executing Statements 91**
104 9.1 Allocating a Statement Handle92
105 9.2 Statement Attributes 93
106 9.2.1 Temporary Changes to Statement Attribute Value 93
107 9.3 Executing a Statement94
108 9.3.1 Direct Execution 95
109 9.3.2 Prepared Execution 96
110 9.3.3 Procedures 97
111 9.3.4 Batches of SQL Statements 99
112 9.3.5 Executing Catalog Functions101
113 9.4 Statement Parameters102

114 9.4.1 Binding Parameters 102
115 9.4.2 Setting Parameter Values 104
116 9.4.3 Sending Long Data 105
117 9.4.4 Procedure Parameters 109
118 9.4.5 Arrays of Parameter Values 109
119 9.5 Asynchronous Execution 116
120 9.5.1 Levels of Asynchronous Support 116
121 9.5.2 Enabling Asynchrony 118
122 9.5.3 Steps in Asynchronous Execution 118
123 9.5.4 Cancelling an Asynchronously-executing Function 119
124 9.5.5 Example Asynchronous Control Flow 121
125 9.5.6 Asynchrony Combined with Other XDBC Features 122
126 9.5.7 Limits on Concurrency 123
127 9.5.8 Example Asynchrony Code 123
128 9.6 Freeing a Statement Handle 124

129 **Chapter 10 Retrieving Results (Basic) 125**
130 10.1 Was a Result Set Created? 126
131 10.2 Result Set Metadata 127
132 10.2.1 How is Metadata Used? 127
133 10.2.2 SQLDescribeCol() and SQLColAttribute() 127
134 10.3 Binding Result Set Columns 129
135 10.3.1 Overview 129
136 10.3.2 Using SQLBindCol() 130
137 10.4 Fetching Data 133
138 10.4.1 Cursors 133
139 10.4.2 Fetching a Row of Data 133
140 10.4.3 Row Status 134
141 10.4.4 Getting Long Data 135
142 10.5 Closing the Cursor 137

143 **Chapter 11 Retrieving Results (Advanced) 139**
144 11.1 Multi-row Fetch 140
145 11.1.1 Binding Styles 141
146 11.1.2 Additional Result Information 145
147 11.1.3 Using Multi-row Fetch 145
148 11.2 Scrollable Cursors 147
149 11.2.1 Scrollable Cursor Types 147
150 11.2.2 Using Scrollable Cursors 150
151 11.2.3 Relative and Absolute Scrolling 153
152 11.2.4 Bookmarks 154
153 11.3 Multiple Results 156

154 **Chapter 12 Updating Data 157**
155 12.1 UPDATE, DELETE, and INSERT Statements 158
156 12.1.1 Positioned UPDATE and DELETE 158
157 12.1.2 Code Example 158
158 12.1.3 Simulating Positioned UPDATE and DELETE 160
159 12.2 Determining the Number of Affected Rows 162
160 12.3 Using SQLSetPos() 163
161 12.3.1 Updating Rows with SQLSetPos() 163

162 12.3.2 Deleting Rows with SQLSetPos() 164
163 12.4 Using SQLBulkOperations() 165
164 12.4.1 Updating Rows by Bookmark with SQLBulkOperations() 165
165 12.4.2 Deleting Rows by Bookmark with SQLBulkOperations() 166
166 12.4.3 Inserting Rows with SQLBulkOperations() 166
167 12.4.4 Long Data and SQLBulkOperations()/SQLSetPos() 167
168 12.4.5 Code Example 168

169 **Chapter 13 Descriptors 170**
170 13.1 Types of Descriptor 171
171 13.2 Descriptor Fields 173
172 13.2.1 Count of Records 174
173 13.2.2 Bound Descriptor Records 174
174 13.3 Operations on Descriptors 175
175 13.3.1 Concise Functions 176
176 13.4 Deferred Fields 178

177 **Chapter 14 Transactions 181**
178 14.1 Transaction Support in XDBC 182
179 14.1.1 Determining Level of Support 182
180 14.1.2 Commit Mode and Transaction Completion 182
181 14.1.3 Side-effects of Transaction Completion 184
182 14.2 Transaction Isolation 186
183 14.2.1 Serializability 186
184 14.2.2 Transaction Isolation Levels186
185 14.2.3 Setting the Transaction Isolation Level 188
186 14.2.4 Scrollable Cursors and Transaction Isolation 188
187 14.3 Concurrency Control191
188 14.3.1 Concurrency Types 191
189 14.3.2 Optimistic Concurrency 192

190 **Chapter 15 Diagnostics 193**
191 15.1 Return Codes194
192 15.2 Diagnostic Records 195
193 15.3 SQLSTATE 196
194 15.4 Application Usage 200
195 15.4.1 Per-row Diagnostics201

196 **Chapter 20 Interface Overview 203**

197 **Chapter 21 Reference Manual Pages 207**
198 *SQLAllocHandle*() 208
199 *SQLBindCol*() 212
200 *SQLBindParam*() 220
201 *SQLBindParameter*() 221
202 *SQLBrowseConnect*() 234
203 *SQLBulkOperations*() 239
204 *SQLCancel*() 247
205 *SQLCloseCursor*() 250
206 *SQLColAttribute*() 252
207 *SQLColumnPrivileges*() 256

208 *SQLColumns*() 261
209 *SQLConnect*() 269
210 *SQLCopyDesc*() 272
211 *SQLDataSources*() 275
212 *SQLDescribeCol*() 277
213 *SQLDescribeParam*() 281
214 *SQLDisconnect*() 284
215 *SQLDriverConnect*() 286
216 *SQLDrivers*() 291
217 *SQLEndTran*() 294
218 *SQLExecDirect*() 297
219 *SQLExecute*() 302
220 *SQLFetch*() 307
221 *SQLFetchScroll*() 316
222 *SQLForeignKeys*() 326
223 *SQLFreeHandle*() 333
224 *SQLFreeStmt*() 336
225 *SQLGetConnectAttr*() 338
226 *SQLGetCursorName*() 341
227 *SQLGetData*() 343
228 *SQLGetDescField*() 350
229 *SQLGetDescRec*() 354
230 *SQLGetDiagField*() 358
231 *SQLGetDiagRec*() 364
232 *SQLGetEnvAttr*() 367
233 *SQLGetFunctions*() 369
234 *SQLGetInfo*() 372
235 *SQLGetStmtAttr*() 407
236 *SQLGetTypeInfo*() 410
237 *SQLMoreResults*() 417
238 *SQLNativeSql*() 420
239 *SQLNumParams*() 423
240 *SQLNumResultCols*() 425
241 *SQLParamData*() 427
242 *SQLPrepare*() 430
243 *SQLPrimaryKeys*() 434
244 *SQLProcedureColumns*() 438
245 *SQLProcedures*() 445
246 *SQLPutData*() 449
247 *SQLRowCount*() 454
248 *SQLSetConnectAttr*() 456
249 *SQLSetCursorName*() 462
250 *SQLSetDescField*() 464
251 *SQLSetDescRec*() 484
252 *SQLSetEnvAttr*() 488
253 *SQLSetPos*() 491
254 *SQLSetStmtAttr*() 503
255 *SQLSpecialColumns*() 516
256 *SQLStatistics*() 522
257 *SQLTablePrivileges*() 528
258 *SQLTables*() 533

259 **Appendix A Diagnostic Reference Information 539**
260 A.1 Class and Subclass Origin 539
261 A.2 SQLSTATECross-reference (Non-normative) 540

262 **Appendix B State Tables 547**
263 B.1 Environment State Transitions548
264 B.2 Connection State Transitions 549
265 B.3 Statement Transitions550
266 B.3.1 Data-at-execute Dialogue 553
267 B.4 Asynchrony State Transitions 554
268 B.5 Descriptor State Transitions 554

269 **Appendix D Data Types 555**
270 D.1 SQL Data Types 556
271 D.2 C Data Types560
272 D.2.1 Date/time Structures 561
273 D.2.2 64-bit Integer Structures 561
274 D.3 Attributes of Data Types 562
275 D.3.1 Column Size562
276 D.3.2 Decimal Digits 564
277 D.3.3 Transfer Octet Length 565
278 D.3.4 Display Size567
279 D.3.5 Constraints on Date/time Values 568
280 D.4 Interval Data Types 569
281 D.5 Using Data Type Identifiers 572
282 D.6 Converting Data from SQL to C Data Types 576
283 D.6.1 SQL to C Data Conversion Examples585
284 D.7 Converting Data from C to SQL Data Types 587
285 D.7.1 C to SQL Data Conversion Examples597

286 **Appendix F Scalar Functions 599**
287 F.1 String Functions 601
288 F.2 Numeric Functions 603
289 F.3 Time, Date, and Interval Functions 605
290 F.4 System Functions 608
291 F.5 Explicit Data Type Conversion609

292 **Appendix I Driver Manager Implementation (Optional) 613**
293 I.1 Introduction 614
294 I.1.1 The Driver Manager614
295 I.1.2 Drivers 615
296 I.2 Choosing a Data Source 617
297 I.3 Role of the Driver Manager in the Connection Process 620
298 I.4 Other Architectural Issues 621
299 I.5 Implementation of the Diagnostic Area622
300 I.5.1 Role of the Driver Manager 623
301 I.5.2 Role of the Driver 624
302 I.6 Changes to the Reference Manual Pages 625
303 I.6.1 Information on Specific XDBC Functions 625
304 I.6.2 SQLSTATEsof Specific XDBC Functions 634

305 Glossary 639

306 ## List of Figures

307 5-1 Initiation, Termination and Transaction Completion 52
308 5-2 Example Control Flow for Statement Processing54
309 7-1 Sales Order Database Structure 65
310 9-1 Providing Parameter Data at Execute Time 106
311 9-2 Example Control Flow for Asynchrony 121
312 11-1 Application Buffer for Column-wise Binding 141
313 11-2 Application Buffer for Row-wise Binding 143
314 11-3 Fetching next, prior, first, and last row-sets 151
315 11-4 Fetching absolute, relative, and bookmarked row-sets 152

316 ## List of Tables

317 7-1 Interpretation of String Arguments of Catalog Functions70
318 9-1 Functions for which Asynchrony is Permitted 117
319 13-1 The Four Types of Descriptor 171
320 13-2 List of Descriptor Header Fields173
321 13-3 List of Descriptor Record Fields173
322 13-4 Descriptor Fields that Relate to Statement Attributes 174
323 20-1 XDBC Functions 203
324 B-1 State Table for Connection Handles 549
325 B-2 State Table for Statement Handles 551
326 B-3 State Table for Statement Handles (Data-at-Execute Dialogue) 553
327 B-4 State Table for Asynchrony 554

328 *Chapter 1*

329
Introduction

330 The X/Open Database Connectivity (XDBC) interface is an application programming interface
331 (API) for database access. XDBC is an alternative invocation technique to dynamic SQL that
332 provides essentially equivalent operations. XDBC is a set of functions that application programs
333 call directly using normal function call facilities, whereas embedded SQL is typically converted
334 by a preprocessor.

335 The definition of XDBC relies heavily on the referenced X/Open **SQL** specification, which
336 defines a database and the intended result of executing SQL statements.

337 This chapter traces the development of XDBC, explains its relationship to the X/Open **SQL**
338 specification and the ISO SQL standard, and defines terms used to gauge compliance with this
339 specification.

340
1.1 Development of XDBC

341 SQL was originally developed as a way to embed, in an application program, static or dynamic
342 operations on a database. Embedded SQL code is typically converted by an implementation-
343 specific preprocessor into code that is compiled and executed.

344 Dynamic SQL makes SQL more flexible and applies it to cases where the database operations are |
345 not defined when the application program is written. For example, in fourth-generation |
346 languages, these operations are often based on interaction with the user. Dynamic SQL lets SQL
347 statement text reside in host-language character strings. The application generates them and the
348 SQL implementation interprets them dynamically during the course of the program's execution.
349 Dynamic SQL is still an embedded invocation technique and still typically works through a
350 preprocessor. The X/Open **SQL** specification specifies both static and dynamic SQL.

351 XDBC advances SQL further in the following areas:

352 • **Portability and interoperability**

353 Use of XDBC lets database applications be written to more easily interwork with a variety of |
354 databases. Application writers can produce portable object modules containing SQL |
355 database operations (''shrink-wrapped applications''), provided the operating system |
356 provides a mechanism to dynamically load libraries.[1] The following features facilitate |
357 portability and interoperability:

358 — **Preprocessor-independence**

359 Embedded SQL's assumption of a preprocessor typically requires that portable
360 applications are distributed as source code. Developers are reluctant to disclose |
361 proprietary source code. XDBC does not require implementation-specific transformations |
362 on source code at compile-time; implementation-specific features and added value reside |
363 in the XDBC run-time library.

364 ________________ |

365 1. Binary portability of object modules may be restricted by factors outside the scope of this document, such as choice of the |
processor, operating system and, sometimes, memory model. |

366 — **Standard coding using escape sequences**

367 Implementations of embedded SQL language vary widely in their approach to certain
368 useful features. XDBC defines a standard escape syntax for these features, which can be
369 translated to the SQL dialect the data source accepts.

370 — **Support for optional two-level architecture**

371 An XDBC implementation that implements the optional Driver Manager architecture (see
372 Appendix I) lets the application select any supported data source at run time without any
373 recompilation or modification.

374 • **Client/server architecture**

375 Databases are increasingly structured as clients and servers. Both the ISO SQL standard and
376 the X/Open **SQL** specification conceptualise SQL in terms of client and server operations.
377 When clients and servers are separated, the application writer may not know what
378 operations it is to perform or even the structure of the database.

379 XDBC is ideally suited for a client/server environment, in which the target database is not
380 known when the application program is built. XDBC provides the same syntax to execute
381 any SQL data definition or data manipulation statement.

382 • **Concurrent processing**

383 Applications are increasingly specifying concurrent processing, including concurrent
384 database operations. The existence of global data areas in SQL raises the question of the
385 scope and visibility of each change to such data.

386 XDBC eliminates global data areas, associating all implementation data that is accessible to
387 the application with a specific handle that the implementation passes to the application.

388 • **Distributed transaction processing (DTP)**

389 DTP distributes work between processors, with the guarantee that either all operations or
390 none are committed (global atomicity). The referenced X/Open **DTP**, **XA** and **Transaction**
391 **Demarcation** specifications address this topic.

392 The X/Open **SQL** specification delimits transactions using the COMMIT and ROLLBACK
393 statements. A transaction begins implicitly when the application operates on a database. The
394 X/Open **SQL** specification mentions a technique to permit SQL work to be completed
395 atomically with non-SQL work, and to permit the application more precisely to delimit
396 transactions.

397 In XDBC, the basic model is that each connection to a data source is a separate transaction.
398 The ability for a transaction to span data sources is implementation-defined. For
399 implementations that let a transaction span data sources, the X/Open **DTP** specifications
400 help show how to delimit and identify units of work by global transaction identification.

401 • **Stored procedures**

402 Stored procedures are database routines that reside at the server. The application invokes
403 such a procedure by name. In a client/server architecture, use of procedures may enhance
404 performance by minimising traffic between client and server.

405 The application can use XDBC to describe the parameters to a stored procedure and to query
406 the metadata and determine the procedures that are present in a database and the parameters
407 that pertain to a specific procedure.

1.2 This Issue

This section describes the major differences between this issue and the predecessor document, the **X/Open Call Level Interface (CLI)** CAE Specification (March 1995).

Alignment with Popular Implementations

The marketplace accepted the March 1995 issue as a basis for the basic XDBC features. Both X/Open and software vendors continued development of advanced features, some of which are listed below. In many areas, vendor developments outpaced work both in X/Open and standards organisations.

In 1996, X/Open elected to align its publication with interfaces gaining acceptance in the marketplace, subject to the usual process of review and consensus. One effect of this approach is a complete replacement of the reference manual pages of the March 1995 issue. In some cases, the text is totally different even though it specifies essentially the same syntax and semantics.

Each reference manual page contains a **CHANGE HISTORY** indicating whether there is an analogous manual page in the March 1995 issue. However, these histories do not try to compare the lexical changes. The syntax and semantics of functions that existed in the March 1995 issue has not changed even though many of the descriptions have changed. Enhancements are implemented in backward-compatible ways, such as additional legal values of some arguments. This level of detail is generally not addressed by the **CHANGE HISTORY** sections.

The March 1995 issue was not the subject of any X/Open branding programme or software testing.

New Features

Alignment with popular implementations has had the effect of adding the following areas of specification to this issue:

- **Bookmarks**

  Bookmarks mark a position in a result set. Bookmarks can be of fixed or variable length. The application can use *SQLFetch*() and *SQLFetchScroll*() to fetch by bookmark. Update, delete, and re-fetch operations, of one row or many at a time, using bookmarks on discontiguous rows are supported.

- **Binding to an array of parameters**

  The *SQLBindParameter*() function can be called with the address of an array of data pointers, rather than a single data pointer.

- **Quick rebinding by offset**

  An application can specify that an offset be added to buffer addresses specified for row data or dynamic parameters. This lets an application change column and parameter bindings without extra function calls. When new addresses always occur at a fixed offset from old addresses, this enables more efficient processing.

- **Batch**

  XDBC lets the application query how implementation reports the results of a batch of SQL statements. A batch can result either from execution of a stored procedure or of a sequence of statements executed in a single call to *SQLExecDirect*() or *SQLExecute*().

- **Positioned UPDATE and DELETE via function call**

  The new *SQLSetPos*() function permits positioned UPDATE and DELETE operations. These operations were achieved in the March 1995 issue exclusively by executing the UPDATE or

451 DELETE statements of embedded SQL.

452 • **Additional catalog functions**

453 Additional functions for querying the metadata appear in this issue. They are
454 *SQLColumnPrivileges*(), *SQLForeignKey*(), *SQLPrimaryKey*(), *SQLProcedureColumns*(),
455 *SQLProcedures*(), and *SQLTablePrivileges*().

456 • **Escape clauses**

457 To allow a standard method of coding in cases where implementations of embedded SQL
458 language vary, XDBC provides escape clauses for outer joins, scalar functions, date/time and
459 interval literals, and stored procedures. The XDBC implementation translates the escape
460 clause to the dialect the data source accepts.

461 • **On-demand descriptor population**

462 The March 1995 issue contained an optional feature that provided that, on implementations
463 that had capabilities analogous to the DESCRIBE INPUT statement of embedded SQL,
464 implementation parameter descriptors could be populated. In the current issue, this
465 population occurs only on demand. The application requests this behaviour by setting a
466 statement attribute.

467 • **Enhanced diagnostics**

468 Parameter status arrays are included in this issue. In addition, after a multi-row fetch,
469 diagnostic information is available that indicates the status of each row fetched. The
470 application can also determine the column number to which any diagnostic information
471 applies.

472 • **New data types**

473 This issue includes interval buffer types, integer application buffer types with specific bit
474 lengths up to 64 bits, binary buffer types, signed and unsigned integer buffer types, and
475 buffer types for NUMERIC, DECIMAL, DATE, TIME, and TIMESTAMP data.

476 • **Connection enhancements**

477 The new *SQLBrowseConnect*() function gives the application an iterative method of
478 determining the capabilities of the available data sources in order to choose a suitable data
479 source to which to connect using *SQLConnect*().

480 The new *SQLDriverConnect*() function is added as an alternative to *SQLConnect*().
481 *SQLDriverConnect*() supports data sources that require more connection information than the
482 three arguments of *SQLConnect*(). *SQLDriverConnect*() also provides that the implementation
483 interacts with the user to obtain any connection information that the caller fails to specify.

484 OP • **Asynchrony**

485 An optional asynchronous calling mode lets XDBC functions return before the requested
486 operation has completed. The application can perform other operations concurrently, can
487 determine when the requested operation has completed, and can obtain the status of that
488 operation.

489 OP • **Multi-row fetch**

490 An optional multi-row fetch feature lets individual calls to *SQLFetch*() and *SQLFetchScroll*()
491 return *row-sets* consisting of more than one row.

492 In a multi-row fetch, the deferred fields are redefined as pointers to arrays, so that they can
493 be bound to the column data of an entire row-set. New data structures are defined to
494 indicate diagnostic events that pertain to the multi-row fetch at large and to specific rows.

495 The diagnostics sequencing rules are extended to cover the case of a multi-row fetch.

496 • **Row-wise binding**

497 When retrieving multiple rows at a time, each column can be bound to an array of column
498 buffers (''column-wise binding''). Alternatively, each column can be bound to a member of a
499 structure, and an array of these structures represents the multiple rows (''row-wise
500 binding''). (In the case of single-row fetch, these methods are equivalent.) Row-wise binding
501 better matches the way applications tend to want to deal with data from result sets once it is
502 fetched. Row-wise binding also applies to dynamic parameters (and is important in the case
503 of arrays of parameters) because of the symmetry of the descriptor model.

504 • **Support for stored routines**

505 The new *SQLProcedureColumns*() and *SQLProcedures*() functions perform metadata queries
506 relating to stored procedures.

507 For implementations in which stored routines can be registered with the capability to return |
508 *ad hoc* result sets, this specification envisages that all such result sets are returned on a single
509 statement handle and can be processed serially. The *SQLMoreResults*() function determines
510 whether any more result sets exist on a statement handle and move to the next result set.

511 • **New items in SQLGetInfo()**

512 The information available through *SQLGetInfo*() has been expanded, with the intent of
513 enabling the application to use *SQLGetInfo*() to determine the status of the implementation's
514 support for most features that this specification designates as implementation-defined.

515 This facility lets an application determine an implementation's ISO SQL compliance level and |
516 degree of support for SQL. The new information items are listed in **Changes to Information**
517 **Items in SQLGetInfo()** on page 405.

518 • **SQLNativeSql()**

519 The new *SQLNativeSql*() function returns a specified dynamic SQL statement as modified by
520 a specified implementation, without actually executing the statement. It is this statement |
521 that the XDBC implementation would send to the data source if the application requested the |
522 execution of the specified statement.

523 • **SQLNumParams()**

524 In the March 1995 issue, applications could determine the number of parameters that an SQL
525 statement contains by obtaining the SQL_DESC_COUNT field of the implementation
526 parameter descriptor. In the current issue, *SQLNumParams*() is a new, concise function that
527 achieves the same result without requiring a descriptor handle.

528 Lists of certain new values for data structures appear in the **CHANGE HISTORY** section of the
529 relevant *Set* function:

530 • **New Connection Attributes in Version 2** on page 461
531 • **Descriptor Fields Added in Version 2** on page 483 |
532 • **New Statement Attributes in Version 2** on page 515

Dropped Features

This issue drops the following features that were present in the March 1995 issue:

- **Use of embedded SQL as basis**

  The March 1995 issue envisaged that one possible implementation of the API was to base it on an X/Open-compliant embedded SQL implementation. A small number of deviations from this rule was enumerated and marked with the EX margin legend. The current issue does not retain this assumption and does not flag aspects of XDBC that do not map to the X/Open **SQL** specification.

- **COBOL bindings**

  This issue specifies a set of C functions. It does not preclude bindings to other languages. The function synopses are given in C language; a method of translating the synopses to other languages is outside the scope of this specification. The code examples are now exclusively in C.

  Text in the March 1995 that accommodated programming languages that do not provide pointer capabilities does not appear in this issue.

- **Call-by-reference**

  The March 1995 issue envisaged two sets of functions: a call-by-value variant, with the prefix *SQL*, and a call-by-reference variant, with the prefix *SQLR*. Call-by-reference would be the variant used in languages such as COBOL.

  The current issue specifies only the call-by-value variant.

  The March 1995 issue described the API in a manner that did not specify the variant. For example, there was a reference manual page for *Fetch*(). In the current issue, the corresponding page appears under *SQLFetch*() and specifies the call-by-value variant.

- **SQLBindParam()**

  The function *BindParam*() was specified in the March 1995 issue. The current issue specifies *SQLBindParameter*(), which subsumes all the material formerly in *BindParam*(). It also allows for input, output, and input/output parameters.

- **SQL_ATTR_METADATA_ID connection attribute**

  In the March 1995 issue, SQL_ATTR_METADATA_ID was defined as both a connection attribute and a statement attribute. In the current issue, it appears only as a statement attribute. However, this issue lets the application set any statement attributes on a connection handle, in order to specify a default value for all statement handles allocated on the connection handle. This rule covers the behaviour of SQL_ATTR_METADATA_ID as it was formerly specified as a connection attribute.

- **Formerly deprecated functions now removed**

  The following functions were labelled deprecated in the March 1995 issue, with a warning that applications should convert to the preferred functions and that X/Open would delete the functions from a future issue. These have now been deleted:

  - *AllocConnect*() — Use *SQLAllocHandle*() with SQL_HANDLE_DBC as *HandleType*.
  - *AllocEnv*() — Use *SQLAllocHandle*() with SQL_HANDLE_ENV as *HandleType*.
  - *AllocStmt*() — Use *SQLAllocHandle*() with SQL_HANDLE_STMT as *HandleType*.
  - *ColAttributes*() — Use *SQLColAttribute*().
  - *Error*() — Use *SQLGetDiagField*() or *SQLGetDiagRec*().
  - *FreeConnect*() — Use *SQLFreeHandle*() with SQL_HANDLE_DBC as *HandleType*.
  - *FreeEnv*() — Use *SQLFreeHandle*() with SQL_HANDLE_ENV as *HandleType*.

578 • *GetConnectOption*() — Use *SQLGetConnectAttr*().
579 • *GetStmtOption*() — Use *SQLGetStmtAttr*().
580 • *SetConnectOption*() — Use *SQLSetConnectAttr*().
581 • *SetParam*() — Use *BindParameter*().
582 • *SetStmtOption*() — Use *SQLSetStmtAttr*().
583 • *Transact*() — Use *SQLEndTran*().

584 Additional details for converting to non-deprecated methods appear in Chapter 8 of the
585 March 1995 issue.

586 • **Formerly deprecated function now undeprecated**

587 The *FreeStmt*() function was deprecated in the March 1995 issue. The SQL_DROP option is
588 deleted from this issue. The other options are documented in this issue, and are no longer
589 deprecated, because there was no equivalent work-around.

590 The *RowCount*() function was deprecated in the March 1995 issue. It is no longer deprecated. |
591 It provides behaviour not available elsewhere — a count of rows with a longer persistence |
592 than the count in the diagnostics area.

1.3 Relation to Other X/Open Documents

1.3.1 Conceptual Differences from Embedded SQL

XDBC introduces a new style of application program binding for SQL that contains elements of X/Open embedded SQL and of direct invocation as defined in the referenced ISO SQL standard. However, XDBC is conceptually different from prior SQL implementations in the following ways:

- **Execution model**

  XDBC introduces a new model for the execution of any SQL statement that is preparable in dynamic SQL. XDBC does not require explicit declaration of cursors, nor does it require a different SQL verb (OPEN as opposed to EXECUTE) depending on the SQL text.

- **Cursor**

  The XDBC cursor model is a mixture of the current dynamic and direct invocation binding styles. Executing a *cursor-specification*[2] can return multiple rows even though the application does not explicitly declare a cursor. The application can also use the normal cursor fetch model on such *cursor-specifications*; it can also use positioned UPDATE and DELETE statements. This follows from the rule that any preparable SQL statement can be executed using XDBC.

- **Statement handles**

  A statement handle is a variable that refers to an implementation-defined data structure used to contain all information related to an SQL statement. The statement handle corresponds roughly to the diagnostics area and SQLSTATE of embedded SQL (see Section 4.1 on page 34).

- **Environment, connection and descriptor handles**

  These other handles take the place of all remaining global variables, of connection-specific state, and of SQL descriptor areas in embedded SQL (see Section 4.1 on page 34).

- **Automatic sizing of data structures**

  For data structures with a variable number of records, such as a diagnostics area or an SQL descriptor, the XDBC implementation takes any necessary action to accommodate however many records are written to the data structure. The application does not have to declare a number of records when it allocates the data structure.

- **Automatic data conversion**

  In XDBC, the application can specify the host-language buffer format for dynamic parameters and column data. If this differs from the format used for communication with the server, the client automatically converts data when it sends dynamic arguments to the database and when it fetches columns from the database.

---

2. Throughout this specification, *cursor-specification* refers to the entire syntax of the *cursor-specification* (SELECT statement) defined in the X/Open **SQL** specification. This does not include the SELECT...INTO syntax of the dynamic FETCH statement.

1.4 Relation to Standards

X/Open's goal is that implementations be able to comply both to XDBC and to the ISO CLI International Standard, and that application writers have clear guidelines for writing applications with maximum portability.

XDBC includes many features not yet included in the ISO CLI International Standard.[3] The XDBC functions that are not yet included in the ISO CLI International Standard are

- The catalog functions *SQLColumnPrivileges*(), *SQLColumns*(), *SQLForeignKey*(), *SQLModules*(), *SQLPrimaryKey*(), *SQLProcedureColumns*(), *SQLProcedures*(), *SQLSpecialColumns*(), *SQLStatistics*(), *SQLTablePrivileges*(), and *SQLTables*().
- The non-catalog functions *SQLBulkOperations*(), *SQLDescribeParam*(), and *SQLSetPos*().

The ISO CLI International Standard includes some features that XDBC does not:

- The ISO CLI International Standard takes advantage of some data types defined in advanced levels of the ISO SQL standard, while XDBC does not. These data types are BIT, BIT VARYING, NATIONAL CHARACTER, NATIONAL CHARACTER VARYING, TIME WITH TIMEZONE, and TIMESTAMP WITH TIMEZONE. The SQL_BIT data type defined in this specification is not the same as the BIT data type of the ISO CLI International Standard.
- The ISO CLI International Standard contains functions that appear in XDBC but are marked deprecated.
- The ISO CLI International Standard contains descriptor fields that specify a character set and a collation, while XDBC does not.
- XDBC does not have descriptor fields relating to stored routine parameters, as described in the emerging ISO PSM standard.

Where both XDBC and the ISO CLI International Standard define the same feature, X/Open intends that the XDBC definition permit an implementation that also complies to the ISO CLI International Standard. When this is not the case (for instance, in cases of oversights or editorial errors), X/Open intends to issue a statement explicitly deferring to the ISO CLI International Standard, so that it is the authority by which any discrepancies are resolved.

---

3. Most of these are listed in **New Features** on page 2. That list compares this specification to the March 1995 issue, which was similar in features to the ISO CLI International Standard.

1.5 Compliance Policy

1.5.1 Language Binding

This document describes a set of XDBC functions that are callable from C. Some XDBC products support additional languages.

The goal of this X/Open specification is the ability to write portable programs. Compliance to this specification means that the XDBC implementation must include bindings to an X/Open-compliant C implementation. It is implementation-defined whether the implementation provides bindings to other languages. X/Open intends to publish, for each XDBC product it brands, the extent to which purchasers can use the product to write portable applications.

1.5.2 SQL Statement Text

This specification gives applications a way to provide SQL statement text for execution. The SQL statement text is typically a statement from the database language specified in the X/Open **SQL** specification. Compliance with this document is separate from, and does not presume, compliance with the X/Open **SQL** specification. However, X/Open recommends that implementations comply with both this specification and the X/Open **SQL** specification.

On an implementation that complies with both this specification and the X/Open **SQL** specification, valid SQL statement text for execution using XDBC is defined as any SQL statement that can be prepared in dynamic SQL, as specified by the X/Open **SQL** specification: ALTER, CREATE, *cursor-specification*, searched DELETE, positioned DELETE, DROP, GRANT, INSERT, REVOKE, searched UPDATE, positioned UPDATE and the vendor escape clause. The COMMIT and ROLLBACK statements of dynamic SQL are specifically excluded from execution using XDBC, as this specification provides other methods of transaction delimitation (see Chapter 14).

In addition, any dynamic arguments must appear so that their data type can be deduced, and prefixes, terminators, comments and embedded variable names are prohibited. Refer, in the X/Open **SQL** specification, to the explanation of the '**42**000' diagnostic for the PREPARE statement.

Other SQL Dialects for XDBC Testing

Testing an XDBC implementation involves, among other things, submitting requests to modify a database, requesting the revised contents of the database, and verifying that the contents seem to have been modified correctly. One method of doing this is to submit SQL statement text. This method must be tested, and doing so requires an assumption about what SQL grammar is available for use. There are model SQL dialects less complete than the language the X/Open **SQL** specification defines, but complete enough to enable XDBC testing while avoiding the features of SQL whose implementation varies. If the implementation does not accept the SQL grammar defined in the X/Open **SQL** specification, it must satisfy any assumptions the testing software makes about what set of SQL statements can be submitted.

1.5.3 Distributed Transaction Delimitation

Chapter 14 discusses transactions, which are sequences of database operations with certain collective characteristics such as atomicity. Transaction **delimitation** must exist in order to define the grouping of database operations into transactions.

All transactions must be delimited in exactly one of the following ways, selected based on implementation-defined criteria:

- A transaction begins implicitly when an application operates on a database, as defined in Chapter 14. The transaction ends when the application calls *SQLEndTran*().
- A transaction begins when an application executes the *tx_begin*() function described in the X/Open **TX** specification; and ends when it executes *tx_end*().
- An implementation-defined transaction delimitation interface is used.

The application must mark the end of a transaction using the same technique it used to mark the start of a transaction.

X/Open-compliant XDBC implementations that also comply with the support the second option for all transactions. The *tx_begin*() and *tx_end*() functions call a transaction manager, which coordinates completion of SQL and non-SQL work to provide global atomicity. The XDBC implementation supports the X/Open XA interface, in the role of a Resource Manager (RM). The XA interface lets the transaction manager inform the XDBC implementation of the delimitation and disposition of transactions.

715 ## 1.6 Compliance Terminology

716 The following compliance terms convey the same meanings as defined in the X/Open **SQL**
717 specification.

718 **Optional features**

719 An optional feature serves as guidance to implementors on the preferred syntax for a
720 feature that is not yet widespread. X/Open does not currently enforce the implementation •
721 of optional features, but intends to make them mandatory in future issues of the XPG, in the
722 manner in which they are specified in this edition. At that time, implementations will be
723 required to provide the feature.

724 X/Open may test implementations to see if they implement optional features as specified in
725 this document. X/Open would make the results available to prospective purchasers.

726 Application writers may use optional features that are known to be available on the
727 implementation in use, at the risk of reduced portability.

728 Discussions of optional features are shaded with the OP margin notation, as shown below.

729 OP The following features are optional:

730 — The ability to have the implementation describe dynamic parameters in prepared
731 statements.

732 — The scalar functions specified in Appendix F. |

733 — The architecture, specified in Appendix I, in which an implementation is divided into a |
734 Driver Manager and various drivers. |

735 — An implementation need not provide all data types defined in Section D.1 on page 556. |
736 An application calls *SQLGetTypeInfo*() to discover which data types are supported.

737 **Deprecated features**

738 Deprecated features include syntax that X/Open views as obsolete or non-optimal.
739 Implementors must provide features labelled deprecated, in the interest of backward-
740 compatibility. Application writers using deprecated features are advised that X/Open
741 intends to remove them from future issues of this specification.

742 Each deprecated feature lists a preferred method of performing the same function.
743 X/Open's policy on deprecated features is to maintain the deprecated designation for at
744 least one issue of the XPG. This gives application writers adequate notice to change their
745 coding to the recommended method. When X/Open reissues the XPG with a feature
746 omitted, implementations may remove support for the feature.

747 Deprecated features in XDBC are as follows:

748 • The *BindParam*() function binds a parameter in an SQL statement to an application
749 variable. Applications should now use *SQLBindParameters*(), which also supports
750 output and input/output parameters.

751 • The SQL_FETCH_DIRECTION and SQL_SCROLL_CONCURRENCY values of *InfoItem*
752 in calls to *SQLGetInfo*() determine details of the implementation of cursors. A new
753 technique using bitmasks gives the application much more information, regarding the
754 implementation of each of four types of cursors, in a symmetric manner. The new
755 technique is discussed in **Detecting Cursor Capabilities with SQLGetInfo()** on page
756 402.

757 **Compliance**

758 An XDBC implementation is X/Open-compliant if it supports all the assertions this
759 document makes that are not labelled optional. The implementation may also support the

760 features labelled optional, or other features not specifically identified in this document.
761 Implementors are free not to implement features marked optional, but if they implement
762 such a feature, they must do so as specified in this document.

763 An application program is X/Open-compliant if it uses only the syntax contained in this
764 document that is not labelled optional.

765 **Implementation-defined**
766 Implementation-defined means that the resolution of the issue in question may vary
767 between implementations, and that each X/Open-compliant implementation must publish
768 information on how it resolves that issue.

769 **Undefined**
770 Undefined means that the resolution of the issue in question may vary between
771 implementations, and that an X/Open-compliant implementation need not publish
772 information on how it resolves that issue.

773 Footnotes are used as a technique to improve readability of the main text. However, information
774 in footnotes is as much a part of this specification as information in the main text.

1.7 XDBC Compliance Levels

An XDBC implementation may give the application access to diverse data sources. The implementation lets the application determine at run time what XDBC capabilities the implementation and each data source supports.

To simplify specification of compliance, XDBC defines three levels. Compliance with a given level implies complete compliance with all lower levels. Compliance levels do not always divide neatly into support for a specific list of XDBC functions, but specify supported features[4] as listed in the following sections. To provide support for a feature, an implementation must support some or all forms of calls to certain XDBC functions (see also **Function Cross-reference** on page 17), setting certain attributes (see also **Attribute Cross-reference** on page 18), and certain descriptor fields (see also **Descriptor Field Cross-reference** on page 19).

The application determines the XDBC compliance level after connecting to a data source by calling *SQLGetInfo*() with the SQL_XDBC_INTERFACE_CONFORMANCE option.

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

Core-level XDBC Compliance

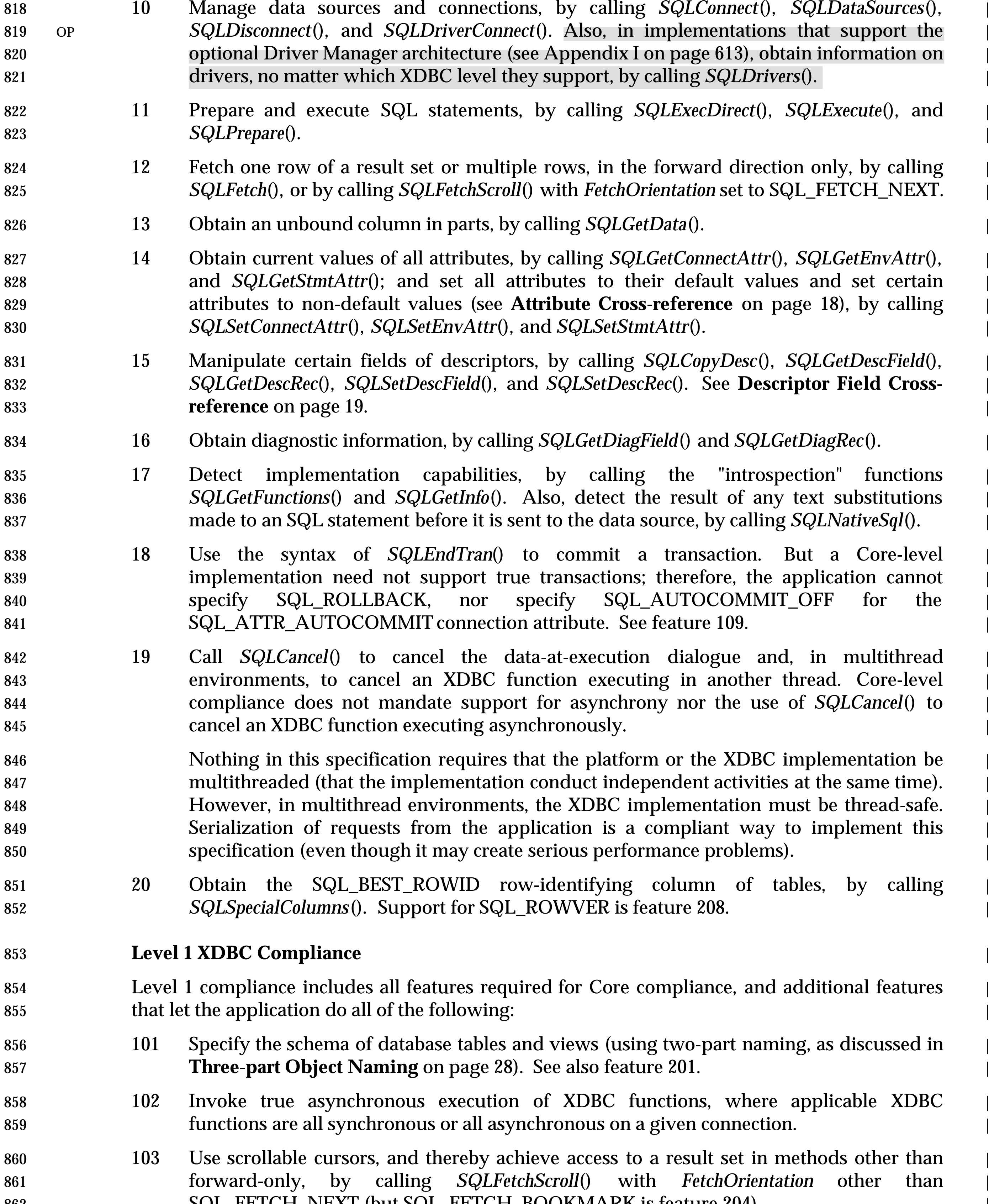
All XDBC implementations exhibit at least Core-level compliance. This lets the implementation work with most applications and corresponds to the non-optional features defined in the March 1995 issue of the X/Open **CLI** specification. A Core-level-compliant XDBC implementation lets the application do all of the following:

1. Allocate and free all types of handle, by calling *SQLAllocHandle*() and *SQLFreeHandle*().
2. Use all forms of the *SQLFreeStmt*() function.
3. Bind result set columns, by calling *SQLBindCol*().
4. 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.)
5. Specify a bind offset.
6. Use the data-at-execution dialogue, involving calls to *SQLParamData*() and *SQLPutData*().
7. Manage cursors and cursor names, by calling *SQLCloseCursor*(), *SQLGetCursorName*(), and *SQLSetCursorName*().
8. 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.)
9. Query the data dictionary, by calling the catalog functions *SQLColumns*(), *SQLGetTypeInfo*(), *SQLStatistics*(), and *SQLTables*(). (The implementation is not required to support multi-part names of database tables and views.[5] See also features 101 and 201.)

---

4. The features listed in the following sections are numbered for ease of reference. The numbers are not official and the sequencing of the list and manner of grouping the material into discrete features is not relevant to compliance.

5. However, certain features of the X/Open **SQL** specification, such as column qualification and names of indexes, are syntactically comparable to multi-part naming. The present list of XDBC features is not intended to introduce new optionality into these aspects of the X/Open **SQL** specification.



10 Manage data sources and connections, by calling *SQLConnect*(), *SQLDataSources*(), *SQLDisconnect*(), and *SQLDriverConnect*(). Also, in implementations that support the optional Driver Manager architecture (see Appendix I on page 613), obtain information on drivers, no matter which XDBC level they support, by calling *SQLDrivers*().

11 Prepare and execute SQL statements, by calling *SQLExecDirect*(), *SQLExecute*(), and *SQLPrepare*().

12 Fetch one row of a result set or multiple rows, in the forward direction only, by calling *SQLFetch*(), or by calling *SQLFetchScroll*() with *FetchOrientation* set to SQL_FETCH_NEXT.

13 Obtain an unbound column in parts, by calling *SQLGetData*().

14 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 (see **Attribute Cross-reference** on page 18), by calling *SQLSetConnectAttr*(), *SQLSetEnvAttr*(), and *SQLSetStmtAttr*().

15 Manipulate certain fields of descriptors, by calling *SQLCopyDesc*(), *SQLGetDescField*(), *SQLGetDescRec*(), *SQLSetDescField*(), and *SQLSetDescRec*(). See **Descriptor Field Cross-reference** on page 19.

16 Obtain diagnostic information, by calling *SQLGetDiagField*() and *SQLGetDiagRec*().

17 Detect implementation capabilities, by calling the "introspection" functions *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*().

18 Use the syntax of *SQLEndTran*() to commit a transaction. But a Core-level implementation need not support true transactions; therefore, the application cannot specify SQL_ROLLBACK, nor specify SQL_AUTOCOMMIT_OFF for the SQL_ATTR_AUTOCOMMIT connection attribute. See feature 109.

19 Call *SQLCancel*() to cancel the data-at-execution dialogue and, in multithread environments, to cancel an XDBC function executing in another thread. Core-level compliance does not mandate support for asynchrony nor the use of *SQLCancel*() to cancel an XDBC function executing asynchronously.

Nothing in this specification requires that the platform or the XDBC implementation be multithreaded (that the implementation conduct independent activities at the same time). However, in multithread environments, the XDBC implementation must be thread-safe. Serialization of requests from the application is a compliant way to implement this specification (even though it may create serious performance problems).

20 Obtain the SQL_BEST_ROWID row-identifying column of tables, by calling *SQLSpecialColumns*(). Support for SQL_ROWVER is feature 208.

Level 1 XDBC Compliance

Level 1 compliance includes all features required for Core compliance, and additional features that let the application do all of the following:

101 Specify the schema of database tables and views (using two-part naming, as discussed in **Three-part Object Naming** on page 28). See also feature 201.

102 Invoke true asynchronous execution of XDBC functions, where applicable XDBC 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 *SQLFetchScroll*() with *FetchOrientation* other than SQL_FETCH_NEXT (but SQL_FETCH_BOOKMARK is feature 204).

863 104 Obtain primary keys of tables, by calling *SQLPrimaryKeys*().

864–867 105 Use stored procedures, through the XDBC escape clause for procedure calls; and query the data dictionary regarding stored procedures, by calling *SQLProcedureColumns*() and *SQLProcedures*(). (The process by which procedures are created and stored on the data source is outside the scope of this specification.)

868–869 106 Connect to a data source by interactively browsing the available servers, by calling *SQLBrowseConnect*().

870–872 107 Use XDBC functions instead of SQL statements to perform certain database operations: *SQLBulkOperations*() with SQL_ADD and *SQLSetPos*() with SQL_POSITION and SQL_REFRESH.

873–874 108 Gain access to the contents of multiple result sets generated by batches and stored procedures, by calling *SQLMoreResults*().

875–876 109 Delimit transactions spanning several XDBC functions, with true atomicity and the ability to specify SQL_ROLLBACK in *SQLEndTran*().

877 **Level 2 XDBC Compliance**

878–879 Level 2 compliance includes all features required for Core and Level 1 compliance, plus additional features that let the application do all of the following:

880–881 201 Use three-part names of database tables and views (see **Three-part Object Naming** on page 28). See also feature 101.

882 202 Describe dynamic parameters, by calling *SQLDescribeParam*().

883–884 203 Use not only input parameters but output and input/output parameters, and result values of stored procedures.

885–889 204 Use bookmarks: Retrieve bookmarks by calling *SQLDescribeCol*() and *SQLColAttribute*() on column number 0; fetch based on a bookmark by calling *SQLFetchScroll*() with *FetchOrientation* set to SQL_FETCH_BOOKMARK; and call *SQLBulkOperations*() with SQL_UPDATE_BY_BOOKMARK, SQL_DELETE_BY_BOOKMARK, and SQL_FETCH_BY_BOOKMARK.

890–891 205 Retrieve advanced information on the data dictionary, by calling *SQLColumnPrivileges*(), *SQLForeignKeys*(), and *SQLTablePrivileges*().

892–895 206 Use XDBC functions instead of SQL statements to perform additional database operations, by calling *SQLSetPos*() with SQL_DELETE, SQL_UPDATE. Includes support for calls to *SQLSetPos*() with *LockType* set to SQL_LOCK_EXCLUSIVE and SQL_LOCK_UNLOCK.

896 207 Enable asynchronous execution of XDBC functions for specified individual statements.

897–898 208 Obtain the SQL_ROWVER row-identifying column of tables, by calling *SQLSpecialColumns*(). See also feature 20.

899–900 209 Set the SQL_ATTR_CONCURRENCY statement attribute to at least one value other than SQL_CONCUR_READ_ONLY.

901–902 210 Set the SQL_ATTR_OUTPUT_NTS to SQL_FALSE to disable null-termination of output character strings.

903 211 Execute transactions with the "serializable" level of isolation.

904 **Optional at All Levels**

905 Features specified as optional are not required to be supported regardless of the
906 implementation's compliance level. See Section 1.6 on page 11 for a list of optional features.

907 **Function Cross-reference**

908 The following table indicates the compliance level of each XDBC function, where this is well-
909 defined.

910 *SQLAllocHandle*(): Core
911 *SQLBindCol*(): Core
912 *SQLBindParam*() DE : Core *
913 *SQLBindParameter*(): Core *
914 *SQLBrowseConnect*(): Level 1
915 *SQLBulkOperations*(): Level 1 *
916 *SQLCancel*(): Core *
917 *SQLCloseCursor*(): Core
918 *SQLColAttribute*(): Core *
919 *SQLColumnPrivileges*(): Level 2
920 *SQLColumns*(): Core
921 *SQLConnect*(): Core
922 *SQLCopyDesc*(): Core
923 *SQLDataSources*(): Core
924 *SQLDescribeCol*(): Core *
925 *SQLDescribeParam*(): Level 2
926 *SQLDisconnect*(): Core
927 *SQLDriverConnect*(): Core
928 *SQLDrivers*() OP : Core
929 *SQLEndTran*(): Core *
930 *SQLExecDirect*(): Core
931 *SQLExecute*(): Core
932 *SQLFetch*(): Core
933 *SQLFetchScroll*(): Core *
934 *SQLForeignKeys*(): Level 2
935 *SQLFreeHandle*(): Core
936 *SQLFreeStmt*(): Core
937 *SQLGetConnectAttr*(): Core
938 *SQLGetCursorName*(): Core
939 *SQLGetData*(): Core
940 *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 **
SQLSetCursorName(): Core
SQLSetDescField(): Core
SQLSetDescRec(): Core
SQLSetEnvAttr(): Core **
SQLSetPos(): Level 1 *
SQLSetStmtAttr(): Core **
SQLSpecialColumns(): Core *
SQLStatistics(): Core
SQLTablePrivileges(): Level 2
SQLTables(): Core

941 ____________________
942 * But significant features of this function are available only at higher compliance levels.
943 ** Setting certain attributes to non-default values depends on the compliance level; see **Attribute Cross-reference** on page 18.

Attribute Cross-reference

All XDBC implementations let applications obtain the current value of any attribute by calling *SQLGetConnAttr*(), *SQLGetEnvAttr*(), or *SQLGetStmtAttr*(). All XDBC implementations allow calls to *SQLSetConnAttr*(), *SQLSetEnvAttr*(), or *SQLSetStmtAttr*() that simply reassert the default value of the attribute. The ability to set an attribute to a non-default value depends on the compliance level, as follows:

Connection attributes

SQL_ATTR_ACCESS_MODE: Core
SQL_ATTR_ASYNC_ENABLE: *
SQL_ATTR_AUTO_IPD: Level 2
SQL_ATTR_AUTOCOMMIT: Level 1
SQL_ATTR_CONNECTION_TIMEOUT: Level 2
SQL_ATTR_CURRENT_CATALOG: Level 2
SQL_ATTR_LOGIN_TIMEOUT: Level 2
SQL_ATTR_PACKET_SIZE: Level 2
SQL_ATTR_QUIET_MODE: Core
SQL_ATTR_TXN_ISOLATION: [6]

Environment attribute

SQL_ATTR_OUTPUT_NTS: Level 2

Statement attributes

SQL_ATTR_APP_PARAM_DESC: Core
SQL_ATTR_APP_ROW_DESC: Core
SQL_ATTR_ASYNC_ENABLE: *
SQL_ATTR_CONCURRENCY: [7]
SQL_ATTR_CURSOR_TYPE: [8]
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

---

6. For Level 1 compliance, the implementation must support one value in addition to the implementation-defined default value (available by calling *SQLGetInfo*() with the SQL_DEFAULT_TXN_ISOLATION option). For Level 2 compliance, the implementation must also support SQL_TXN_SERIALIZABLE.

* Applications that support connection-level asynchrony (required for Level 1) must support setting this statement 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.

7. For Level 2 compliance, the implementation must support SQL_CONCUR_READ_ONLY and at least one other value.

8. For Level 1 compliance, the implementation must support SQL_CURSOR_FORWARD_ONLY and at least one other value. For Level 2 compliance, the implementation must support all values defined in this specification.

989 SQL_ATTR_PARAMS_PROCESSED_PTR: Core
990 SQL_ATTR_PARAMSET_SIZE: Core
991 SQL_ATTR_QUERY_TIMEOUT: Level 1
992 SQL_ATTR_RETRIEVE_DATA: Level 1
993 SQL_ATTR_ROW_ARRAY_SIZE: Core
994 SQL_ATTR_ROW_BIND_OFFSET_PTR: Core
995 SQL_ATTR_ROW_BIND_TYPE: Core
996 SQL_ATTR_ROW_NUMBER: Level 1
997 SQL_ATTR_ROW_OPERATION_PTR: Level 1
998 SQL_ATTR_ROW_STATUS_PTR: Core
999 SQL_ATTR_ROWS_FETCHED_PTR: Core
1000 SQL_ATTR_SIMULATE_CURSOR: Level 2
1001 SQL_ATTR_USE_BOOKMARKS: Level 2

1002 **Descriptor Field Cross-reference**

1003 *Header fields*

1004 SQL_DESC_ALLOC_TYPE: Core
1005 SQL_DESC_ARRAY_SIZE: Core
1006 SQL_DESC_ARRAY_STATUS_PTR: Core (for APD); Level 1 (for ARD).
1007 SQL_DESC_BIND_OFFSET_PTR: Level 1
1008 SQL_DESC_BIND_TYPE: Core
1009 SQL_DESC_COUNT: Core
1010 SQL_DESC_ROWS_PROCESSED_PTR: Core

1011 *Record fields*

1012 SQL_DESC_AUTO_UNIQUE_VALUE: Level 2
1013 SQL_DESC_BASE_COLUMN_NAME: Core
1014 SQL_DESC_BASE_TABLE_NAME: Level 1
1015 SQL_DESC_CASE_SENSITIVE: Core
1016 SQL_DESC_CATALOG_NAME: Level 2
1017 SQL_DESC_CONCISE_TYPE: Core
1018 SQL_DESC_DATA_PTR: Core
1019 SQL_DESC_DATETIME_INTERVAL_CODE: Core *
1020 SQL_DESC_DATETIME_INTERVAL_PRECISION: Core *
1021 SQL_DESC_DISPLAY_SIZE: Core
1022 SQL_DESC_FIXED_PREC_SCALE: Core
1023 SQL_DESC_INDICATOR_PTR: Core
1024 SQL_DESC_LABEL: Level 2
1025 SQL_DESC_LENGTH: Core
1026 SQL_DESC_LITERAL_PREFIX: Core
1027 SQL_DESC_LITERAL_SUFFIX: Core
1028 SQL_DESC_LOCAL_TYPE_NAME: Core
1029 SQL_DESC_NAME: Core
1030 SQL_DESC_NULLABLE: Core
1031 SQL_DESC_OCTET_LENGTH: Core
1032 SQL_DESC_OCTET_LENGTH_PTR: Core
1033 SQL_DESC_PARAMETER_TYPE: [9]

1034 ________________
1035 * Support for these record fields is only required if the implementation supports the applicable data types.

1036 SQL_DESC_PRECISION: Core
1037 SQL_DESC_SCALE: Core
1038 SQL_DESC_SCHEMA_NAME:
1039 SQL_DESC_SEARCHABLE: Core
1040 SQL_DESC_TABLE_NAME: Level 1
1041 SQL_DESC_TYPE: Core
1042 SQL_DESC_TYPE_NAME: Core
1043 SQL_DESC_UNNAMED: Core
1044 SQL_DESC_UNSIGNED: Core
1045 SQL_DESC_UPDATABLE: Core

1046 ___

1047 9. For Core-level compliance, the implementation must support SQL_PARAM_INPUT. For Level 2 compliance, the implementation must also support SQL_PARAM_INPUT_OUTPUT and SQL_PARAM_OUTPUT.

1.8 SQL Registry

X/Open maintains a registry of values associated with the Structured Query Language (database language SQL) and the Call-Level Interface (CLI) for SQL. This registry provides several different categories of values, such as return values of CLI functions and character strings representing different implementations. For each category, the registry may indicate that a specific value, a list of values, or a range of values is reserved for the ISO standard, to X/Open, or to a vendor of a related product.

Implementors and vendors of SQL-related products should consult this registry whenever assigning values for any relevant category. Vendors of SQL or CLI products should request registry of values (or ranges of values) specific to their implementations. (See **Submitting Requests to the Registry** below.)

Obtaining Copies of the Registry

Copies of the X/Open SQL registry (including CLI) are available from X/Open as follows:

- On the World-Wide Web using the following Universal Resource Locators (URL):
  - Plain text in **http://www.xopen.org/infosrv/SQL_Registry/registry.txt**
  - Acrobat PDF in **http://www.xopen.org/infosrv/SQL_Registry/registry.pdf**
  - HTML in **http://www.xopen.org/infosrv/SQL_Registry/registry.htm**
  - PostScript in **http://www.xopen.org/infosrv/SQL_Registry/registry.ps**

  Links to these can be found on the public pages for the X/Open SQL Access Group at **http://www.xopen.org/public/tech/datam/index.htm**

- X/Open offers anonymous access to a File Transfer Protocol (FTP) server. Access to the service is available only over the Internet. Anonymous FTP allows on-line access to a restricted area of filestore, where publically available files are stored. Users can retrieve them interactively.

  The text below describes the anonymous ftp service.

  - From a machine with FTP capabilities and access to the Internet, type:

    ```
    ftp ftp.xopen.co.uk
    ```

  - At the login prompt, enter `anonymous` or `ftp` as your user name.
  - You will then be prompted for a password. Respond by typing your full e-mail address including Internet domain. You will be granted access to any of the files that have been made available for anonymous FTP, but not to other files on the system.
  - Select the SQL Registry by typing:

    ```
    cd pub/SQL_Registry
    ```

  - Retrieve registry information by typing one of the following:

`get registry.pdf`	for the Acrobat PDF version
`get registry.ps`	for the PostScript version
`get registry.txt`	for a plain text version

  There are also password-protected FTP services that disclose information to validated users on a need-to-know basis. Full instructions for use of the FTP server are accessible on the World Wide Web as `http://www.xopen.org/connections/ftpserver`

1088 **Submitting Requests to the Registry**

1089 Before submitting a request, obtain a copy of the registry, as described above, and determine the
1090 table in the registry in which your organisation requires entries.

1091 To register one or more values (or ranges of values), an electronic mail message should be sent to
1092 **sql.registry@xopen.co.uk**. The message should contain the character string SQL REGISTRY
1093 REQUEST in the **From:** field. Failure to include this string, exactly as shown, including the use
1094 of all upper-case letters, may delay the response.

1095 The text of the message must specify:

1096 • The organization on whose behalf the request is being made

1097 • The name of the individual making the request

1098 • The exact title of each table in the Registry into which values should be allocated

1099 • For each category of values, an approximation of the number of values needed.

1100 In most cases, the registrar will assign a limited range of values, similar to the ranges assigned to
1101 other requestors, as shown in the existing registry. Requirements for a large number of values
1102 should please include a justification.

1103 All requests will be answered within a month. The registrar's response will indicate the specific
1104 values or ranges assigned for each category for which a request was made. When the registrar
1105 determines that values and/or ranges have already been assigned to the requesting organization;
1106 the response will point the requestor to the previous requestor from the same organization, thus
1107 avoiding redundancy.

Chapter 3

XDBC Architecture

The XDBC architecture has the following components:

- **Application**

  Performs processing and calls XDBC functions to perform database operations. A typical way of doing this is by using XDBC to submit SQL statements for execution and to retrieve results. Applications are discussed further in Section 3.3 on page 26.

- **XDBC Implementation**

  This specification uses ''XDBC implementation'' to refer to the software that accepts the requests and performs the database operations. The implementation may be spread over physical locations and over discrete software components.

- **Data source**

  The data to which the user wants to gain access, along with its associated operating system and any network platform used to gain access to the data. Also called database, database management system (DBMS), or database engine. Data sources are discussed further in Section 3.4 on page 27.

3.1 XDBC Implementation

It is implementation-defined whether the XDBC implementation comprises multiple components that are visible to the application writer or user. In the simplest case, an application, through the way it is linked or through administrative action, might be permanently associated with a single data source. Typically, however, it is important that the application or the user be able to select the data source from many choices.

For maximum portability — especially in the case where the application is a software product that is sold to and used by many users in many organisations — it is desirable that the application be applicable not just to multiple data sources but to a variety of database architectures.

The SQL language defined in the ISO SQL standard and the X/Open **SQL** specification has achieved some uniformity among data sources, but implementations of SQL still vary; there are still proprietary SQL dialects and vendor-specific enhancements to SQL that are useful.

1137 ## 3.2 Implementation Architecture

1138 OP A technique that lets the XDBC implementation dynamically adapt to diverse data sources,
1139 including data sources yet to be invented, is to divide the implementation into a Driver Manager
1140 and one or more drivers:

1141 - **Driver Manager**

1142 Loads and unloads drivers on behalf of an application. Processes XDBC function calls or
1143 passes them to a driver.

1144 - **Driver**

1145 Processes XDBC function calls, submits SQL requests to a specific data source, and returns
1146 results to the application. If necessary, the driver modifies an application's request so that the
1147 request complies with syntax supported by the associated data source.

1148 It is implementation-defined whether an implementation uses this divided architecture; but if it
1149 does so, it must do so as specified in Appendix I.

3.3 Applications

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

- **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 data sources. Examples include a spreadsheet or statistics package that uses XDBC to import data for further analysis and a word processor that uses XDBC to get a mailing list from a database.

  An important sub-category of generic applications are application development environments. Although the application constructed with these environments will probably work only with a single data source, the environment itself needs to work with multiple data sources.

  What all generic applications have in common is that they are highly interoperable among data sources and they need to use XDBC in a relatively generic manner.

- **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 specific customer, the application works with a single data source.

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

- **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 data source) and create a single report. XDBC is a common interface that saves programmers from having to learn multiple interfaces. Such applications are generally not interoperable and are written to specific data sources.

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

- Select a data source and connect to it.
- Submit an SQL statement for execution.
- Retrieve results (if any).
- Process errors.
- Commit or roll back the transaction enclosing the SQL statement.
- Disconnect from the data source.

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

3.4 Data Sources

A *data source* is any source of the data an application manipulates with XDBC. The data can be in files, in a hierarchy of tables, or any other information a computer can acquire. (The creation of data sources is outside the scope of this specification.)

The purpose of a data source is to gather all of the technical information needed to access the data — the network address, connection technology, and so on — into a single place and conceal any technical details from the user. The user should be able to look at a list that includes Payroll, Inventory, and Personnel, choose Payroll from the list, and have the application connect to the payroll data, without having to know where the payroll data resides or how the application got to it.

A data source can be *file-based*, in which each table (or perhaps each catalog) maps directly to a file in the underlying operating system. The filename or pathname is part of the information required to select the table. Alternatively, a data source can be *SQL-based*, in which case any relationship between the database objects and actual files is not exposed to the application using XDBC.

Data sources are stored on the system with a user-defined name. Associated with the data source name is all of the information the implementation needs to connect to the data source. For a file-based data source, this might include the full path of the directory containing the relevant files and options that indicate how to use those files (such as single-user mode or read-only). For an SQL-based data source, this might include a specification of the catalog and schema names. (User identification and password are typically not part of the information associated with a data source, but are obtained by interacting with the user.)

Later, the application passes the name of a data source to the implementation, which uses this name to retrieve the necessary data.

Several other terms can be confused with *data source*. In this specification, *database* refers to a database program or engine. A further distinction is made between *desktop databases*, designed to run on personal computers and often lacking in full SQL and transaction support, and *server databases*, designed to run in a client/server situation and characterized by a standalone database engine and rich SQL and transaction support.

Database also refers to a particular collection of data, such as a collection of Xbase files in a directory or a database on SQL Server. It's generally equivalent to the term *catalog*, used elsewhere in this specification.

1225 ## 3.5 Client and Server

1226 The operation of a database system effectively involves two processors, a **client** and a **server**.
1227 The terms client and server in this document always mean SQL client and SQL server. The
1228 *SQLConnect*() function manages the associations between a client and one or more servers.

1229 When an application program is active, it is bound in an implementation-defined manner to a
1230 single client that processes the first implicit or explicit *SQLConnect*() call. The client
1231 communicates with one or more servers, manages connections to servers, maintains the
1232 diagnostics area, and allocates data structures and handles. The server processes other XDBC
1233 functions, including all operations on the database. Following these operations, diagnostic
1234 information is passed (in an undefined way) into the diagnostics area of the client.

1235 **Metadata and Data**

1236 Each server provides a **database**, which consists of **metadata** and **data**.

1237 - Metadata is the definitions of all active base tables, viewed tables, indexes, stored modules,
1238 stored routines, privileges and user names. (An item of metadata is active if it has been
1239 defined and has not subsequently been dropped.)

1240 - Data comprises every value in every active table.

1241 **Schemata**

1242 A **schema** is a collection of related objects. A schema may contain base tables, and contains any
1243 indexes that are defined on the base tables. Namespace issues and ownership of schemata are
1244 discussed in the X/Open **SQL** specification.

1245 **Three-part Object Naming**

1246 A server may support catalogs, in which case every schema resides in a catalog. An application
1247 can uniquely identify a table or index by qualifying the table or index identifier (preceding it
1248 with its catalog and schema name). The use of catalog, schema, and object name is called three-
1249 part naming. Periods separate the catalog, schema, and object name.

1250 It is implementation-defined which of the following object naming systems is supported[10] as
1251 valid syntax for the qualifier:

1252 - *catalog-name.schema-name*

1253 - *schema-name*

1254 On servers that support catalog names, there may be a catalog that does not have a name.

1255 Certain XDBC catalog functions (see Chapter 7) return result sets whose TABLE_CAT column
1256 indicates the catalog name of an object. For objects that do not have a catalog name, this column
1257 contains a zero-length string, except that if the implementation does not support catalog names,
1258 the column may instead be null.

1259 The *catalog-name* and *schema-name* are each syntactically a *user-defined-name*. Other
1260 implementation-defined object naming systems may also be supported. In all cases, use of
1261 qualification is optional for the application program.

1262 ________________

1263 10. An application can determine whether the server supports catalog names by obtaining the
SQL_CATALOG_NAME characteristic from a call to *SQLGetInfo*().

1264 (Some implementations impose other restrictions on qualification; see the X/Open **SQL**
1265 specification.)

1266 ## 3.6 System Information

1267 It is implementation-defined how an XDBC installation stores initialization information,
1268 installation preferences, default values, and any other information required to establish a
1269 connection to a data source based on a data source name. This specification refers to such
1270 information as the *system information*.

1271 Depending on the method used to store the system information, there may be implementation-
1272 defined restrictions on the characters that are valid for use in keywords in connection strings (see
1273 *SQLBrowseConnect*() and *SQLDriverConnect*()). However, alphabetic and numeric characters are
1274 always valid for use in these strings.

1275 ## 3.7 Tables and Views

1276 XDBC functions apply equally to tables and views. The term table is used for both tables and
1277 views, except where the term view is used explicitly.

Chapter 4

Fundamentals

This chapter covers a number of concepts fundamental to writing XDBC applications:

- Handles
- Buffers
- Data types
- Environment, connection, and statement attributes

4.1 Handles

Handles are opaque, 32-bit values that identify a particular item: an environment, connection, statement, or descriptor. When the application calls *SQLAllocHandle*(), the implementation creates a new item of the specified type and returns the handle to it to the application. The application later uses the handle to identify that item when calling XDBC functions. The implementation uses the handle to locate information about the item. The application calls *SQLFreeHandle*() to free a handle.

The following example 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.
SQLUINTEGER 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);
```

For example, suppose the implementation 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 implementation sends the SQL statement to the

1331 data source, which prepares it and returns an access plan identifier. The implementation uses the
1332 handle to find the structure in which to store this identifier.

1333 Later, when the application calls *SQLExecute*() to generate the result set of line numbers for a
1334 particular sales order, it passes the same handle. The implementation again uses the handle, this
1335 time to retrieve the access plan identifier from the structure. It sends the identifier to the data
1336 source to tell it which plan to execute.

1337 4.1.1 Environment Handles

1338 An *environment* is a global context in which to access data; associated with an environment is any
1339 information that is global in nature, such as:

- 1340 The environment's state
- 1341 The current environment-level diagnostics
- 1342 The handles of connections currently allocated on the environment
- 1343 The current settings of each environment attribute

1344 Environment handles are allocated with *SQLAllocHandle*() and freed with *SQLFreeHandle*().
1345 They are always used in calls to *SQLDataSources*() and *SQLDrivers*() and sometimes used in calls
1346 to *SQLEndTran*(), *SQLGetDiagField*(), and *SQLGetDiagRec*().

1347 Some implementations limit the number of active environments they support; the
1348 SQL_ACTIVE_ENVIRONMENTS option in *SQLGetInfo*() specifies how many active
1349 environments are supported.

1350 *Notes to Reviewers*

1351 *This section with side shading will not appear in the final copy. - Ed.*

1352 Active environments and active connections need to be discussed.

1353 4.1.2 Connection Handles

1354 A *connection* comprises a data source and whatever connection technology is required to gain
1355 access to it. A connection handle identifies each connection. The connection handle identifies a
1356 structure that contains connection information, such as:

- 1357 The connection's state
- 1358 The current connection-level diagnostics
- 1359 The handles of statements and descriptors currently allocated on the connection
- 1360 The current settings of each connection attribute

1361 Connection handles are allocated with *SQLAllocHandle*() and freed with *SQLFreeHandle*().
1362 Connection handles are used primarily when connecting to the data source (*SQLConnect*(),
1363 *SQLDriverConnect*(), or *SQLBrowseConnect*()), disconnecting from the data source
1364 (*SQLDisconnect*()), getting information about the connection (*SQLGetInfo*()), retrieving
1365 diagnostics (*SQLGetDiagField*() and *SQLGetDiagRec*()) and performing transactions
1366 (*SQLEndTran*()). They are also used when setting and getting connection attributes
1367 (*SQLSetConnectAttr*() and *SQLGetConnectAttr*()) and when getting the native format of an SQL
1368 statement (*SQLNativeSql*()).

1369 Some implementations limit the number of active connections they support; the
1370 SQL_MAX_DRIVER_CONNECTIONS option in *SQLGetInfo*() specifies how many active
1371 connections are supported.

1372 ### 4.1.3 Statement Handles

1373 A *statement* is most easily thought of as an SQL statement, such as **SELECT * FROM Employee**.
1374 However, a statement is more than just an SQL statement — it consists of all the information
1375 associated with that SQL statement, such as any result sets created by the statement and
1376 parameters used in the execution of the statement. The application does not always define an
1377 SQL statement. For example, a catalog function such as *SQLTables*() conceptually executes a
1378 predefined SQL statement that returns a list of table names.

1379 Each statement is identified by a statement handle. A statement is associated with a single
1380 connection, and there can be multiple statements on that connection. Some implementations
1381 limit the number of active statements they support; the
1382 SQL_MAX_CONCURRENT_ACTIVITIES option in *SQLGetInfo*() specifies how many active
1383 statements are supported on a specified connection. A statement is defined to be active if it has
1384 results pending, where results are either a result set or the count of rows affected by an INSERT,
1385 UPDATE, or DELETE statement, or data is being sent with multiple calls to *SQLPutData*().

1386 The statement handle identifies a structure that contains statement information, such as:

1387 • The statement's state

1388 • The current statement-level diagnostics

1389 • The addresses of the application variables bound to the statement's parameters and result set
1390 columns

1391 • The current settings of each statement attribute.

1392 Statement handles are allocated with *SQLAllocHandle*() and freed with *SQLFreeHandle*().
1393 Statement handles are used in the functions to bind parameters and result set columns
1394 (*SQLBindParameter*() and *SQLBindCol*()), prepare and execute statements (*SQLPrepare*(),
1395 *SQLExecute*(), and *SQLExecDirect*()), retrieve metadata (*SQLColAttribute*() and *SQLDescribeCol*()),
1396 fetch results (*SQLFetch*()), and retrieve diagnostics (*SQLGetDiagField*() and *SQLGetDiagRec*()).
1397 They are also used in catalog functions (*SQLColumns*(), *SQLTables*(), and so on) and a number of
1398 other functions.

1399 ### 4.1.4 Descriptor Handles

1400 Descriptors (see Chapter 13) are data structures that holds information about either column data
1401 or dynamic parameters. Applications use handles to refer to descriptors.

1402 ## 4.2 State Transitions

1403 XDBC defines discrete *states* for each environment, connection, and statement. For example, the
1404 environment can be in the unallocated, allocated, and connected state.

1405 Such an item, as identified by its handle, changes state (makes a *state transition*) when the
1406 application calls a certain function and passes that handle. For example, allocating an
1407 environment handle with *SQLAllocHandle*() changes its state from unallocated to allocated.
1408 XDBC defines legal state transitions, which requires that XDBC functions be called in a certain
1409 sequence. Some functions do not affect an item's state. Some functions affect the state of more
1410 than one item; for example, allocating a connection handle with *SQLAllocHandle*() affects the
1411 state of the connection and also changes the state of the environment to connected.

1412 Some functions cannot be called except when an item is in a specified state. If an application
1413 calls a function out of order, the function returns a *state transition error*, denoted by a SQLSTATE
1414 of HY010 (Function sequence error). For example, if an environment is in the connected state
1415 and the application calls *SQLFreeHandle*() with that environment handle, *SQLFreeHandle*()
1416 returns a state transition error, because it can be called only when the environment is in the
1417 allocated state. By defining this as an invalid state transition, XDBC prevents the application
1418 from freeing the environment while there are active connections.

1419 Some state transitions are intuitively obvious considering the design of XDBC. For example,
1420 *SQLExecute*() executes a prepared statement. If the statement handle passed to it isn't in the
1421 prepared state, *SQLExecute*() returns a state transition error. Well-written applications whose
1422 sequence of calls to XDBC is logical in all cases do not encounter state transition errors.

1423 Some logic errors are not state transition errors. For example, one can't allocate a connection
1424 handle without first allocating an environment handle, because the function that allocates a
1425 connection handle requires an environment handle. Calling *SQLAllocHandle*() in this case cannot
1426 be a state transition error of the environment, because there is no environment. Instead, the
1427 application must have passed *SQLAllocHandle*() an erroneous item as the environment handle.

1428 Logic errors based not just on sequence of function calls but on use of other XDBC data
1429 structures are not included in state transition errors. For example, an XDBC function called in an
1430 inappropriate sequence based on the state of a cursor instead sets SQLSTATE to 24000 (Invalid
1431 cursor state).

1432 This specification tends not to explicitly mention state transitions. Instead, it describes the order
1433 in which functions must be called. For a complete description of states and state transitions, see
1434 Appendix B.

1435 ## 4.3 Buffers

1436 A buffer is any piece of application memory used to pass data between the application and the
1437 implementation. For example, application buffers can be associated with, or *bound to*, result set
1438 columns with *SQLBindCol*(). As each row is fetched, the data is returned for each column in
1439 these buffers. *Input buffers* are used to pass data from the application to the implementation;
1440 *output buffers* are used to return data from the implementation to the application.

1441 This discussion concerns itself primarily with buffers of indeterminate type. The addresses of
1442 these buffers appear as arguments of type SQLPOINTER, such as the *TargetValuePtr* argument in
1443 *SQLBindCol*(). However, some of the items discussed here, such as the arguments used with
1444 buffers, also apply to arguments used to pass strings to the implementation, such as the
1445 *TableName* argument in *SQLTables*().

1446 These buffers generally come in pairs. *Data buffers* are used to pass the data itself, while
1447 *length/indicator buffers* are used to pass the length of the data in the data buffer or a special value
1448 such as SQL_NULL_DATA, which indicates that the data is NULL. The length of the data in a
1449 data buffer is different from the length of the data buffer itself.

1450 A length/indicator buffer is required any time the data buffer contains variable-length data, such
1451 as character or binary data. If the data buffer contains fixed-length data, such as an integer or
1452 date structure, a length/indicator buffer is needed only to pass indicator values because the
1453 length of the data is already known. If an application uses a length/indicator buffer with fixed-
1454 length data, the implementation ignores any lengths passed in it.

1455 The length of both the data buffer and the data it contains is measured in octets as opposed to
1456 characters. For programs that use character sets in which each character occupies a single octet,
1457 lengths in octets and characters are the same. However, applications should be coded to
1458 preserve the distinction in order to be adaptable to other code sets for which there is not a one-
1459 to-one correspondence between octets and characters.

1460 **SQL_IS_POINTER**

1461 The XDBC implementation can determine how to treat values of descriptor fields, diagnostic
1462 fields, and attributes in one of the following ways:

1463 • For fields and attributes defined in XDBC, XDBC specifies the data type.

1464 • If the length buffer contains a value greater than zero, the value is a string.

1465 If neither is true — that is, when gaining access to values of a fixed-length descriptor field,
1466 diagnostic field, or attribute that is not defined by XDBC — the application must inform the
1467 XDBC implementation whether to interpret the contents of the data buffer as an actual value or
1468 as a pointer. The application places one of the following constants in the length buffer:

1469 SQL_IS_POINTER
1470 The data buffer contains a pointer to data whose length is fixed.

1471 SQL_IS_NOT_POINTER
1472 The data buffer contains not a pointer but an actual data value.

1473 These values indicate only whether the data buffer[11] is a pointer or not; in the case of

1474 ________________

1475 11. The data buffer is an argument of the XDBC ''Set'' function that sets the value, and is pointed to by an argument of the XDBC ''Get'' function that retrieves the value. The
1476 argument of the Get function is an output argument and thus a pointer, but this is not what SQL_IS_POINTER refers to.

SQL_IS_POINTER, there is nothing in the XDBC interface to indicate the data type of the thing pointed to.

4.3.1 Deferred Buffers

A *deferred buffer* is one whose value is used not at the time it is specified in a function call but at a later point in time. 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, octet length, and type of the buffer. The implementation saves this information but doesn't examine the contents of the buffer. Later, when the application executes the statement, the implementation retrieves the information and uses it to retrieve the parameter data and send it to the data source. Thus, 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 implementation still expects it to exist; for more information, see Section 4.3.2 on page 39.

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*() *SQLExecDirect*()
Sending data to update or insert a row in a result set	Deferred	*SQLBindCol*()	*SQLSetPos*()
Returning data for output and input/output parameters	Deferred output	*SQLBindParameter*()	*SQLExecute*() *SQLExecDirect*()
Returning result set data	Deferred output	*SQLBindCol*()	*SQLFetch*() *SQLFetchScroll*() *SQLSetPos*()

4.3.2 Allocating and Freeing Buffers

All buffers are allocated and freed by the application. If a buffer isn't 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,
           sizeof(InfoValuePtr), &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 implementation 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;
```

```
// 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, sizeof(ValuePtr),
           &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 implementation:

```
SQLRETURN rc;
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.
}
```

4.3.3 Using Data Buffers

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

1562 ### 4.3.4 Data Buffer Type

1563 The C data type of a buffer is specified by the application. In the case of a single variable, this
1564 occurs when the application allocates the variable. In the case of generic memory — that is,
1565 memory pointed to by a pointer of type void * — this occurs when the application casts the
1566 memory to a particular type. There are two ways in which the implementation discovers this
1567 type:

1568 - **Data buffer type argument**

1569 Buffers used to transfer parameter values and result set data, such as the buffer bound with
1570 *TargetValuePtr* in *SQLBindCol*(), usually have an associated type argument, such as the
1571 *TargetType* argument in *SQLBindCol*(). In this argument, the application passes the C type
1572 identifier corresponding to the type of the buffer. For example, in the following call to
1573 *SQLBindCol*(), the value SQL_C_TYPE_DATE tells the implementation that the Date buffer is
1574 a SQL_DATE_STRUCT.

```
1575 SQL_DATE_STRUCT Date;
1576 SQLINTEGER      DateInd;
1577 SQLBindCol(hstmt, 1, SQL_C_TYPE_DATE, &Date, 0, &DateInd);
```

1578 For more information on type identifiers, see Section 4.4 on page 46.

1579 - **Predefined type**

1580 Buffers used to send and retrieve options or attributes, such as the buffer pointed to by the
1581 *InfoValuePtr* argument in *SQLGetInfo*(), have a fixed type that depends on the option
1582 specified. The implementation assumes that the data buffer is of this type; it is the
1583 application's responsibility to allocate a buffer of this type. For example, in the following call
1584 to *SQLGetInfo*(), the implementation assumes the buffer is a 32-bit integer because this is
1585 what the SQL_STRING_FUNCTIONS option requires:

```
1586 SQLUINTEGER StringFuncs;
1587 SQLGetInfo(hdbc, SQL_STRING_FUNCTIONS, (SQLPOINTER)
1588    &StringFuncs, 0, NULL);
```

1589 The implementation uses the C data type to interpret the data in the buffer.

1590 **Data Buffer Address**

1591 The application passes the address of the data buffer to the implementation in an argument with
1592 a name such as *ValuePtr*. For example, in the following call to *SQLBindCol*(), the application
1593 specifies the address of the Date variable.

```
1594 SQL_DATE_STRUCT Date;
1595 SQLINTEGER      DateInd;
1596 SQLBindCol(hstmt, 1, SQL_C_TYPE_DATE, &dsDate, 0, &DateInd);
```

1597 As mentioned in Section 4.3.2 on page 39, the address of a deferred buffer must remain valid
1598 until the buffer is unbound.

1599 Unless it is specifically prohibited, the address of a data buffer can be a null pointer. For buffers
1600 used to send data to the implementation, this makes the implementation ignore the information
1601 normally contained in the buffer. For buffers used to retrieve data from the implementation, this
1602 inhibits the implementation from returning a value. In both cases, the implementation ignores
1603 the corresponding data buffer length argument.

1604 **Data Buffer Length**

1605 The application passes the octet length of the data buffer to the implementation in an argument
1606 with a name such as *BufferLength*. For example, in the following call to *SQLBindCol*(), the
1607 application specifies the length of the **ValuePtr* buffer (**sizeof**(*ValuePtr*)).

```
1608 SQLCHAR    *ValuePtr[50];
1609 SQLINTEGER  ValueLenOrInd;
1610 SQLBindCol(hstmt, 1, SQL_C_CHAR, ValuePtr, sizeof(ValuePtr),
1611            &ValueLenOrInd);
```

1612 Data buffer lengths are required only for output buffers; the implementation uses them to avoid
1613 writing past the end of the buffer. However, the implementation checks the data buffer length
1614 only when the buffer contains variable-length data, such as character or binary data. If the buffer
1615 contains fixed-length data, such as an integer or date structure, the implementation ignores the
1616 data buffer length and assumes the buffer is large enough to hold the data; that is, it never
1617 truncates fixed-length data. It is therefore important for the application to allocate a large
1618 enough buffer for fixed-length data.

1619 Data buffer lengths aren't required for input buffers because the implementation doesn't write to
1620 these buffers.

1621 4.3.5 Using Length/Indicator Values

1622 The length/indicator buffer is used to pass the octet length of the data in the data buffer or a
1623 special indicator such as SQL_NULL_DATA, which indicates that the data is NULL. Depending
1624 on the function in which it is used, a length/indicator buffer is defined to be an SQLINTEGER or
1625 an SQLSMALLINT. Therefore, a single argument is needed to describe it. If the data buffer is a
1626 non-deferred input buffer, this argument contains the octet length of the data itself or an
1627 indicator value. It is often named *StrLen_or_Ind* or a similar name. For example, the following
1628 code calls *SQLPutData*() to pass a buffer full of data; the octet length (*ValueLen*) is passed directly
1629 because the data buffer (*ValuePtr*) is an input buffer.

```
1630 SQLCHAR    ValuePtr[50];
1631 SQLINTEGER ValueLen;
1632 // Call local function to place data in ValuePtr. In ValueLen,
1633 // return the number of octets of data placed in ValuePtr. If there
1634 // is not enough data, this will be less than 50.
1635 FillBuffer(ValuePtr, sizeof(ValuePtr), &ValueLen);
1636 // Call SQLPutData to send the data.
1637 SQLPutData(hstmt, ValuePtr, ValueLen);
```

1638 If the data buffer is a deferred input buffer, a non-deferred output buffer, or an output buffer, the
1639 argument contains the address of the length/indicator buffer. It is often named *StrLen_or_IndPtr*
1640 or a similar name. For example, the following code calls *SQLGetData*() to retrieve a buffer full of
1641 data; the octet length is returned to the application in the length/indicator buffer
1642 (*ValueLenOrInd*), whose address is passed to *SQLGetData*() because the corresponding data
1643 buffer (*ValuePtr*) is a non-deferred output buffer.

```
1644 SQLCHAR    ValuePtr[50];
1645 SQLINTEGER ValueLenOrInd;
1646 SQLGetData(hstmt, 1, SQL_C_CHAR, ValuePtr, sizeof(ValuePtr),
1647            &ValueLenOrInd);
```

1648 Unless it is specifically prohibited, a length/indicator buffer argument can be 0 (if non-deferred
1649 input) or a null pointer (if output or deferred input). For input buffers, this causes the
1650 implementation to ignore the octet length of the data. This is an error when passing variable-

length data but is common when passing non-null fixed-length data, as neither a length nor an indicator value is needed. For output buffers, this causes the implementation to not return the octet length of the data or an indicator value. This is an error if the data returned by the implementation is NULL but is common when retrieving fixed-length, non-nullable data as neither a length nor an indicator value is needed.

As with the address of a deferred data buffer, 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:

- A length greater than 0.
- 0.
- SQL_NTS. A string sent to the implementation in the corresponding data buffer is null terminated; this is a convenient way for C programmers to pass strings without having to calculate their octet length. This value is legal only when the application sends data to the implementation. When the implementation returns data to the application, it always returns the actual octet length of the data.

The following special length/indicator value can appear in the INDICATOR_PTR 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 implementation.

The following special length/indicator values can appear in the OCTET_LENGTH_PTR field:

- SQL_DATA_AT_EXEC. The data buffer doesn't contain any data. Instead, the data will be sent with *SQLPutData*() when the statement is executed or *SQLBulkOperations*() or *SQLSetPos*() is called. This value is legal only for SQL data sent to the implementation. For more information, see *SQLBindParameter*() and *SQLSetPos*().
- Result of the SQL_LEN_DATA_AT_EXEC(length) macro. This value is similar to SQL_DATA_AT_EXEC. For more information, see Section 9.4.3 on page 105.
- SQL_NO_TOTAL. The implementation cannot determine the number of octets of long data still available to return in an output buffer. This value is legal only for SQL data retrieved from the implementation.
- 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_IGNORE. The value in the data buffer should be ignored. When *SQLSetPos*() updates a row of data, the column value isn't changed. When *SQLBulkOperations*() or *SQLSetPos*() inserts a new row of data, the column value is set to its default or, if the column doesn't have a default, to NULL.

4.3.6 Data Length, Buffer Length, and Truncation

The *data length* is the octet 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 octet length of the data before conversion to the data source's type. For data being retrieved from the data source, this is the octet 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 octet 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 as the

implementation assumes the data buffer is the size of the data type and doesn't 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 octet length of the data is separate from and often different from the octet length of the buffer. If the octet length of the data is greater than the octet length of the buffer, the implementation truncates data being fetched to the octet length of the buffer and returns SQL_SUCCESS_WITH_INFO with SQLSTATE 01004 (Data truncated). However, the returned octet length is the length of the untruncated data.

For example, suppose an application allocates 50 octets for a binary data buffer. If the implementation has 10 octets of binary data to return, it returns those 10 octets in the buffer. The octet length of the data is 10 and the octet length of the buffer is 50. If the implementation has 60 octets of binary data to return, it truncates the data to 50 octets, returns those octets in the buffer and returns SQL_SUCCESS_WITH_INFO. The octet length of the data is 60 (the length before truncation) and the octet length of the buffer is still 50.

A diagnostic record is created for each column that is truncated. Because it takes time for the implementation 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 isn't true in all cases.

4.3.7 Character Data and C Strings

Null Termination

Input parameters that reference 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, then it provides as an argument either the length in octets of the string (not including the null terminator) or SQL_NTS (Null-terminated String).

Thus, 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 implementation to determine the length of the string by locating the null terminator.

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

When character data is passed from the application to the implementation, the application can null-terminate it; this is required only when the application passes SQL_NTS instead of the actual octet length of the data in the length/indicator buffer. If the data source does not use null termination, the implementation strips any null terminator from the end of the string before sending it to the data source.

When character data is returned from the implementation to the application, the implementation 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 isn't large enough to return all of the character data, the implementation truncates it to the octet length of the buffer less the number of octets required by the null terminator, null-terminates the truncated data, and stores it in the buffer. Thus, applications must always allocate extra space for the null terminator in buffers

used to retrieve character data. For example (assuming a single-octet character set), a 51-octet buffer is needed to retrieve 50 characters of data.

Special care must be taken by both the application and implementation 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 terminator on these strings must be stripped before the data can be reassembled.

C Language

When C strings are used to hold character data, the null terminator isn't considered to be part of the data and isn't counted as part of its octet length. For example, the character data 'ABC' can be held as the C string 'ABC\0' or the character array {'A', 'B', 'C'}. The octet length of the data is three regardless of whether it is treated as a string or a character array.

Although applications and implementations 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 octet length passed separately in the length/indicator buffer.

1758 ## 4.4 Data Types in XDBC

1759 XDBC provides for two varieties of data types:

1760 • *SQL data types* describe how values are represented at the data source.

1761 • *C data types* describe how values are represented in application variables, using the C
1762 language as a model.

1763 ### 4.4.1 Type Identifiers

1764 To describe SQL and C data types, XDBC defines two sets of *type identifiers*. A type identifier
1765 describes the type of an SQL column or a C buffer. It is a **#define** value and is generally passed
1766 as a function argument or returned in metadata. For example, the following call to
1767 *SQLBindParameter*() binds a variable of type SQL_DATE_STRUCT to a date parameter in an SQL
1768 statement. The C type identifier SQL_C_TYPE_DATE specifies the type of the Date variable and
1769 the SQL type identifier SQL_TYPE_DATE specifies the type of the dynamic parameter.

```
1770 SQL_DATE_STRUCT Date;
1771 SQLINTEGER      DateInd = 0;
1772 SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_TYPE_DATE,
1773                  SQL_TYPE_DATE, 0, 0, &Date, 0, &DateInd);
```

1774 ### 4.4.2 SQL Data Types in XDBC

1775 SQL data types are the types in which data is stored in the data source.

1776 #### SQL Type Identifiers

1777 Each data source defines its own SQL data types. XDBC defines SQL type identifiers and
1778 describes the general characteristics of the SQL data types that might be mapped to each type
1779 identifier. It is implementation-defined how each data type in the underlying data source maps
1780 to an SQL type identifier of XDBC.

1781 For example, SQL_CHAR is the type identifier for a character column with a fixed length,
1782 typically between 1 and 254 characters. These characteristics correspond to the CHAR data type
1783 found in many SQL data sources. Thus, when an application discovers that the type identifier for
1784 a column is SQL_CHAR, it can assume it is probably dealing with a CHAR column. However, it
1785 should still check the octet length of the column before assuming it is between 1 and 254
1786 characters; the implementation for a non-SQL data source, for example, might map a fixed-
1787 length character column of 500 characters to SQL_CHAR or SQL_LONGVARCHAR, since
1788 neither is an exact match.

1789 XDBC defines a wide variety of SQL type identifiers. However, the implementation isn't
1790 required to use all of these identifiers. Instead, it only uses those identifiers it needs to expose the
1791 SQL data types supported by the underlying data source. If the underlying data source supports
1792 SQL data types to which no type identifier corresponds, the implementation can define
1793 additional type identifiers.

1794 For a complete description of SQL type identifiers, see Appendix D.

Retrieving Data Type Information with SQLGetTypeInfo()

Because the mappings from underlying SQL data types to XDBC type identifiers are approximate, XDBC provides a function (*SQLGetTypeInfo*()) through which a implementation 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 is generally 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.

- XDBC doesn't define data type names for use in CREATE TABLE and ALTER TABLE statements, since the names of SQL data types vary (more widely than other aspects of SQL). Instead, the application should use the name returned in the TYPE_NAME column of the result set returned by *SQLGetTypeInfo*(). Rather than forcing implementations to parse SQL statements and replace standard data type names with data-source-specific data type names, XDBC requires applications to use the data-source-specific names in the first place.

SQLGetTypeInfo() doesn't necessarily describe all 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 XDBC 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*().

4.4.3 C Data Types in XDBC

XDBC 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.

XDBC also defines a default mapping from each SQL data type to a C data type. For example, a 2-octet integer in the data source is mapped to a 2-octet 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.

1836 ### 4.4.4 Data Type Conversions

1837 Data can be converted from one type to another at one of four times: when data is transferred
1838 from one application variable to another (C to C), when data in an application variable is sent to
1839 a statement parameter (C to SQL), when data in a result set column is returned in an application
1840 variable (SQL to C), and when data is transferred from one data source column to another (SQL
1841 to SQL).

1842 Any conversion that occurs when data is transferred from one application variable to another is
1843 outside the scope of this specification.

1844 When an application binds a variable to a result set column or statement parameter, it implicitly
1845 specifies a data type conversion in its choice of the data type of the application variable. For
1846 example, suppose a column contains integer data. If the application binds an integer variable to
1847 the column, it specifies that no conversion be done; if it binds a character variable to the column,
1848 it specifies that the data be converted from integer to character.

1849 XDBC defines how data is converted between each SQL and C data type. Basically, it supports
1850 all reasonable conversions, such as character to integer and integer to float, and doesn't support
1851 ill-defined conversions, such as float to date. Implementations are required to support all
1852 conversions for each SQL data type they support. For a complete list of conversions from SQL to
1853 C data types, see Section D.6 on page 576. For a complete list of conversions from C to SQL data
1854 types, see Section D.7 on page 587.

1855 The following functions convert data at the data source from one SQL data type to another:

1856 - The CAST function defined in the X/Open **SQL** specification.

1857 - The **CONVERT** scalar function defined in Section F.5 on page 609. An escape sequence by
1858 which portable applications can invoke **CONVERT** is described in Section 8.3.3 on page 86.

1859 An XDBC implementation maps the **CONVERT** scalar function to the underlying scalar
1860 function or functions defined to perform conversions in the data source. Because it is
1861 mapped to data-source-specific functions, XDBC doesn't define how these conversions work
1862 or what conversions must be supported.

4.5 Environment, Connection, and Statement Attributes

A number of attributes are associated with the environment, connection, or statement.

Environment attributes affect the entire environment, such as whether strings can be null-terminated. Environment attributes are set with *SQLSetEnvAttr*() and retrieved with *SQLGetEnvAttr*().

Connection attributes affect each connection individually, such as how long an implementation should wait while attempting to connect to a data source before timing out. Connection attributes are set with *SQLSetConnectAttr*() and retrieved with *SQLGetConnectAttr*(). Connection attributes are discussed further in Section 6.3 on page 60.

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*(). Statement attributes can also be set with *SQLSetConnectAttr*(), in which case it applies to all statements on the connection and becomes the default for any new statements. However, statement attributes cannot be retrieved by calling *SQLGetConnectAttr*(). Statement attributes are discussed further in Section 9.2 on page 93.

A few statement attributes are read-only attributes and cannot be set. For example, the SQL_ATTR_ROW_NUMBER statement attribute is used to retrieve the number of the current row in the cursor.

In addition to attributes defined by XDBC, an implementation can define its own connection and statement attributes. Vendor-defined attributes must be registered with X/Open (see Section 1.8 on page 21) to ensure that two vendors do not assign the same integer value to different, proprietary attributes.

For a complete list of attributes, see *SQLSetConnectAttr*(), *SQLSetStmtAttr*(), and *SQLGetStmtAttr*(). Most attributes are also described in the description of the XDBC function that they affect.

1888 *Chapter 5*

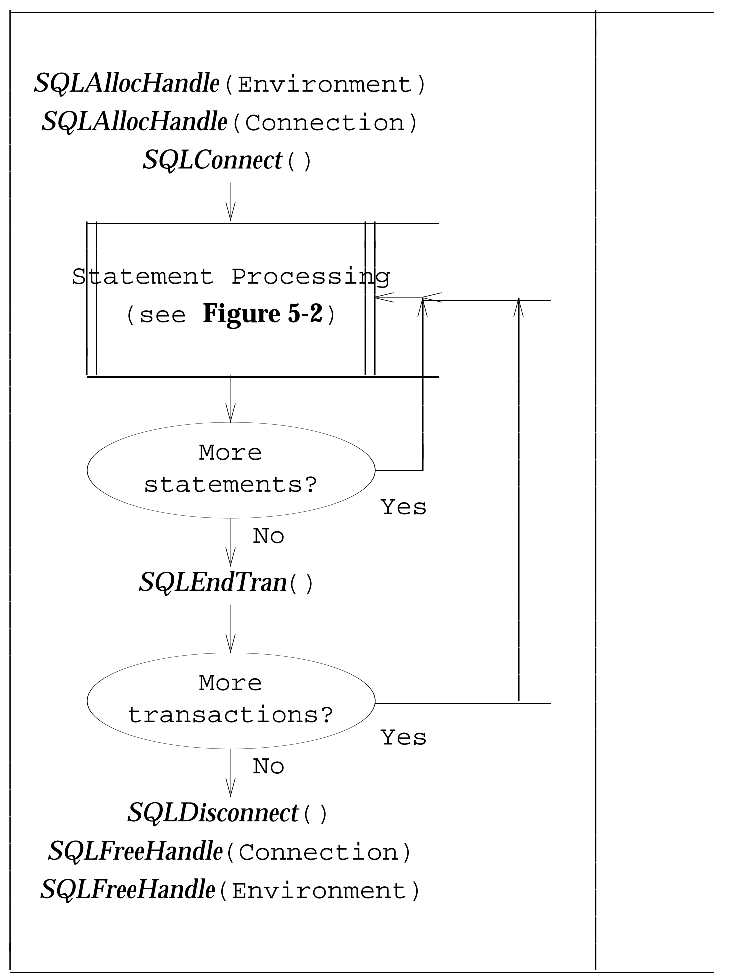
1889 Basic Application Steps

1890 The following diagrams show the basic flow of control for the use of XDBC functions. These
1891 diagrams assume (and do not illustrate) that the application performs error checking where
1892 appropriate using *SQLGetDiagField*() or *SQLGetDiagRec*().

1893 For more detailed information concerning control flow and function sequencing rules, refer to
1894 the state transition tables in Appendix B.

1895 ## 5.1 Basic Control Flow

1896 The following figure shows the initiation sequence, the termination sequence and an overview of
1897 transaction completion. A discussion of the major steps, and references to other sections of this
1898 specification, follow the figure.



1899

1900 Figure **5-1.** Initiation, Termination and Transaction Completion

1901 **Connecting to the Data Source**

1902 The first step in connecting to the data source is to allocate the environment handle with
1903 *SQLAllocHandle*(). For more information, see Section 6.1 on page 58.

1904 Next, the application allocates a connection handle with *SQLAllocHandle*() and connects to the
1905 data source with *SQLConnect*(), *SQLDriverConnect*(), or *SQLBrowseConnect*(). For more
1906 information on allocating a connection handle, see Section 6.2 on page 59. Various connection
1907 methods are discussed later in Chapter 6.

1908 The application then sets any connection attributes, such as whether to manually commit
1909 transactions. For more information, see Section 6.3 on page 60.

1910 After connecting to a data source, it is also typical to call *SQLGetInfo*() to determine its
1911 capabilities.

1912 **Completing the Transaction**

1913 The application calls *SQLEndTran*() to commit or roll back the transaction. The application only
1914 performs this step if it set the transaction commit mode to manual commit; if the transaction
1915 commit mode is auto-commit, which is the default, the transaction is automatically committed
1916 when the statement is executed. For more information, see Chapter 14.

1917 **Disconnecting from the Data Source**

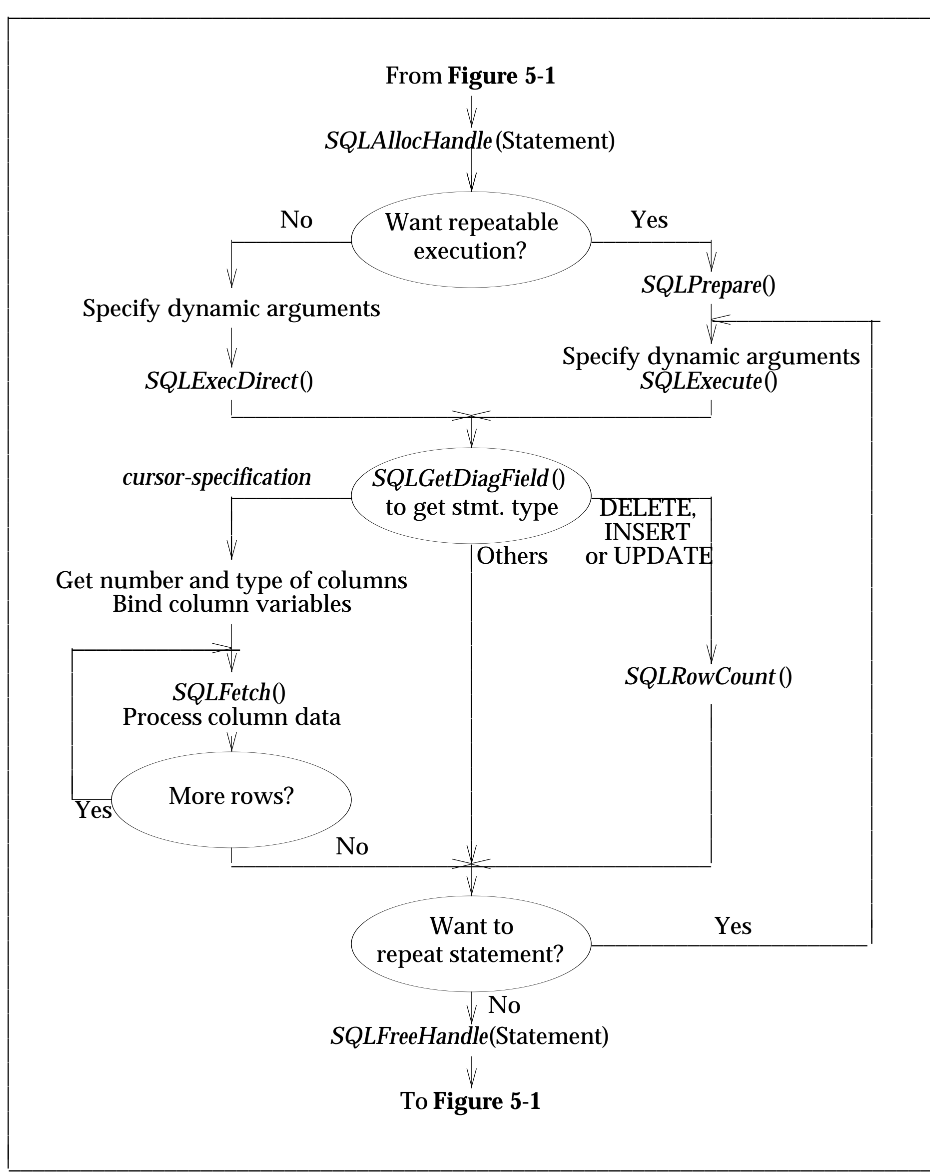
1918 The application frees any statement handles by calling *SQLFreeHandle*(). For more information,
1919 see Section 9.6 on page 124. Next, the application disconnects from the data source with
1920 *SQLDisconnect*() and frees the connection handle with *SQLFreeHandle*(). For more information,
1921 see Section 6.5 on page 64.

1922 Finally, the application frees the environment handle with *SQLFreeHandle*(). For more
1923 information, see Section 6.1 on page 58.

1924 ## 5.2 Example Control Flow for SQL Statement Processing

1925 Figure 5-2 on page 54 shows typical control flow for processing SQL statements, including the
1926 allocation and release of a statement handle.

1927 While this is the basic control flow for SQL statements executed interactively, there are other
1928 valid sequences, such as modifying the buffer descriptor between successive fetches. A
1929 discussion of the major steps, and references to other sections of this specification, follow the
1930 figure.



1931

1932 Figure 5-2. Example Control Flow for Statement Processing

1933 **Allocating a Statement Handle**

1934 All applications need to allocate a statement handle with *SQLAllocHandle*() as described in
1935 Section 9.1 on page 92. After doing this, many applications set statement attributes, such as the
1936 cursor type, with *SQLSetStmtAttr*(), as described in Section 9.2 on page 93.

1937 **Building and Executing an SQL Statement**

1938 There are many ways to generate and execute an SQL statements. The application might prompt
1939 the user to enter the statement, build the statement based on user input, or use a hard-coded SQL
1940 statement. For more information, see Chapter 8.

1941 If the SQL statement contains parameters, the application binds them to application variables by
1942 calling *SQLBindParameter*() for each parameter. For more information, see Section 9.4 on page
1943 102.

1944 After the SQL statement is built and any parameters are bound, the statement is executed with
1945 *SQLExecDirect*(). If the statement will be executed multiple times, it can be prepared with
1946 *SQLPrepare*() and executed with *SQLExecute*(). For more information, see Section 9.3 on page 94.

1947 Instead of executing an SQL statement, the application might call a function to return a result set
1948 containing catalog information, such as the available columns or tables. For more information,
1949 see Chapter 7.

1950 What the application does next depends on the type of SQL statement executed:

1951 • If the SQL statement is a SELECT statement or a catalog function, the application can call
1952 *SQLNumResultCols*() to determine the number of columns in the result set.

1953 The application can retrieve the name, data type, precision, and scale of each result set
1954 column with *SQLDescribeCol*(). Again, this isn't necessary for applications such as vertical
1955 and custom applications that already know this information. It passes this information to
1956 *SQLBindCol*(), which binds an application variable to a column in the result set.

1957 The application now calls *SQLFetch*() to retrieve the first row of data and place the data from
1958 that row in the variables bound with *SQLBindCol*(). If there is any long data in the row, it
1959 then calls *SQLGetData*() to retrieve that data. The application continues to call *SQLFetch*()
1960 and *SQLGetData*() to retrieve additional data. After it has finished fetching data, it calls
1961 *SQLCloseCursor*() to close the cursor.

1962 For a complete description of retrieving results, see Chapter 10 and Chapter 11.

1963 • If the statement executed was DELETE, INSERT, or UPDATE, the application can retrieve the
1964 count of affected rows with *SQLRowCount*(). For more information, see Section 12.2 on page
1965 162.

Chapter 6

Connecting to a Data Source

An application can be connected to any number of XDBC implementations. These can be a variety of client-side implementations and data sources, the same implementation and a variety of data sources, or multiple connections to the same implementation and data source.

1971 ## 6.1 Allocating the Environment Handle

1972 Before an application can call any other XDBC function, it must initialize the XDBC environment
1973 and allocate an environment handle. To do this:

1974 • The application declares a variable of type SQLHENV. It then calls *SQLAllocHandle*() and
1975 passes the address of this variable and the SQL_HANDLE_ENV option. For example:

```
1976 SQLHENV henv1;
1977 SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv1);
```

1978 • The XDBC implementation initializes itself, allocates a structure in which to store
1979 information about the environment, and returns the environment handle in the variable.

1980 When the application has finished using XDBC, it frees the environment handle with
1981 *SQLFreeHandle*(). After freeing the environment, it's an application programming error to use
1982 the environment's handle in a call to an XDBC function; doing so has undefined but probably
1983 fatal consequences.

1984 When *SQLFreeHandle*() is called, the implementation releases the structure used to store
1985 information about the environment. *SQLFreeHandle*() cannot be called for an environment
1986 handle until after all connection handles on that environment handle have been freed.

1987 For more information about the environment handle, see Section 4.1.1 on page 35.

1988 ## 6.2 Allocating a Connection Handle

1989 Before the application can connect to a data source, it must allocate a connection handle. To do
1990 this:

1991 - The application declares a variable of type SQLHDBC. It then calls *SQLAllocHandle*() and
1992 passes the address of this variable, the handle of the environment in which to allocate the
1993 connection, and the SQL_HANDLE_DBC option. For example:

```
1994 SQLHDBC hdbc1;
1995 SQLAllocHandle(SQL_HANDLE_DBC, henv1, &hdbc1);
```

1996 - The implementation allocates a structure in which to store information about the statement
1997 and returns the connection handle in the variable.

1998 For more information about connection handles, see Section 4.1.2 on page 35.

1999 ## 6.3 Connection Attributes

2000 Connection attributes are characteristics of the connection. For example, because transactions
2001 occur at the connection level, the transaction isolation level is a connection attribute. Similarly,
2002 the login timeout, or number of seconds to wait while trying to connect before timing out, is a
2003 connection attribute.

2004 Connection attributes are set with *SQLSetConnectAttr*() and their current settings retrieved with
2005 *SQLGetConnectAttr*(). There is no requirement that an application set any connection attributes;
2006 all connection attributes have defaults, some of which are implementation-specific.

2007 The proper time to set a connection attribute varies among attributes:

2008 • The login timeout (SQL_ATTR_LOGIN_TIMEOUT) and the network packet size
2009 (SQL_ATTR_PACKET_SIZE) apply to the connection process and must be set before
2010 connecting.

2011 • For certain other connection attributes, portable applications must specify any changes to the
2012 default values before connecting. These include the attributes to specify whether a data
2013 source is read-only or read-write (SQL_ATTR_ACCESS_MODE) and the current catalog
2014 (SQL_ATTR_CURRENT_CATALOG). Some implementations also let applications change
2015 these after connecting.

2016 • Other connection attributes can be set at any time.

2017 For more information, see *SQLSetConnectAttr*().

6.4 Establishing a Connection

After allocating environment and connection handles and setting any connection attributes, the application is ready to connect to the data source. There are three different functions the application can use to do this: *SQLConnect*(), *SQLDriverConnect*(), and *SQLBrowseConnect*(). 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 implementations 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 are supported.

6.4.1 Default Data Source

The implementation 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 *ServerName* is a zero-length string, a null pointer, or DEFAULT.
- In a call to *SQLDriverConnect*() where *InConnectionString* either specifies **DSN=DEFAULT** or specifies with **DSN** a data source that is not contained in the system information.

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

6.4.2 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 don't require a user ID or password. It also works well for applications that want to control their own ''look and feel.'' 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*(). *SQLConnect*() is also appropriate for applications that do not have a user interface.

6.4.3 Connection String

A *connection string* is a string. 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 reference manual entry.) The connection string is used by:

- *SQLDriverConnect*() (see Section 6.4.4), which completes it by interaction with the user.
- *SQLBrowseConnect*() (see Section 6.4.5 on page 62), 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).

2056 ### 6.4.4 Connecting with SQLDriverConnect()

2057 *SQLDriverConnect*() is used instead of *SQLConnect*() for the following reasons:

2058 • To let the application use implementation-specific connection information.

2059 • To request that the implementation prompt the user for connection information.

2060 • To connect without specifying a data source.

2061 If the application uses *SQLConnect*() and needs to prompt the user for any connection
2062 information, such as a user name and password, it must do so itself. This lets the application
2063 control the user interface but might force it to contain implementation-specific connection
2064 information. This is infeasible for generic applications, which must work with any and all
2065 implementations, including implementations that don't exist when the application is written.

2066 *SQLDriverConnect*() can prompt the user for connection information. For example, a custom
2067 program could pass the following connection string to *SQLDriverConnect*():

2068
```
DSN=XYZ Corp;
```

2069 The implementation might then display a dialog box to prompt for user IDs and passwords.

2070 The ability to prompt for connection information is particularly useful to generic and vertical
2071 applications because it keeps implementation-specific information out of the application. This is
2072 shown by the previous example. By passing only the data source name to the implementation,
2073 the application didn't contain any implementation-specific information and was therefore not
2074 tied to a particular implementation. (An application could also pass *SQLDriverConnect*() a
2075 complete connection string, even though this would tie the application to an implementation
2076 that could interpret that string.)

2077 A generic application might take this one step further and not even specify a data source. When
2078 *SQLDriverConnect*() receives an empty connection string, the implementation interacts with the
2079 user. After the user selects a data source, the implementation constructs a connection string
2080 specifying that data source.

2081 ### 6.4.5 Connecting with SQLBrowseConnect()

2082 By using *SQLBrowseConnect*(), an application can construct a complete connection string at run
2083 time. This lets the application do two things:

2084 • Build its own dialog boxes to prompt for this information, thereby retaining control over its
2085 ''look and feel.''

2086 • *Browse* the system for data sources that can be used by a particular implementation, possibly
2087 in several steps. For example, the user might first browse the network for servers and, after
2088 choosing a server, browse the server for databases accessible by the implementation.

2089 The application calls *SQLBrowseConnect*() and passes a connection string, known as the ***browse***
2090 ***request connection string***, that specifies a data source. The implementation returns a connection
2091 string, known as the ***browse result connection string***, that contains keywords, possible values (if the
2092 keyword accepts a discrete set of values), and user-friendly names. The application builds a
2093 dialog box with the user-friendly names and prompts the user for values. It then builds a new
2094 browse request connection string from these values and returns this to the implementation with
2095 another call to *SQLBrowseConnect*().

2096 Because connection strings are passed back and forth, the implementation can provide several
2097 levels of browsing by returning a new connection string when the application returns the old
2098 one. For example, the first time an application calls *SQLBrowseConnect*(), the implementation
2099 might return keywords to prompt the user for a server name. When the application returns the
2100 server name, the implementation might return keywords to prompt the user for a database. The

2101 browsing process would be complete after the application returned the database name.

2102 Each time *SQLBrowseConnect*() returns a new browse result connection string, it returns
2103 SQL_NEED_DATA as its return code. This tells the application that the connection process isn't
2104 complete. Until *SQLBrowseConnect*() returns SQL_SUCCESS, the connection is in a Need Data
2105 state and cannot be used for other purposes, such as to set a connection attribute. The
2106 application can terminate the connection browsing process by calling *SQLDisconnect*().

6.5 Disconnecting from a Data Source

When an application has finished using a data source, it calls *SQLDisconnect*(). This frees any statements that are allocated on the connection and disconnects 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's an application programming error to use the connection's handle in a call to an XDBC function; doing so has undefined but probably fatal consequences. When *SQLFreeHandle*() is called, the implementation releases the structure used to store information about the connection.

The application can also 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 trade-off, the application must also make assumptions about the likelihood and timing of further operations on the same data source.

Chapter 7

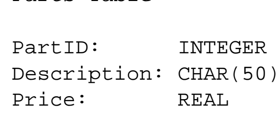
Catalog Functions

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 figure, in which the ID columns are used to link the tables.

```
Orders Table

OrderID:     INTEGER
CustID:      INTEGER
OpenDate:    DATE
SalesPerson: CHAR(10)
Status:      CHAR(6)
```

```
Lines Table

OrderID:  INTEGER
Line:     INTEGER
PartID:   INTEGER
Quantity: INTEGER
```

```
Customers Table

CustID:  INTEGER
Name:    CHAR(50)
Address: CHAR(50)
Phone:   CHAR(10)
```



```
Parts Table

PartID:      INTEGER
Description: CHAR(50)
Price:       REAL
```

```
Pictures Table

PartID:  INTEGER
Picture: LONG VARBINARY
```

Figure **7-1.** Sales Order Database Structure

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. 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.

2146 ## 7.1 Uses of Catalog Data

2147 Here are some common ways in which applications use catalog data:

2148 • **Constructing SQL statements at run time.**

2149 Vertical applications, such as an order entry application, contain hard-coded SQL statements.
2150 The tables and columns that are used by the application are fixed ahead of time, as are the
2151 statements that access these tables. For example, an order entry application usually contains
2152 a single, parameterized INSERT statement for adding new orders to the system.

2153 Generic applications, such as a spreadsheet program that uses XDBC to retrieve data, often
2154 construct SQL statements at run time based on input from the user. Such an application
2155 could require the user to type the names of the tables and columns to use. However, it would
2156 be easier for the user if the application displayed lists of tables and columns from which the
2157 user could make selections. To build these lists, the application would call *SQLTables*() and
2158 *SQLColumns*().

2159 • **Constructing SQL statements during development.**

2160 Application development environments typically allow the programmer to create database
2161 queries while developing a program. The queries are then hard-coded in the application
2162 being built.

2163 Such environments could also use *SQLTables*() and *SQLColumns*() to create lists from which
2164 the programmer could make selections. They might also use *SQLPrimaryKeys*() and
2165 *SQLForeignKeys*() to automatically determine and show relationships between selected
2166 tables, and use *SQLStatistics*() to determine and highlight indexed fields so the programmer
2167 can create efficient queries.

2168 • **Constructing cursors.**

2169 An application, XDBC implementation, or other software component that simulates
2170 scrollable cursors, can use *SQLSpecialColumns*() to determine which column or columns
2171 uniquely identify a row. The program could build a *keyset* containing the values of these
2172 columns for each row that has been fetched. When the application scrolls back to the row, it
2173 would then use these values to fetch the most recent data for the row. For more information
2174 about scrollable cursors and keysets, see Chapter 11.

2175 ## 7.2 Catalog Functions

2176 XDBC contains the following catalog functions:

SQLTables()	Returns a list of catalogs, schemas, tables, or table types in the data source.
SQLColumns()	Returns a list of columns in one or more tables.
SQLStatistics()	Returns a list of statistics about a single table. Also returns a list of indexes associated with that table.
SQLSpecialColumns()	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.
SQLPrimaryKeys()	Returns a list of columns that compose the primary key of a single table.
SQLForeignKeys()	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.
SQLTablePrivileges()	Returns a list of privileges associated with one or more tables.
SQLColumnPrivileges()	Returns a list of privileges associated with one or more columns in a single table.
SQLProcedures()	Returns a list of procedures in the data source.
SQLProcedureColumns()	Returns a list of input and output parameters, the return value, and the columns in the result set of a single procedure.
SQLGetTypeInfo()	Returns a list of the SQL data types supported by the data source. These data types are generally used in CREATE and ALTER TABLE statements.

2198 *SQLTables*(), *SQLColumns*(), *SQLStatistics*(), *SQLSpecialColumns*(), and *SQLGetTypeInfo*() are in
2199 XDBC Level 1. The remaining catalog functions in XDBC Level 2.

7.3 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 implementation or stored in a procedure in the data source. For information on how to retrieve data from a result set, see Chapter 10.

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 implementation-defined columns after the last predefined column.

Applications should bind implementation-defined columns relative to the end of the result set. That is, they should calculate the number of an implementation-defined column as the number of the last column (retrieved with *SQLNumResultCols*()) less the number of columns that occur after the column to be bound. This obviates changing the application when new columns are added to the result set in future XDBC implementations. (For this scheme to work, new implementation-defined columns must be located before old implementation-defined columns, so that column numbers don't change relative to the end of the result set.)

Identifiers that are returned in the result set aren't quoted, even if they contain special characters. For example, suppose the identifier quote character (which is implementation-defined 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** on page 82.

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

7.3.1 COLUMN_DEF Column

In the result set returned by the *SQLColumns*() and *SQLProcedureColumns*() catalog functions, there is a COLUMN_DEF column that specifies a column default value.

The value of COLUMN_DEF uses legal syntax for *default-value* in the *column-definition* of the CREATE TABLE or ALTER TABLE statement defined in the X/Open **SQL** specification. If the default value is a character string, then this column is that string enclosed in single quotes. If the default value is a numeric literal, then this column contains the original character representation with no enclosing single quotes. If the default value is a date/time or interval literal, then the column contains the appropriate keyword followed by the date/time or interval value enclosed in single quotes; and, for an interval literal, terminated by the *interval-qualifier* syntactic element defined in the X/Open **SQL** specification (for example, 'YEAR TO MONTH') If the default value is a *pseudo-literal*, then this column contains the keyword, such as CURRENT_DATE, with no enclosing single quotes.

If NULL was specified as the default value, then this column is the word NULL, not enclosed in quotes. If the default value cannot be represented without truncation, then this column contains TRUNCATED with no enclosing single quotes. If no default value was specified, then this column is null.

The value of COLUMN_DEF is suitable for use in generating a new *column-definition*, except when it contains the value TRUNCATED.

7.4 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);
```

2255 Catalog function string arguments can be interpreted in four different ways. The arguments are
2256 termed ordinary arguments (OA), pattern value arguments (PV), identifier arguments (ID), and
2257 value list arguments (VL); these types are defined following the table. Interpretation usually
2258 depends on the value of the SQL_ATTR_METADATA_ID statement attribute. The following
2259 table specifies the interpretation of each argument of each catalog function.

			SQL_ATTR_METADATA_ID=	
	Function	**Argument**	**SQL_FALSE**	**SQL_TRUE**
2262	*SQLColumnPrivileges*()	*CatalogName*	OA	ID
2263		*SchemaName*	OA	ID
2264		*TableName*	OA	ID
2265		*ColumnName*	PV	ID
2266	*SQLColumns*()	*CatalogName*	OA	ID
2267		*SchemaName*	PV	ID
2268		*TableName*	PV	ID
2269		*ColumnName*	PV	ID
2270	*SQLForeignKeys*()	*PKCatalogName*	OA	ID
2271		*PKSchemaName*	OA	ID
2272		*PKTableName*	OA	ID
2273		*FKCatalogName*	OA	ID
2274		*FKSchemaName*	OA	ID
2275		*FKTableName*	OA	ID
2276	*SQLPrimaryKeys*()	*CatalogName*	OA	ID
2277		*SchemaName*	OA	ID
2278		*TableName*	OA	ID
2279	*SQLProcedureColumns*()	*CatalogName*	OA	ID
2280		*SchemaName*	PV	ID
2281		*ProcName*	PV	ID
2282		*ColumnName*	PV	ID
2283	*SQLProcedures*()	*CatalogName*	OA	ID
2284		*SchemaName*	PV	ID
2285		*ProcName*	PV	ID
2286	*SQLSpecialColumns*()	*CatalogName*	OA	ID
2287		*SchemaName*	OA	ID
2288		*TableName*	OA	ID
2289	*SQLStatistics*()	*CatalogName*	OA	ID
2290		*SchemaName*	OA	ID
2291		*TableName*	OA	ID
2292	*SQLTablePrivileges*()	*CatalogName*	OA	ID
2293		*SchemaName*	PV	ID
2294		*TableName*	PV	ID
2295	*SQLTables*()	*CatalogName*	PV	ID
2296		*SchemaName*	PV	ID
2297		*TableName*	PV	ID
2298		*TableType*	VL	VL

2299 Table **7-1.** Interpretation of String Arguments of Catalog Functions

2300 **Ordinary Arguments (OA)**

2301 When a catalog function string argument is an ordinary argument, it is treated as a literal string.
2302 An ordinary argument accepts neither a string search pattern nor a list of values. The case of an
2303 ordinary argument is significant, and quote characters in the string are taken literally. These
2304 arguments are treated as ordinary arguments if the SQL_ATTR_METADATA_ID statement
2305 attribute is set to SQL_FALSE; they are treated as identifier arguments instead if this attribute is
2306 set to SQL_TRUE.

2307 If an ordinary argument is set to a null pointer and the argument is a required argument, the
2308 function returns SQL_ERROR and SQLSTATE HY009 (Invalid use of null pointer). The
2309 following arguments are required arguments:

2310	Function	Arguments that cannot be a null pointer
2311	*SQLColumnPrivileges*()	*TableName*
2312	*SQLForeignKeys*()	*PKTableName*, *FKTableName*
2313	*SQLPrimaryKeys*()	*TableName*
2314	*SQLSpecialColumns*()	*TableName*
2315	*SQLStatistics*()	*TableName*

2316 **Pattern Value (PV) Arguments**

2317 Some arguments in the catalog functions, such as the *TableName* argument in *SQLTables*(), accept
2318 search patterns. These arguments accept search patterns if the SQL_ATTR_METADATA_ID
2319 statement attribute is set to SQL_FALSE; they are identifier arguments that do not accept a
2320 search pattern if this attribute is set to SQL_TRUE.

2321 The search pattern characters are:

2322 • An underscore (_), which represents any single character.

2323 • A percent sign (%), which represents any sequence of zero or more characters.

2324 • An escape character, which is implementation-defined and is used to include underscores,
2325 percent signs, and the escape character as literals.

2326 The escape character is retrieved with the SQL_SEARCH_PATTERN_ESCAPE option in
2327 *SQLGetInfo*(). It must precede any underscore, percent sign, or escape character in an argument
2328 that accepts search patterns to include that character as a literal. For example:

2329	Search pattern	Description
2330	`%A%`	All identifiers containing the letter A.
2331	`ABC_`	All four-character identifiers starting with ABC.
2332	`ABC_`	The identifier ABC_ (assuming the escape character is a backslash).
2333 2334	`\\%`	All identifiers starting with a backslash (assuming the escape character is a backslash).

2335 Special care must be taken to escape search pattern characters in arguments that accept search
2336 patterns. This is particularly true for the underscore character, which is commonly used in
2337 identifiers. A common mistake in applications is to retrieve a value from one catalog function
2338 and pass that value to a search pattern argument in another catalog function. For example,
2339 suppose an application retrieves the table name MY_TABLE from the result set for *SQLTables*()
2340 and passes this to *SQLColumns*() to retrieve a list of columns in MY_TABLE. Instead of getting
2341 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.

Passing a null pointer to a search pattern argument doesn't 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 (ID) Arguments

An identifier argument is treated as a quoted identifier whether or not it is actually quoted. If the string is quoted, the implementation removes leading and trailing blanks, and treats the string within the quotation marks literally. If the string is not quoted, the implementation 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; they 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 implementation-defined 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 a table whose name included double quotation marks:

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

Quoted identifiers should be 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, or with the 'ROWID' column if the pseudo-column does not exist.

For more information about quoted identifiers, see **Quoted Identifiers** on page 82.

Value List (VL) Arguments

A value list argument consists of a list of comma-separated values to be used for matching. Pattern values are legal within the list, and a null pointer is the same as a list containing a single value of '%'. These arguments are not affected by the SQL_ATTR_METADATA_ID statement attribute. There is only one value list argument in the XDBC catalog functions: the *TableType* argument in *SQLTables*().

Schema Views

An application can retrieve metadata information from the data source either by calling XDBC catalog functions or by using INFORMATION_SCHEMA views. These views provide applications with an alternative method for retrieving metadata. The views are defined by the ISO SQL standard.

If supported by the implementation, the INFORMATION_SCHEMA views provide a more powerful and comprehensive means of retrieving metadata than the XDBC catalog functions provide. An application can execute its own custom SQL query 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 often not supported. This may change as

2387 compliance with ISO SQL standard becomes more widespread.

2388 To determine which views are supported, an application calls *SQLGetInfo*() with the
2389 SQL_INFO_SCHEMA_VIEWS option. To retrieve metadata from a supported view, the
2390 application executes a SELECT statement that specifies the schema information required.

2391 *Chapter 8*

2392 SQL Statements

2393 XDBC applications perform almost all of their database access by submitting SQL statements to
2394 the XDBC implementation for execution.

2395 Section 8.1 on page 76 discusses different methods by which the text of the SQL statement is
2396 built. Section 8.2 on page 80 discusses the choice of portable SQL or proprietary SQL, and
2397 discusses several aspects of SQL grammar that affect an application's portability. Section 8.3 on
2398 page 84 defines the XDBC escape clause, which provides a standard syntax for features for
2399 which data sources tend to specify different syntax.

2400 ## 8.1 Building SQL Statements

2401 SQL statements can be built in one of three ways: hard-coded during development, built at run
2402 time, or entered directly by the user. The choice of method depends on the needs of the
2403 application.

2404 ### 8.1.1 Hard-Coded SQL Statements

2405 Applications that perform a fixed task usually contain hard-coded SQL statements. For example,
2406 an order entry system might use the following call to list open sales orders:

```
2407 SQLExecDirect(hstmt, "SELECT OrderID FROM Orders WHERE Status = 'OPEN'",
2408               SQL_NTS);
```

2409 There are several advantages to hard-coded SQL statements: they can be tested when the
2410 application is written, they are simpler to implement than statements built at run time, and they
2411 simplify the application.

2412 Using statement parameters and preparing statements provide even better ways to use hard-
2413 coded SQL statements. For example, suppose the Parts table contains the PartID, Description,
2414 and Price columns. One way to insert a new row into this table would be to generate and execute
2415 an INSERT statement:

```
2416 SQLUINTEGER PartID;
2417 SQLCHAR     *Desc, *Statement;
2418 SQLREAL     Price;
2419 // Allocate memory for Desc and Statement.  Code not shown.
2420 // Set part ID, description, and price.
2421 GetNewValues(&PartID, &Desc, &Price);
2422 // Build INSERT statement.
2423 sprintf(Statement, "INSERT INTO Parts (PartID, Description,Price) \
2424         VALUES (%d, '%s', %f)", PartID, Desc, Price);
2425 // Execute the statement
2426 SQLExecDirect(hstmt, Statement, SQL_NTS);
```

2427 An even better way is to use a hard-coded, parameterized statement. This has two advantages
2428 over a statement with hard-coded data values. First, it is easier to build a parameterized
2429 statement because the data values can be sent in their native types, such as integers and floating
2430 point numbers, rather than converting them to strings. Second, such a statement can be easily
2431 used more than once by simply changing the parameter values and reexecuting it; there is no
2432 need to rebuild it.

```
2433 SQLCHAR *Statement = "INSERT INTO Orders (PartID, Description, Price) \
2434                       VALUES (?, ?, ?)";
2435 SQLUINTEGER PartID;
2436 SQLCHAR     Desc[51];
2437 SQLREAL     Price;
2438 SQLINTEGER  PartIDInd = 0, DescLenOrInd = SQL_NTS, PriceInd = 0;
2439 // Bind the parameters.  We are assuming that the octet length of the
2440 // Description column is known to be 50.
2441 SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_ULONG, SQL_INTEGER,
2442                  5, 0, &PartID, 0, &PartIDInd);
2443 SQLBindParameter(hstmt, 2, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR,
2444                  50, 0, Desc, sizeof(Desc), &DescLenOrInd);
2445 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:

```
SQLCHAR *Statement = "INSERT INTO Orders (PartID, Description, Price) \
                      VALUES (?, ?, ?)";
SQLUINTEGER PartID;
SQLCHAR     Desc[51];
SQLREAL     Price;
SQLINTEGER  PartIDInd = 0, DescLenOrInd = SQL_NTS, PriceInd = 0;
// Prepare the INSERT statement.
SQLPrepare(hstmt, Statement, SQL_NTS);
// Bind the parameters.  We are assuming that the octet length of the
// Description column is known to be 50.
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,
                 50, 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 build a procedure containing the statement, as shown in the following code example. Because the procedure is built at development time and stored on the data source, it doesn't need to be prepared at run time. The syntax for creating procedures is data-source-specific and procedures must be built separately for each data source on which the application is to run.

```
SQLUINTEGER PartID;
SQLCHAR     Desc[51];
SQLREAL     Price;
SQLINTEGER  PartIDInd = 0, DescLenOrInd = SQL_NTS, PriceInd = 0;
// Bind the parameters.  Assume that the octet length of the
// Description column is known to be 50.
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,
                 50, 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);
```

2492 For more information on parameters, prepared statements, and procedures, see Chapter 9.

2493 8.1.2 SQL Statements Built at Run Time

2494 Applications that perform *ad hoc* analysis may build SQL statements at run time. For example, a
2495 spreadsheet might allow a user to select columns from which to retrieve data:

```
2496 SQLCHAR     *Statement, *TableName;
2497 SQLCHAR     **TableNamesArray, **ColumnNamesArray;
2498 BOOL        *ColumnSelectedArray;
2499 BOOL        CommaNeeded;
2500 SQLSMALLINT i, NumColumns;
2501 // Use SQLTables to build a list of tables (TableNamesArray[]). Let
2502 // the user select a table and store the selected table in TableName.
2503 // Use SQLColumns to build a list of the columns in the selected table
2504 // (ColumnNamesArray). Set NumColumns to the number of columns in the
2505 // table. Let the user select one or more columns and flag these
2506 // columns in ColumnSelectedArray[].  Build a SELECT statement from
2507 // the selected columns.
2508 CommaNeeded = FALSE;
2509 strcpy(Statement, "SELECT ");
2510 for (i = 0; i = NumColumns; i++) {
2511    if ColumnSelectedArray[i] {
2512       if CommaNeeded strcat(Statment, ",") else CommaNeeded = TRUE;
2513       strcat(Statement, ColumnNamesArray[i]);
2514    }
2515 }
2516 strcat(Statement, " FROM ");
2517 strcat(Statement, TableName);
2518 // Execute the statement directly. Because it will only be executed
2519 // once, do not prepare it.
2520 SQLExecDirect(hstmt, Statement, SQL_NTS);
```

2521 Another class of applications that commonly build SQL statements at run time are application
2522 development environments. However, the statements they build are hard-coded in the
2523 application they are building, where they can usually be optimized and tested.

2524 Applications that build SQL statements at run time give powerful flexibility to the user. As seen
2525 in the preceding example, which didn't even support such common operations as WHERE
2526 clauses, ORDER BY clauses, or joins, building SQL statements at run time is vastly more
2527 complex than hard-coding statements. Furthermore, testing such applications is problematic, as
2528 they can build an arbitrary number of SQL statements.

2529 Building SQL statements at run time takes more time than using a hard-coded statement. This is
2530 rarely a problem because the time the application spends building SQL statements is generally
2531 small compared to the time the user spends entering criteria.

8.1.3 SQL Statements Entered by the User

Applications that perform *ad hoc* analysis often let the user enter SQL statements directly. For example:

```
SQLCHAR    *Statement, SqlState[6], Msg[SQL_MAX_MESSAGE_LENGTH - 1];
SQLINTEGER i, NativeError, MsgLen;
SQLRETURN  rc1, rc2;

// Allocate memory for Statement.  Code not shown.
// Prompt user for SQL statement.
GetSQLStatement(Statement);

// Execute the statement directly. Because it will only be executed
// 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. It is sufficiently hard to write a graphical user interface that adequately exposes the intricacies of SQL that simply asking the user to enter the SQL statement text may be a preferable alternative. However, it requires the user to know not only SQL but 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.

8.2 Interoperability of SQL Statements

SQL statements must obey some grammar, portable or vendor-specific, in order to be acceptable to any data source. The choice of using portable or vendor-specific SQL statements depends on the type of application. Custom applications are less likely to use portable SQL because they usually exploit the capabilities of one or two data sources. Generic applications use portable SQL in order to work with a variety of data sources.

- **Portable SQL**

  The ISO SQL standard defines a standard SQL language and specifies various levels of compliance. The X/Open **SQL** specification follows the standard closely but includes some other features based on their presence in the marketplace. FIPS 127-2 is a U.S. Government procurement specification for SQL databases. The X/Open **SQL** specification aligns with the Transitional Level defined in FIPS 127-2.

- **Proprietary SQL**

  Virtually every data source vendor defines its own grammar, some parts of which are non-standard. If the application takes advantage of proprietary SQL grammar, it can exploit vendor-specific features not available using portable SQL.

- **Effects of XDBC on the SQL language**

  X/Open intends that compliance to XDBC be independent of compliance to SQL. However, there are interdependencies; the compliance policy is defined in Section 1.5.2 on page 9.

  XDBC includes one aspect that directly affects the grammar of SQL statements: It defines *escape clauses* containing standard grammar for commonly available language features, such as a large number of scalar functions, that aren't specified in the X/Open **SQL** specification. See Section 8.3 on page 84.

  Implementations scan SQL statements for escape clauses and perform text substitution to produce text that is acceptable to the data source. (This can be disabled by setting the SQL_ATTR_NOSCAN statement attribute.) The implementation need not parse SQL statements. When an implementation encounters grammar it doesn't recognize, it assumes the grammar is data-source-specific and passes the SQL statement without modification to the data source for execution.

  XDBC also defines escape clauses for language features, such as outer join, that have recently been incorporated in the X/Open **SQL** specification, but for which implementations have diverged. If the application codes these escape clauses, even in preference to the syntax specified in the X/Open **SQL** specification, then XDBC-compliant implementations guarantee to translate the escape clauses to the data-source-specific SQL text.

Thus, portable applications should use X/Open SQL with XDBC escape clauses. Custom applications can use this or a proprietary SQL.

If the application includes escape clauses in its SQL statements, it can determine how the implementation modifies them using the optional *SQLNativeSql*() function. This is often useful when debugging applications. *SQLNativeSql*() accepts an SQL statement and returns it after the implementation has modified it.

8.2.1 Constructing Interoperable SQL Statements

Even portable applications that elect to use X/Open SQL grammar may need to use a feature, such as outer joins, that isn't supported by all data sources. The application writer must decide which language features are required and which are optional. The application can respond to the failure of a particular data source to support a feature that it requires by simply refusing to run with that data source; or through a work-around, such as disabling parts of the interface that let the user select the feature.

The application can call *SQLGetInfo*() to determine support for various features of SQL, and can call *SQLGetTypeInfo*() for information about the data types supported.

The following sections list considerations when building interoperable SQL statements.

Catalog and Schema Usage

Data sources don't necessarily support catalog and schema names as object name qualifiers 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 among data sources. For example, in an Xbase data source, the catalog name is a directory; there is no schema name; the table is an operating-system file; and the catalog name is usually separated from the table name by a backslash (\). The following figure illustrates this condition.

```
<--Catalog Name-> <--Table-->
\XBASE\SALES\CORP\PARTS.DBF
                 ↑
       Catalog Separator (\)
```

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

```
   Sales.Corporate.Parts
     ↑  ↑    ↑       ↑
Catalog | Schema   Table
        |
   Catalog Separator (.)
```

2636 In an Oracle data source, the catalog is also the database, but follows the table name and is
2637 separated from the schema and table names by an at sign (@).

```
2638 Corporate.Parts@Sales
2639     ↑       ↑  ↑   ↑
2640  Schema  Table | Catalog
2641                |
2642      Catalog Separator (@)
```

2643 To determine the catalog separator and the location of the catalog name, an application calls
2644 *SQLGetInfo*() with the SQL_QUALIFIER_NAME_SEPARATOR and
2645 SQL_QUALIFIER_LOCATION options. Interoperable applications should build identifiers
2646 according to these values.

2647 When quoting identifiers that contain more than one part, applications must be careful to quote
2648 each part separately and not quote the character that separates the identifiers. For example, the
2649 following statement to select all of the rows and columns of an Xbase table quotes the catalog
2650 (\XBASE\SALES\CORP) and table (PARTS.DBF) names, but not the catalog separator (\):

```
2651 SELECT * FROM '\XBASE\SALES\CORP'\'PARTS.DBF'
```

2652 The following statement to select all of the rows and columns of an Oracle table quotes the
2653 catalog (Sales), schema (Corporate), and table (Parts) names, but not the catalog (@) or schema (.)
2654 separators:

```
2655 SELECT * FROM 'Corporate'.'Parts'@'Sales'
```

2656 **Quoted Identifiers**

2657 In an SQL statement, identifiers containing special characters or reserved keywords must be
2658 enclosed in *identifier quote characters*; these identifiers are known as *quoted identifiers*. For example,
2659 the Accounts Payable identifier is quoted in the following SELECT statement:

```
2660 SELECT * FROM 'Accounts Payable'
```

2661 Quoting identifiers makes the statement parseable. For example, if Accounts Payable weren't
2662 quoted in the previous statement, the parser would assume there were two tables, Accounts and
2663 Payable, and return a syntax error that they weren't separated by a comma. The identifier quote
2664 character is implementation-specific and is retrieved with the
2665 SQL_IDENTIFIER_QUOTE_CHAR option in *SQLGetInfo*(). The lists of special characters and of
2666 keywords are retrieved with the SQL_SPECIAL_CHARACTERS and SQL_KEYWORDS options
2667 in *SQLGetInfo*().

2668 To be safe, interoperable applications often quote all identifiers except those for pseudo-
2669 columns. *SQLSpecialColumns*() returns a list of pseudo-columns.

2670 **Identifier Case**

2671 In SQL statements and catalog function arguments, identifiers and quoted identifiers can be
2672 either case sensitive or case insensitive. An application determines which they are by calling
2673 *SQLGetInfo*() with the SQL_IDENTIFIER_CASE and SQL_QUOTED_IDENTIFIER_CASE
2674 options.

2675 Each of these options has four possible return values: one stating that the identifier or quoted
2676 identifier case is sensitive and three stating that it is insensitive. The three case insensitive values
2677 further describe the case in which identifiers are stored in the system catalog. How identifiers are
2678 stored in the system catalog is relevant only for display purposes, such as when an application
2679 displays the results of a catalog function; it doesn't change the case sensitivity of identifiers.

2680 **Literal Prefixes and Suffixes**

2681 In an SQL statement, a *literal* is a character representation of an actual data value. For example,
2682 in the following statement, ABC, FFFF, and 10 are literals:

```
2683 SELECT CharCol, BinaryCol, IntegerCol FROM MyTable
2684    WHERE CharCol = 'ABC' AND BinaryCol = 0xFFFF AND IntegerCol = 10
```

2685 Literals for some data types require special prefixes and suffixes. In the preceding example, the
2686 character literal (ABC) requires a single quotation mark (') as both a prefix and a suffix, the
2687 binary literal (FFFF) requires the characters 0x as a prefix but no suffix, and the integer literal (10)
2688 doesn't require a prefix or suffix.

2689 For all data types except date, time, and timestamps, interoperable applications should use the
2690 values returned in the LITERAL_PREFIX and LITERAL_SUFFIX columns in the result set
2691 created by *SQLGetTypeInfo*(). For date, time, timestamp, and date/time interval literals,
2692 interoperable applications should use the escape clauses discussed in the previous section.

2693 **Parameter Markers in Procedure Calls**

2694 When calling procedures that accept parameters, interoperable applications should use
2695 parameter markers instead of literal parameter values. Some data sources don't support the use
2696 of literal parameter values in procedure calls. For more information about parameters, see
2697 Section 9.4 on page 102. For more information about calling procedures, see Section 8.3.6 on
2698 page 88.

2699 **DDL Statements**

2700 Data Definition Language (DDL) statements vary among data sources. X/Open SQL defines
2701 statements for the most common data definition operations: creating and dropping tables,
2702 indexes, and views, altering tables, and granting and revoking privileges. Other data-source-
2703 specific DDL operations are best left to the proprietary database administration software
2704 shipped with most data sources.

2705 In addition, data type names also vary among data sources. Rather than defining standard data
2706 type names and forcing implementations to convert them to data-source-specific names,
2707 *SQLGetTypeInfo*() lets applications determine data-source-specific data type names.
2708 Interoperable applications should use these names in SQL statements to create and alter tables.

8.3 Escape Clauses

A number of language features, such as outer joins and scalar function calls, are commonly implemented by data sources. However, the syntax for these features tend to be data-source-specific, even where the X/Open **SQL** specification defines standard syntax. Because of this, XDBC defines escape clauses that contain standard syntaxes for the following language features:

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

Because the implementation translates escape clauses to data-source-specific syntax, an application can use either the escape clause or data-source-specific syntax. However, use of the escape clause promotes portability.

When using the escape clause, applications must not set the SQL_ATTR_NOSCAN statement attribute, which directs the implementation to send SQL text directly to the data source. Sending XDBC escape clauses to the data source usually causes a syntax error.

Implementations only support those escape clauses that they can map to underlying language features. For example, if the data source doesn't support outer joins, neither does the implementation. To determine which escape clauses are supported, an application calls *SQLGetTypeInfo*() and *SQLGetInfo*().

Syntax

An escape clause consists of an *extension* to standard SQL enclosed within braces:[12]

```
{extension}
```

8.3.1 Date, Time and Timestamp Literals

The X/Open **SQL** specification specifies the format of date, time, and timestamp literals compatibly with the ISO SQL standard. An application can determine if the implementation supports this format for literals by calling *SQLGetInfo*() with the SQL_ANSI_SQL_DATETIME_LITERALS option.

The XDBC escape clause for date, time and timestamp literals is:

```
{literal-type 'value'}
```

where *literal-type* is one of the following:

---

12. **SQL-language escape clauses.** Section 7.2 of the X/Open **SQL** specification defines a separate escape clause to be used for extensions to SQL. Two forms are defined, one for extensions by future formal standards, and a *vendor-escape-clause* for vendor extensions, which has the following format:

```
--(* VENDOR(vendor-name), PRODUCT(product-name) extension *)--
```

Some implementations may accept the *vendor-escape-clause*, where *vendor-name* and *product-name* are specified by the vendor, and where *extension* is one of the same extensions to SQL documented in this specification.

Use of a *vendor-escape-clause* raises the same potential portability problems as use of other vendor-specific SQL syntax. Moreover, for an unrecognised XDBC escape clause, *extension* is passed to the data source without change, whereas the X/Open **SQL** specification specifies that the SQL implementation "conceptually deletes the entire" unrecognized escape clause.

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

To determine if an implementation supports the XDBC escape clauses for date, time, timestamp, or date/time interval literals, an application calls *SQLGetTypeInfo*(). If the data source supports a date, time, timestamp, or interval data type, it must also support the corresponding escape clause.

Examples

Each of the following SQL statements updates the open date of sales order 1023 in the Orders table. The first statement uses standard syntax from the X/Open **SQL** specification. The second statement uses an XDBC escape clause. The third statement uses proprietary syntax for a certain data source and is not portable.

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

The escape clause for a date, time, or timestamp literal can also 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.
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);
```

2790 ### 8.3.2 Interval Literals

2791 The X/Open **SQL** specification specifies the format of interval literals compatibly with the ISO
2792 SQL standard. An application can determine if the implementation supports this format for
2793 literals by calling *SQLGetInfo*() with the SQL_ANSI_SQL_DATETIME_LITERALS option.

2794 The XDBC escape clause for interval literals is:

2795 {*interval-literal'*}

2796 where *interval-literal* is exactly the interval literal defined in the ISO SQL standard. The first
2797 syntactic element within an *interval-literal* is always the word INTERVAL; this distinguishes this
2798 escape clause from other escape clauses.

2799 Here is an example of an XDBC escape clause for an interval literal representing minus five
2800 hours:

```
2801 {INTERVAL – '5:00:00' HOUR TO SECOND}
```

2802 To determine if an implementation supports the XDBC escape clauses for date, time, timestamp,
2803 or date/time interval literals, an application calls *SQLGetTypeInfo*(). If the data source supports
2804 an interval data type, it must also support the corresponding escape clause.

2805 ### 8.3.3 Scalar Function Calls

2806 Scalar functions return a value for each row. For example, the absolute value scalar function
2807 takes a numeric column as an argument and returns the absolute value of each value in the
2808 column. The escape clause for calling a scalar function is:

2809 {fn *scalar-function*}

2810 where scalar-function is one of the functions listed in Appendix F. For example, the following
2811 SQL statements create the same result set of upper-case customer names. The second statement
2812 uses proprietary syntax and is not portable:

```
2813 SELECT {fn UCASE(Name)} FROM Customers
2814 SELECT uppercase(Name) FROM Customers
```

2815 It is valid but not portable for an application to mix calls uses of proprietary syntax and XDBC
2816 escape clauses in the same SQL statement.

2817 Appendix F contains more details and indicates how an application determines which scalar
2818 functions the data source supports.

2819 ### 8.3.4 LIKE Predicate Escape Character

2820 In a LIKE predicate, the percent character (%) matches zero or more of any character and the
2821 underscore character (_) matches any one character. To match an actual percent or underscore
2822 characters in a LIKE predicate, an escape character must precede the percent or underscore
2823 character.

2824 Standard syntax for outer joins is defined in the ISO SQL standard. The X/Open **SQL**
2825 specification presents the same syntax. Applications can determine if the data source supports
2826 standard outer join syntax by calling *SQLGetInfo*() with the SQL_ANSI_SQL_CONFORMANCE
2827 option. If it discloses any compliance (Entry level or above) then the standard LIKE...ESCAPE
2828 clause is available.

2829 The escape clause that defines the LIKE predicate escape character is:

2830 {escape '*escape-character*'}

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

For example, the following SQL statements create the same result set of customer names that start with the characters '%AAA'. The second statement uses proprietary syntax and is not portable. (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.

8.3.5 Outer Joins

Standard syntax for outer joins is defined in the ISO SQL standard. The X/Open **SQL** specification presents the same syntax. Applications can determine if the data source supports standard outer join syntax by calling *SQLGetInfo*() with the SQL_SQL92_RELATIONAL_JOIN_OPERATORS option (and testing the SQL_SRJO_FULL_OUTER_JOIN, SQL_SRJO_LEFT_OUTER_JOIN, and SQL_SRJO_RIGHT_OUTER_JOIN bitmasks).

The XDBC escape clause 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
```

and *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 any WHERE clause.

For example, the following SQL statements create the same result set that lists all customers and shows which has open orders. The second statement uses proprietary syntax and is not portable.

```
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 whether the data source supports outer joins and the implementation supports the outer join escape clause, an application calls *SQLGetInfo*() with the SQL_OUTER_JOIN option. To determine the types of outer joins a data source and implementation support, an application calls *SQLGetInfo*() with the SQL_OJ_CAPABILITIES option. 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 don't 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 XDBC comparison operator.

2872 ### 8.3.6 Procedure Calls

2873 A *procedure* is an executable object stored on the data source (see Section 9.3.3 on page 97). There
2874 is not yet standard syntax for calling a procedure. Applications can determine whether the
2875 implementation supports procedure calls and the XDBC escape clause defined below by calling
2876 *SQLGetInfo*() with the SQL_PROCEDURES option.

2877 The escape clause for calling a procedure is:

```
2878 {[?=] call procedure-name[([parameter][,...])]}
```

2879 where *procedure-name* specifies the name of a procedure and *parameter* specifies a procedure
2880 parameter.

2881 **Procedure Parameters**

2882 A procedure can have zero or more parameters. Each parameter can be an *input parameter* (used
2883 only to supply a value from the calling application to the procedure), *output parameter* (used only
2884 to return a value from the procedure to the calling application), or an *input/output parameter*
2885 (capable of both uses).

2886 A procedure can also return a value, as indicated by the optional parameter marker `?=` at the
2887 start of the syntax. The return value mechanism provides the same capabilities as an output
2888 parameter. The writer of a procedure should disclose to its callers information on the number
2889 and meaning of parameters when calling the procedure.

2890 The application must use a dynamic parameter marker for the procedure's output parameters.
2891 Portable applications should code a dynamic parameter marker for each procedure parameter.
2892 Parameter markers must be bound with *SQLBindParameter*() before the procedure call statement
2893 is executed.

2894 Some SQL implementations at the data source allow the following coding options:

2895 • The use of a literal as an input or input/output procedure parameter

2896 • The omission of an input or input/output procedure parameter. When omitting a parameter,
2897 the comma separating it from other parameters must still appear. The procedure uses the
2898 default value of the parameter. A portable way to direct the procedure to use the default
2899 value of an input or input/output parameter is to set the associated length/indicator buffer
2900 to SQL_DEFAULT_PARAM.

2901 All XDBC implementations accept these options in the XDBC escape clause for procedure calls
2902 and translate them into appropriate SQL syntax for the data source. If these options are invalid
2903 at the data source, either the XDBC implementation or the data source may issue diagnostics.

2904 If a procedure includes parentheses with nothing between them, it implies a single, omitted
2905 parameter. This optional syntax is not valid in some SQL implementations. To call a procedure
2906 that does not accept parameters, omit the parentheses.

2907 If an input/output parameter is omitted or if a literal is supplied for the parameter, the
2908 implementation discards the output value. Similarly, if the parameter marker for the return
2909 value of a procedure is omitted, the implementation discards the return value. Finally, if an
2910 application specifies a return value parameter for a procedure that doesn't return a value, the
2911 implementation sets the value of the length/indicator buffer bound to the parameter to
2912 SQL_NULL_DATA.

2913 **Example**

2914 Suppose the procedure PARTS_IN_ORDERS creates a result set containing a list of orders which
2915 contain a particular part number. The following code calls this procedure for part number 544:

```
2916 SQLUINTEGER PartID;
2917 SQLINTEGER  PartIDInd = 0;

2918 // Bind the parameter.
2919 SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_SLONG,
2920                  SQL_INTEGER, 0, 0, &PartID, 0, PartIDInd);

2921 // Place the department number in PartID.
2922 PartID = 544;

2923 // Execute the statement.
2924 SQLExecDirect(hstmt, "{call PARTS_IN_ORDERS(?)}", SQL_NTS);
```

2925 To determine if a data source supports procedures, an application calls *SQLGetInfo*() with the
2926 SQL_PROCEDURES option.

2927 *Notes to Reviewers*

2928 *This section with side shading will not appear in the final copy. - Ed.*

2929 We need to update this section to discuss named parameters. This information will be added to
2930 the ODBC specification.

2931 *Chapter 9*

2932 Executing Statements

2933 XDBC applications perform almost all of their database access by executing SQL statements. The
2934 general sequence of events is to allocate a statement handle, set any statement attributes, execute
2935 the statement, retrieve any results, and free the statement handle.

2936
9.1 Allocating a Statement Handle

2937 Before the application can execute a statement, it must allocate a statement handle. To do this,
2938 The application declares a variable of type HSTMT. It then calls *SQLAllocHandle*() with the
2939 address of this variable, the handle of the connection in which to allocate the statement, and the
2940 SQL_HANDLE_STMT option. For example:

```
2941 SQLHSTMT hstmt1;
2942 SQLAllocHandle(SQL_HANDLE_STMT, hdbc1, &hstmt1);
```

2943 The statement handle identifies which statement to use when calling XDBC functions. For more
2944 information about statement handles, see Section 4.1.3 on page 36.

2945 ## 9.2 Statement Attributes

2946 Statement attributes are characteristics of the statement. For example, whether to use
2947 bookmarks and what kind of cursor to use with the statement's result set are statement
2948 attributes.

2949 Statement attributes are set with *SQLSetStmtAttr*() and their current settings retrieved with
2950 *SQLGetStmtAttr*(). Statement attributes can also be set with *SQLSetConnectAttr*(); this applies
2951 the new value to all statements on a connection and makes it the default for any new statements.
2952 There is no requirement that an application set any statement attributes; all statement attributes
2953 have defaults, some of which are implementation-defined.

2954 When a statement attribute can be set depends on the attribute itself. The
2955 SQL_ATTR_CONCURRENCY, SQL_ATTR_CURSOR_TYPE, SQL_ATTR_SIMULATE_CURSOR,
2956 and SQL_ATTR_USE_BOOKMARKS statement attributes must be set before the statement is
2957 executed. The SQL_ATTR_ASYNC_ENABLE and SQL_ATTR_NOSCAN statement attributes
2958 can be set at any time, but are not applied until the statement is used again.
2959 SQL_ATTR_MAX_LENGTH, SQL_ATTR_MAX_ROWS, and SQL_ATTR_QUERY_TIMEOUT
2960 statement attributes can be set at any time, but it's implementation-defined whether they are
2961 applied before the statement is used again. The remaining statement attributes can be set at any
2962 time.

2963 For more information, see *SQLSetStmtAttr*().

2964 ### 9.2.1 Temporary Changes to Statement Attribute Value

2965 When an application calls *SQLExecDirect*(), *SQLExecute*(), *SQLGetTypeInfo*(), or *SQLPrepare*() it is
2966 possible that the current values of the following statement attributes are incompatible with the
2967 capabilities of the implementation or the data source:

2968 SQL_ATTR_CONCURRENCY
2969 SQL_ATTR_CURSOR_TYPE
2970 SQL_ATTR_KEYSET_SIZE
2971 SQL_ATTR_MAX_LENGTH
2972 SQL_ATTR_MAX_ROWS
2973 SQL_ATTR_QUERY_TIMEOUT
2974 SQL_ATTR_SIMULATE_CURSOR

2975 Thus the SQL statement could not be executed or other operations specified by the XDBC
2976 function could not be completed with the specified statement attributes.

2977 Under implementation-defined criteria, the implementation may temporarily substitute a value
2978 for one or more of these statement attributes. In this case, the XDBC function succeeds, returns
2979 SQL_SUCCESS_WITH_INFO, and sets SQLSTATE to 01S02 (Attribute value changed). The
2980 application can call *SQLGetStmtAttr*() for the attributes listed above to obtain the current value
2981 and thereby determine what changes the implementation made.

2982 The substitute value is valid for the statement handle until the first of the following occurs:

2983 • The cursor is closed by any means.

2984 • *SQLMoreResults*() is called on the statement handle.

2985 • *SQLCloseCursor*() is called on the statement handle.

2986 At this point, the statement attribute reverts to its previous value.

9.3 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**

  One or more SQL statements are compiled at some time before the application executes and are stored on the data source as a procedure. The provider of the data source may provide built-in procedures. The procedure is executed one or more times at run time. The application can call *SQLProcedures*() to determine what procedures are available for execution.

- **Catalog functions**

  The application calls a catalog function of XSQL. This function conceptually executes a predefined SQL statement, or calls a procedure created for this purpose, which returns a result set. The function is executed one or more times at run time.

A particular statement (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 repeatedly executed 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.
```

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

```
SQLHSTMT    hstmt1;
SQLUINTEGER 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(hstmt, 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);
}
```

On many implementations, allocating statements is costly, 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 Section 10.5 on page 137.

Reusing statements also forces the application to avoid a limitation in some implementations of the number of statements that can be active at one time. A statement is active if it has been prepared or has been executed and still has results available. For example, after an INSERT statement has been prepared, it's generally considered to be active; after a SELECT statement has been executed and the cursor is still open, it's generally considered to be active; after a CREATE TABLE statement has been executed, it's not generally considered to be active.

An application determines how many statements can be active at one time by calling *SQLGetInfo*() with the SQL_MAX_CONCURRENT_ACTIVITIES option. Applications should observe this limit and limit concurrent activities by executing statements in sequence rather than concurrently. Another option is to open multiple connections to the data source, but opening and maintaining multiple connections is relatively costly.

Applications can limit the amount of time allotted for a statement to execute with the SQL_ATTR_QUERY_TIMEOUT statement attribute. Setting a timeout provides that the statement fails if it does not complete by the end of this interval. It returns diagnostic information that indicates that the nature of the failure was a timeout. By default, there is no timeout.

9.3.1 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 is most suited to statements executed a single time. *SQLExecDirect*() should not be used to execute the same statement repeatedly because it will prepare the statement again, which is unnecessary. The application cannot retrieve information about any result set created by the statement 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:

- Sets the values of any parameters. For more information, see Section 9.4 on page 102.
- Calls *SQLExecDirect*() and passes it a string containing the SQL statement.

When *SQLExecDirect*() is called, the implementation:

- Performs text substitutions for any escape clauses (see Section 8.3 on page 84).
- Retrieves the current parameter values and converts them as necessary. For more information, see Section 9.4 on page 102.
- Sends the statement and converted parameter values to the data source for execution.

- Returns diagnostic information (see Chapter 15.

9.3.2 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.

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;
SQLUINTEGER 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 may also reduce network traffic if the data source lets the implementation send it an access plan identifier each time the statement is executed, rather than an entire SQL statement.

Prepared execution shouldn't be used for statements executed a single time. For such statements, it's slightly slower than direct execution because it requires an additional XDBC function call.

Transaction completion may have side-effects on cursors and on access plans of prepared statements. See Section 14.1.3 on page 184.

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. (This can also be done before Step 1.) For more information, see Section 9.4 on page 102.

   The application can also retrieve the metadata (see Section 10.2 on page 127) for the result set now. (However, see **Performance Note** on page 279.)
3. Calls *SQLExecute*() and does any additional processing that is necessary, such as fetching data.
4. Repeats steps 2 and 3 as necessary.

When *SQLPrepare*() is called, the implementation:

- Performs text substitutions for any escape clauses (see Section 8.3 on page 84).

- Sends the statement to the data source for preparation. The data source compiles the statement and prepares an access plan.
- Stores the returned access plan identifier for later execution (if the preparation succeeded) or returns diagnostic information (if the preparation failed). (See Chapter 15).

  Note: Some implementations defer the reporting of errors until a catalog function is called or until the statement is executed. Thus, *SQLPrepare*() might appear to have succeeded when in fact it has failed.

When *SQLExecute*() is called, the implementation:

- Retrieves the current parameter values and converts them as necessary. For more information, see Section 9.4 on page 102.
- Sends the access plan identifier and converted parameter values to the data source.
- Returns diagnostic errors (see Chapter 15).

A data source need not support statement preparation. The data source might accept the SQL statement at *SQLPrepare*() but take no other action until *SQLExecute*(). If the data source supports syntax checking without execution, the implementation might submit the statement for checking when *SQLPrepare*() is called and submit the statement for execution when *SQLExecute*() is called.

If the implementation 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 isn't perfect, *SQLExecute*() can return any errors normally returned by *SQLPrepare*().

9.3.3 Procedures

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

Procedures can be invoked using input parameters and output parameters.

The term *procedure* in this specification encompasses procedures that can return a single value that is used in the syntactic context in which the procedure appears. This type of procedure is also known as a *function*.

When to Use Procedures

The advantages to using procedures are 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, compilation occurs in advance and only execution occurs when the application runs.

- **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.

- **Data-source-specific SQL**

  Procedures provide a way for applications to exploit data-source-specific SQL and still remain interoperable. For example, a procedure on a data source that supports control-of-flow statements in SQL might trap and recover from errors, while a procedure on a data source that doesn't 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 data-source-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 specialized programmers: Application programmers write user interface code and database programmers write procedures.

Procedures are generally used by vertical and custom applications. These applications tend to perform fixed tasks and it's 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 shouldn't 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 data-source-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 data-source-specific SQL statements.

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

The second disadvantage is that many data sources don't support procedures. Again, this is most likely to be a problem for vertical applications designed to run with a number of data sources. 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 X/Open SQL doesn't define a standard grammar for creating procedures. Thus, although applications can call procedures interoperably, they cannot create them interoperably.

Executing Procedures

XDBC defines a standard escape clause for executing procedures in Section 8.3.6 on page 88.

To execute a procedure, an application:

- Sets the values of any parameters. For more information, see Section 9.4 on page 102.
- Calls *SQLExecDirect*() and passes it a string containing the SQL statement that executes the procedure. This statement can use the escape clause defined by XDBC or data-source-specific syntax; statements that use data-source-specific syntax aren't interoperable.

When *SQLExecDirect*() is called, the implementation:

- Retrieves the current parameter values and converts them as necessary. For more information, see Section 9.4 on page 102.
- Calls the procedure in the data source and sends it the converted parameter values.
- 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 implementation returns any errors.

9.3.4 Batches of SQL Statements

A batch of SQL statements is a sequence of two or more SQL statements or a single SQL statement that has the same effect as such a sequence. An entire batch is submitted together for execution. This is often more efficient than submitting statements separately, as network traffic can often be reduced and the data source can sometimes optimize execution of a batch. Batches take the following forms:

- **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)
```

- **Procedures**

  A procedure that contains more than one SQL statement is a batch.

- **Arrays of Parameters**

  Arrays of parameters can be used with a parameterized SQL statement as an effective way to perform bulk operations. For example, arrays of parameters can be used with the following INSERT statement to insert multiple rows into the Lines table while only executing a single SQL statement:

  ```
  INSERT INTO Lines (OrderID, Line, PartID, Quantity)
     VALUES (?, ?, ?, ?)
  ```

  If a data source doesn't support arrays of parameters, the implementation can emulate them by executing the SQL statement once for each set of parameters. For more information, see Section 9.4 on page 102 and Section 9.4.5 on page 109.

Results of a Batch

The *result* of an SQL statement includes the following information:

- A result set, for certain SQL statements such as SELECT.
- A row count, for certain SQL statements such as UPDATE and DELETE.

The term *batch* as used in this specification refers only to batches of result-generating statements.

When different types of batch are nested, the method of retrieving results is undefined. For example, after executing an explicit batch that includes procedure calls, an explicit batch that uses arrays of parameters, or a procedure call that uses arrays of parameters, the method of retrieving result set and row count is undefined.

Executing Batches

Before an application executes a batch of statements, it should first check the level of support. To do this, the application calls *SQLGetInfo*() with the following options:

- The SQL_BATCH_SUPPORT option indicates whether row count and result set generating statements are supported in explicit batches and procedures.
- The SQL_PARAM_ARRAY_ROW_COUNTS and SQL_PARAM_ARRAY_SELECTS options indicate how these statements behave with arrays of parameters.

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 Section 11.3 on page 156.

If a batch of statements includes parameter markers, these are numbered from left to right 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 Section 9.4 on page 102.

Errors

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, it is implementation-defined whether they return SQL_SUCCESS or SQL_SUCCESS_WITH_INFO. In any case, further error information can be retrieved with *SQLGetDiagField*() or *SQLGetDiagRec*(). However, the nature and depth of this information is data-source-specific. Furthermore, this information is unlikely to exactly identify the statement in error.

Retrieving Results from a Batch

Section 11.3 on page 156 describes how to determine the implementation's level of support for returning multiple results, and describes calling *SQLMoreResults*() to discard the results of one SQL statement and move to the results of the next statement.

It is implementation-defined which of the following is true:

- The entire batch is executed as a unit (to the extent possible; see **Errors** above) and then results are made available.

- The batch is executed up to the point at which it produces results, then control is returned to the application, which can retrieve and process results. When the application calls *SQLMoreResults*() to indicate readiness to receive the next result, execution of the batch continues up to the point at which it produces more results; this process repeats until the batch is completed.

9.3.5 Executing Catalog Functions

Calling a catalog functions is similar in effect to executing an SQL statement that generates a result set. (Catalog functions are often implemented by executing predefined SQL statements.) The rules for retrieving result sets apply equally 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.

For more information about catalog functions, see Chapter 7.

9.4 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's 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 isn't 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 implementation 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 aren't 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 implementation 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, as 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. Refer, in the X/Open **SQL** specification, to the explanation of the '**42**000' diagnostic for the PREPARE statement.

9.4.1 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 implementation. It also describes the parameter itself — SQL data type, precision, and so on. The implementation 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 doesn't 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 aren't freed until after they are unbound. For more information, see Section 4.3.2 on page 39.

Because parameter bindings are just information stored in the structure maintained by the implementation 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

3386 then executes the following SQL statement:

```
3387 INSERT INTO Parts (PartID, Description, Price) VALUES (?, ?, ?)
```

3388 If the application then immediately executes the SQL statement:

```
3389 SELECT * FROM Orders WHERE OrderID = ?, OpenDate = ?, Status = ?
```

3390 on the same statement handle, the parameter bindings for the INSERT statement are used
3391 because those are the bindings stored in the statement structure. In most cases, this is a poor
3392 programming practice and should be avoided. Instead, the application should call
3393 *SQLFreeStmt*() with the SQL_RESET_PARAMS option to unbind all the old parameters and then
3394 bind new ones.

3395 Using SQLBindParameter()

3396 The application binds parameters by calling *SQLBindParameter*(). *SQLBindParameter*() binds one
3397 parameter at a time. With it, the application specifies:

- 3398 The parameter number. Parameters are numbered from left to right in the SQL statement,
3399 starting with the number 1. While it's legal to specify a parameter number that is higher than
3400 there are parameters in the SQL statement, the parameter value is ignored when the
3401 statement is executed.
- 3402 The parameter type (input, input/output, or output). Except for parameters in procedure
3403 calls, all parameters are input parameters. For more information, see Section 9.4.4 on page
3404 109.
- 3405 The C data type, address, and octet length of the variable bound to the parameter. The
3406 implementation must be able to convert the data from the C data type to the SQL data type
3407 or an error is returned. For a list of supported conversions, see Appendix D.
- 3408 The SQL data type, precision, and scale of the parameter itself.
- 3409 The address of a length/indicator buffer. It provides the octet length of binary or character
3410 data, specifies that the data is NULL, or specifies that the data will be sent with
3411 *SQLPutData*().

3412 For example, the following code binds *SalesPerson* and *CustID* to parameters for the SalesPerson
3413 and CustID columns. Because SalesPerson contains character data, which is variable length, the
3414 code specifies the octet length of SalesPerson (11) and binds *SalesPersonLenOrInd* to contain the
3415 octet length of the data in SalesPerson. This information isn't necessary for CustID because it
3416 contains integer data, which is of fixed length.

```
3417 SQLCHAR     SalesPerson[11];
3418 SQLINTEGER  SalesPersonLenOrInd, CustIDInd;
3419 SQLUINTEGER CustID;

3420 // Bind SalesPerson to the parameter for the SalesPerson column and
3421 // CustID to the parameter for the CustID column.
3422 SQLBindParameter(hstmt1, 1, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR, 10, 0,
3423                  SalesPerson, sizeof(SalesPerson), &SalesPersonLenOrInd);
3424 SQLBindParameter(hstmt1, 2, SQL_PARAM_INPUT, SQL_C_ULONG, SQL_INTEGER, 10, 0,
3425                  &CustID, 0, &CustIDInd);

3426 // Set the values of the salesperson and customer ID variables and length/indicators.
3427 strcpy(SalesPerson, 'Garcia');
3428 SalesPersonLenOrInd = SQL_NTS;
3429 CustID = 1331;
3430 CustIDInd = 0;

3431 // Execute a statement to get data for all orders made to the specified
3432 // customer by the specified salesperson.
3433 SQLExecDirect(hstmt1,'SELECT * FROM Orders WHERE SalesPerson=? AND CustID=?',SQL_NTS);
```

3434 When *SQLBindParameter*() is called, the implementation associates this information with the
3435 statement. When the statement is executed, it uses the information to retrieve the parameter data
3436 and send it to the data source.

3437 **Describing Parameters**

3438 *SQLBindParameter*() has arguments that describe the parameter: its SQL type, precision, and
3439 scale. The implementation uses this information, or *metadata*, to convert the parameter value to
3440 the type needed by the data source. At first glance, it might seem that the implementation is in a
3441 better position to know the parameter metadata than the application; after all, the
3442 implementation can easily discover the metadata for a result set column. As it turns out, this
3443 isn't the case. First, most data sources don't provide a way for the implementation to discover
3444 parameter metadata. Second, most applications already know the metadata.

3445 If an SQL statement is hard-coded in the application, then the application writer already knows
3446 the type of each parameter. If an SQL statement is constructed by the application at run time,
3447 the application can determine the metadata as it builds the statement. For example, when the
3448 application constructs the clause

```
3449 WHERE OrderID = ?
```

3450 it can call *SQLColumns*() for the OrderID column.

3451 To determine the parameter metadata is when the user enters a parameterized statement, the
3452 application calls *SQLPrepare*() to prepare the statement, *SQLNumParams*() to determine the
3453 number of parameters, and *SQLDescribeParam*() to describe each parameter.

3454 9.4.2 Setting Parameter Values

3455 To set the value of a parameter, the application sets the value of the variable bound to the
3456 parameter. This can be done at any time before the statement is executed, before or after binding
3457 the variable. The value can be changed without limit. When the statement is executed, the
3458 implementation retrieves the current value of the variable. This is particularly useful when a
3459 prepared statement is executed more than once; the application sets new values for some or all
3460 of the variables each time the statement is executed. For an example of this, see Section 9.3.2 on
3461 page 96.

3462 If a length/indicator buffer was bound in the call to *SQLBindParameter*(), it must be set to one of
3463 the following values before the statement is executed:

3464 • The octet length of the data in the bound variable. The implementation checks this length
3465 only if the variable is character or binary (if *ValueType*is SQL_C_CHAR or SQL_C_BINARY).

3466 • SQL_NTS. The data is a null-terminated string.

3467 • SQL_NULL_DATA.The data value is NULL and the implementation ignores the value of the
3468 bound variable.

3469 • SQL_DATA_AT_EXECor the result of the SQL_LEN_DATA_AT_EXECmacro. The value of
3470 the parameter is to be sent with *SQLPutData*(). For more information, see Section 9.4.3 on
3471 page 105.

3472 The following table shows the values of the bound variable and the length/indicator buffer that
3473 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"	SQL_CHAR	SQL_C_CHAR	ABC\0 [a]	SQL_NTS or 3
10	SQL_INTEGER	SQL_C_SLONG	10	--
10	SQL_INTEGER	SQL_C_CHAR	10\0 [a]	SQL_NTS or 2
1 P.M.	SQL_TYPE_TIME	SQL_C_TYPE_TIME	13,0,0 [b]	--
1 P.M.	SQL_TYPE_TIME	SQL_C_CHAR	{t '13:00:00'} [a,c]	SQL_NTS or 14
NULL	SQL_SMALLINT	SQL_C_SSHORT	--	SQL_NULL_DATA

3482 [a] ''\0'' represents a null terminator. The null terminator is required only if the value in the
3483 length/indicator buffer is SQL_NTS.

3484 [b] The numbers in this list are the numbers stored in the fields of the TIME_STRUCT structure.

3485 [c] The string uses the XDBC date escape clause. For more information, see Section 8.3.1 on
3486 page 84.

3487 [d] Implementations must always check this value to see if it's a special value such as
3488 SQL_NULL_DATA.

3489 What an implementation does with a parameter value at execution time is implementation-
3490 defined. If necessary, the implementation converts the value from the C data type and octet
3491 length of the bound variable to the SQL data type, precision, and scale of the parameter. In most
3492 cases, the implementation then sends the value to the data source. In some cases, it formats the
3493 value as text and inserts it into the SQL statement before sending the statement to the data
3494 source.

3495 9.4.3 Sending Long Data

3496 Data sources define *long data* as any character or binary data over a certain size, such as 254
3497 characters. It may be infeasible to store an entire item of long data in memory, such as when the
3498 item represents a long document or a bitmap. Therefore, the data source sends it to the
3499 implementation in parts with *SQLPutData*() when the statement is executed.[13] Parameters for
3500 which data is sent at execution time are known as *data-at-execution parameters*.

3501 Input Parameters

3502 To indicate that a bound input parameter will be a data-at-execute parameter, the application
3503 does the following:

3504 • Sets the OCTET_LENGTH_PTR field in the corresponding record of the application
3505 parameter descriptor to a variable that, at execute time, will contain the value
3506 SQL_DATA_AT_EXEC. This indicates that the data for the parameter will be sent with
3507 *SQLPutData*().

3508 ---

3509 13. 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
3510 *SQLPutData*(). It's much easier to bind the buffer and let the implementation retrieve the data from the buffer.

3511 Alternatively, the application can set this field to the result of the
3512 SQL_LEN_DATA_AT_EXEC(length) macro. This also indicates that the data for the
3513 parameter will be sent with *SQLPutData*(). SQL_LEN_DATA_AT_EXEC(length) is used
3514 when sending long data to a data source that needs to know how many octets of long data
3515 will be sent so that it can preallocate space. To determine if a data source requires this value,
3516 the application calls *SQLGetInfo*() with the SQL_NEED_LONG_DATA_LEN option. All
3517 implementations must support this macro; if the data source doesn't require the octet length,
3518 the implementation can ignore it.

3519 • If there is more than one such field, it sets each DATA_PTR field to some value that it will
3520 recognise as uniquely identifying the field in question. The implementation does not analyze
3521 this value.

3522 (The application can make these settings directly by calling *SQLSetDescField*() or
3523 *SQLSetDescRec*(), or by providing suitable *StrLen_or_Ind* and *ParameterValue* arguments in a call
3524 to *SQLBindParam*()).

3525 When the application calls *SQLExecDirect*() or *SQLExecute*(), if there are any data-at-execute
3526 parameters, the call returns [SQL_NEED_DATA]. The application responds as follows:

3527 1. It calls *SQLParamData*() to advance to the first such parameter. This function returns
3528 [SQL_NEED_DATA] and provides the contents of the DATA_PTR field of the application
3529 parameter descriptor to identify the information required.

3530 2. It calls *SQLPutData*() to pass the actual data for the parameter. Long dynamic arguments
3531 can be sent in pieces by calling *SQLPutData*() repeatedly.

3532 3. It calls *SQLParamData*() again after it has provided the complete dynamic argument.

3533 If more data-at-execute parameters exist, *SQLParamData*() returns [SQL_NEED_DATA]
3534 and the application repeats steps 2 and 3 above.

3535 When no more data-at-execute parameters exist, *SQLParamData*() completes execution of
3536 the SQL statement. The *SQLParamData*() function produces a return value and diagnostics
3537 as the original *SQLExecDirect*() or *SQLExecute*() statement would have produced.

3538 The following flowchart illustrates this technique:

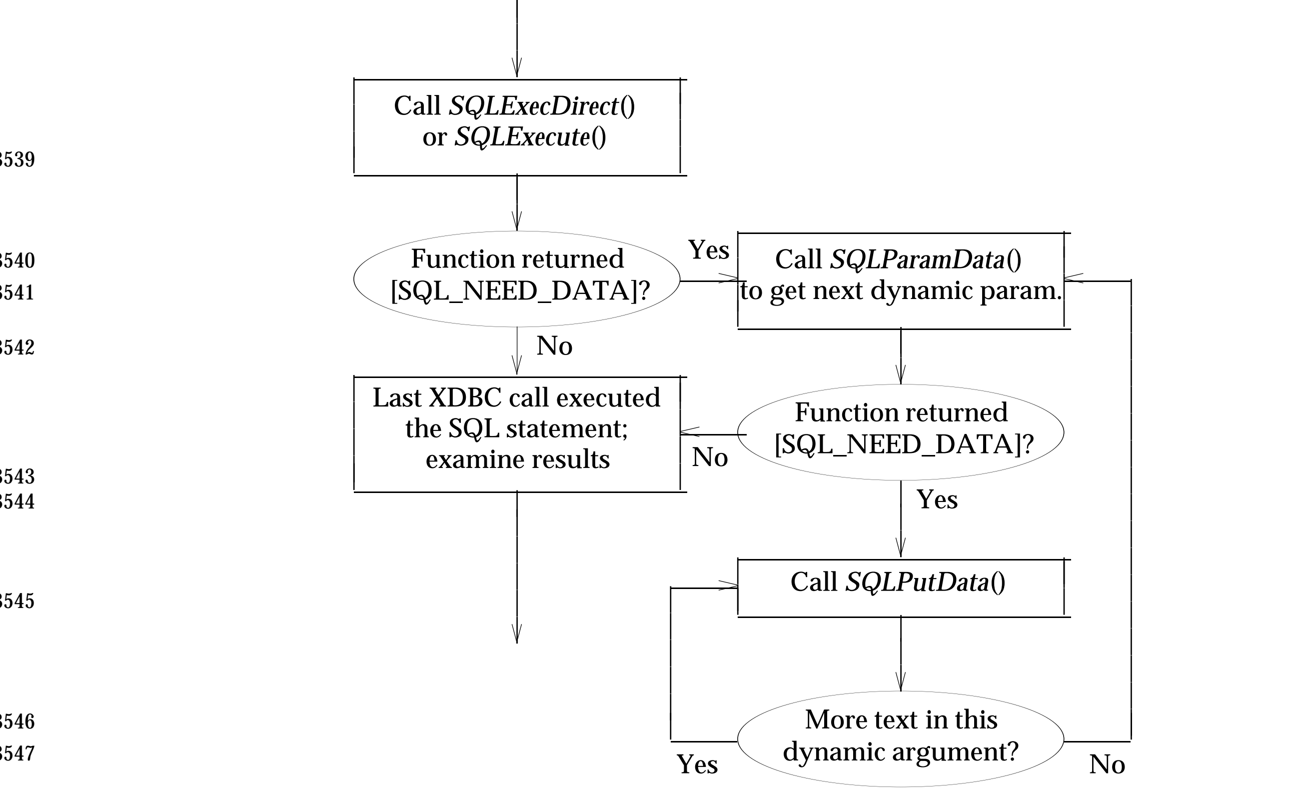


Figure **9-1.** Providing Parameter Data at Execute Time

While the data-at-execute dialogue is in progress, the only XDBC functions the application can call are:

- The *SQLParamData*() and *SQLPutData*() functions, as discussed above
- The *SQLCancel*() function, to cancel the data-at-execute dialogue and force an orderly exit from the loop shown above without executing the SQL statement
- The diagnostic functions.

Moreover, the application cannot end the transaction (see Chapter 14) nor set any connection attribute that would have an impact on the treatment of the statement handle.

Output Parameters

For output dynamic parameters, the application may retrieve arguments in one of the following ways:

- If the application has used an application parameter descriptor to bind parameters, then output values are present in the application variables to which the respective parameters are bound.
- If the parameters are unbound, the application can read argument values by calling *SQLGetData*().
- The application may use both the above techniques if some parameters are bound and some are unbound.

3567 To bind output dynamic parameters to application variables, the application sets the DATA_PTR
3568 field of the corresponding records of the application parameter descriptor. The application sets
3569 other descriptor fields to define the data type, type attributes and variables that will hold
3570 indicator and length information.

3571 If the data type is character-string, the application sets the LENGTH field to the maximum
3572 number of characters of the parameter and sets the OCTET_LENGTH_PTR field to a variable
3573 that, at execute time, will describe the length in octets of the dynamic argument.

3574 The application can call *SQLBindParam*() or *SQLSetDescRec*() to make a complete specification
3575 for an output dynamic parameter, or can call *SQLSetDescField*() to set individual descriptor
3576 fields.

3577 If the application parameter descriptor specifies different data types or type attributes from the
3578 implementation parameter descriptor, the implementation performs type conversion on the
3579 affected parameters when it moves the data values.

3580 **Unbound Output Parameters**

3581 For output dynamic arguments, the application may elect to not bind any of the parameters. In
3582 this instance, it need not reference the parameter descriptor but may obtain parameter data for
3583 these unbound parameters by calling *SQLGetData*().

3584 All dynamic arguments whose mode is IN or INOUT must be bound.

3585 **Bound and Unbound Output Parameters**

3586 If the application is binding some but not all output dynamic arguments, the application must
3587 reference the parameter descriptor for those argument, but need not specify any field of a
3588 descriptor record that pertains to an unbound parameter.

3589 For bound output parameters, after execution of the SQL statement, the implementation
3590 implicitly copies dynamic argument values to the application variables to which the parameters
3591 are bound, and may perform type conversion of the bound parameter data, as described above.

3592 For unbound output parameters following the highest-numbered bound parameter, portable
3593 applications obtain the parameter data by calling *GetData*() in ascending order of parameter
3594 number (from left to right). It is implementation-defined whether an application can obtain
3595 parameter data in a different sequence. It is implementation- defined whether parameter data for
3596 lower-numbered, unbound parameters is available.[14]

3597 The application can achieve type conversion of the parameter data by specifying in the call to
3598 *SQLGetData*() either the desired target type or the value SQL_APD_TYPE, which means that the
3599 application parameter descriptor indicates the desired target type even though the parameter is
3600 unbound.

3601 For bound output parameters, the application uses its knowledge of the parameters to allocate
3602 the maximum memory the value could occupy, in order to avoid truncation of the value. For
3603 unbound output parameters, the value can be arbitrarily long. If the length of the parameter
3604 value exceeds the length of the application's buffer, a feature of *SQLGetData*() lets the
3605 application use repeated calls to obtain the value of a single parameter of CHAR or VARCHAR
3606 type in pieces of manageable size.

3607 ---

3608 14. The application can call *SQLGetInfo*() with SQL_GETPARAM_EXTENSIONS to determine whether the implementation supports this capability.

9.4.4 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(?, ?, ?)}
```

Input/output parameters are used both to send values to procedures and retrieve values from procedures.

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(?, ?)}
```

Implementations handle values for input and input/output parameters in procedures no differently from 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, implementations store the returned values of input/output and output parameters in the variables bound to those parameters. Note that these aren't guaranteed to be set until after all results returned by the procedure have been fetched.

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.

9.4.5 Arrays of Parameter Values

It's 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. This provides the following advantages:

- If the data source supports parameter arrays, network traffic is reduced, as data for many statements is sent in a single packet.
- Some data sources can execute SQL statements using arrays faster than executing the same number of separate SQL statements.
- When the data is stored in an array, as 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.

On a data source that does not support parameter arrays, an implementation can emulate parameter arrays by executing an SQL statement once for each set of parameter values. This could lead to speed increases since the implementation may be able to prepare this SQL statement only once. It might also produce 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 is equivalent to the way that column-wise binding is used for column data, in which all data for a single column is bound using a single data structure.
- 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 is equivalent to the way that row-wise binding is used for column data, in which a structure is defined for each row of column data.

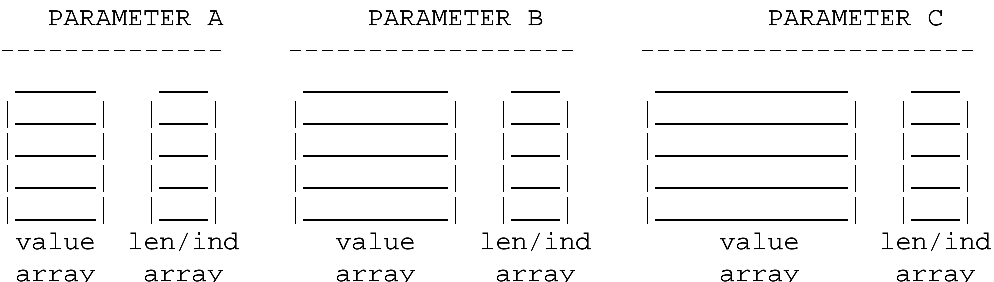
Column-wise binding is the default binding style for arrays of parameters.

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 or row-wise binding. Column-wise binding is the default binding style for arrays of parameters. 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.

The implementation executes the SQL statement multiple times, each time using values from successive rows of each array. The following diagram shows how column-wise binding works.



For example, the following code binds 10-element arrays to the OrderID, SalesPerson, and Status columns.

```
SQLCHAR       NameArray[10], PhoneArray[10];
SQLUINTEGER   AgeArray[10];
SQLINTEGER    NameLenOrIndArray[10], PhoneLenOrIndArray[10];
SQLUSMALLINT  i;
SQLRETURN     rc;

// Set the SQL_ATTR_PARAM_BIND_TYPE statement attribute to use column-wise binding.
SQLSetStmtAttr(hstmt, SQL_ATTR_PARAM_BIND_TYPE, SQL_BIND_BY_COLUMN, 0);

// Specify the number of elements in each parameter array.
SQLSetStmtAttr(hstmt, SQL_ATTR_PARAMSET_SIZE, 10, SQL_IS_NOT_POINTER);
```

```
3697 // Specify the address of a variable in which to return the array row number in case of an error.
3698 SQLSetStmtAttr(hstmt, SQL_ATTR_PARAMS_PROCESSED_ARRAY, ValuePtr, SQL_IS_POINTER);
3699 // Prepare a statement to insert data into the Employee table.
3700 rc = SQLPrepare(StatementHandle, 'INSERT INTO EMPLOYEE (NAME, AGE, PHONE) VALUES (?, ?, ?)', SQL_NTS);
3701 If (rc == SQL_SUCCESS) {
3702   // Bind arrays to the Name, Age, and Phone parameters.
3703   SQLBindParameter(StatementHandle, 1, SQL_C_CHAR, NameArray, sizeof(NameArray[0],
3704              NameLenOrIndArray);
3705   SQLBindParameter(StatementHandle, 2, SQL_C_UINTEGER, AgeArray, 0, AgeIndArray);
3706   SQLBindParameter(StatementHandle, 3, SQL_C_CHAR, PhoneArray, sizeof(PhoneArray[0]),
3707              PhoneLenOrIndArray);
3708 // Set the value of each element of the Name, Age, and Phone arrays.
3709   // Execute the SQL statement 10 times to insert arrays of parameters into the table.
3710   // Code to check if rc equals SQL_SUCCESS_WITH_INFO or SQL_ERROR
3711   // not shown.
3712   while ((rc = SQLExecute(StatementHandle)) != SQL_ERROR) {
3713      for (i = 0; i < 10; i++) {
3714         if ((RowStatusArray[i] == SQL_ROW_SUCCESS) ||
3715             (RowStatusArray[i] == SQL_ROW_SUCCESS_WITH_INFO)) {
3716            if (OrderIDIndArray[i] == SQL_NULL_DATA) printf(' NULL      ')
3717            else printf('%d', OrderIDArray[i]);
3718            if (SalesPersonLenOrIndArray[i] == SQL_NULL_DATA) printf(' NULL       ')
3719            else printf('%s', SalesPersonArray[i]);
3720            if (StatusLenOrIndArray[i] == SQL_NULL_DATA) printf(' NULL\n')
3721            else printf('%s\n', StatusArray[i]);
3722         }
3723   }
3724 }
```

3725 **Row-wise Binding**

3726 When using row-wise binding, an application defines a structure for each set of parameters. The
3727 structure contains one or two elements for each parameter for which data is to be provided. The
3728 first element holds the parameter value and the second element holds the length/indicator
3729 buffer. The application then allocates an array of these structures, which contains as many
3730 elements as there are values for each parameter.

3731 The application binds the addresses of the parameters in the first structure of the array. Thus,
3732 the implementation can calculate the address of the data for a particular row and column as:

```
3733 Address = Bound Address + ((Row Number - 1) * Structure Size)
```

3734 where rows are numbered from 1 to the size of the parameter set. The following diagram shows
3735 how row-wise binding works. Generally, only parameters that will be bound are included in the
3736 structure. The parameters can be placed in the structure in any order, but are shown in

sequential order for clarity.

```
  PARAM A      PARAM B        PARAM C
----------- ------------- ---------------
 ________________________________________________
|_______|___|__________|___|_____________|___| <--array[0]
 ________________________________________________
|_______|___|__________|___|_____________|___| <--array[1]
 ________________________________________________
|_______|___|__________|___|_____________|___| <--array[2]
 ________________________________________________
|_______|___|__________|___|_____________|___| <--array[3]
   ^      ^       ^       ^       ^        ^
   |      |       |       |       |        |
 value len/ind value len/ind  value   len/ind
element elem. element elem.  element   elem.
```

For example, the following code creates a structure with elements in which to provide data for the Name, Age, and Phone parameters and length/indicators for the Name and Phone columns. It allocates 10 of these structures, defines a ten-element array of these structures, and binds the elements of the first structure in the array to the Name, Age, and Phone parameters.

```
// Define the ORDERINFO struct and allocate an array of 10 structs.
typedef struct {
   SQLUINTEGER OrderID;
   SQLINTEGER  OrderIDInd;
   SQLCHAR     SalesPerson[11];
   SQLINTEGER  SalesPersonLenOrInd;
   SQLCHAR     Status[7];
   SQLINTEGER  StatusLenOrInd;
} ORDERINFO;
ORDERINFO OrderInfoArray[10];

SQLUINTEGER  NumRowsFetched;
SQLUSMALLINT RowStatusArray[10], i;
HRESULT      rc;

// 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 row-set
// 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, 10, 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 10 rows at a time. Print the actual number of rows fetched; this number
// is returned in NumRowsFetched. Check the row status array to only print 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++) {
```

```
3794            if (RowStatusArray[i] == SQL_ROW_SUCCESS||SQL_ROW_SUCCESS_WITH_INFO) {
3795               if (OrderInfoArray[i].OrderIDInd == SQL_NULL_DATA)
3796                  printf(' NULL      ')
3797               else
3798                  printf('%d', OrderInfoArray[i].OrderID);
3799               if (OrderInfoArray[i].SalesPersonLenOrInd == SQL_NULL_DATA)
3800                  printf(' NULL       ')
3801               else
3802                  printf('%s', OrderInfoArray[i].SalesPerson);
3803               if (OrderInfoArray[i].StatusLenOrInd == SQL_NULL_DATA)
3804                  printf(' NULL\n')
3805               else
3806                  printf('%s\n', OrderInfoArray[i].Status);
3807            }
3808         }
3809      }
3810      // Close the cursor.
3811      SQLCloseCursor(hstmt);
```

3812 **Bind Offsets**

3813 An application can specify that an offset be added to buffer addresses bound for parameters.
3814 This offset is added to bound buffer addresses and the corresponding length/indicator pointers
3815 when *SQLExecDirect*() or *SQLExecute*() is called.

3816 Bind offsets let an application change bindings without calling *SQLBindParameter*() for
3817 previously bound columns. A call to *SQLBindParameter*() to rebind data changes the buffer
3818 address and the length/indicator pointer. Rebinding with an offset, on the other hand, simply
3819 adds an offset to the existing buffer address and length/indicator pointer. A new offset can be
3820 specified at any time, and is always added to the originally bound values.

3821 To specify a bind offset, the application sets the SQL_ATTR_PARAM_BIND_OFFSET_PTR
3822 statement attribute to the address of an SQLINTEGER buffer. Before the application calls a
3823 function that uses the bindings, it places an offset in octets in this buffer. When the function is
3824 called, the implementation adds the offset to both the address in the binding and the
3825 length/indicator pointer as long as the address or length/indicator pointer isn't 0 and the bound
3826 parameter is in the SQL statement. The sum of the address and the offset must be a valid
3827 address. (This means that the address to which the offset is added need not be a valid address, if
3828 the offset is a valid address.)

3829 The use of bind offsets is defined only for the case of row-wise bindings. Any bind offset value
3830 is ignored in the case of column-wise bindings.

3831 **Using Arrays of Parameters**

3832 Using parameter arrays differs from using single parameter values in only two ways. First, the
3833 application passes the address of a parameter value array and (if needed) a length/indicator
3834 array to *SQLBindParameter*(), rather than the addresses of single variables. (This is the case when
3835 column-wise binding is used; when row-wise binding is use, the application passes the address
3836 of the parameter (and if needed, the length/indicator) in the first parameter structure in a call to
3837 *SQLBindParameter*().) Second, it calls *SQLSetStmtAttr*() with an Attribute of
3838 SQL_ATTR_PARAMSET_SIZE to specify the number of elements in each parameter array, calls
3839 *SQLSetStmtAttr*() with an Attribute of SQL_ATTR_PARAMS_PROCESSED_ARRAY to specify
3840 the address of a variable in which the implementation can return the array row number in case
3841 of an error, and calls *SQLSetStmtAttr*() with an *Attribute* of SQL_ATTR_PARAM_STATUS_PTR
3842 to point to an array containing status information for each row of parameter values. Note that all
3843 parameter arrays bound to an SQL statement must have the same number of elements. The
3844 implementation stores the array addresses, the count of array elements, and the address of the
3845 row number variable in the structure it maintains for the statement.

Before executing the statement, the application sets the value of each element of each bound array. When the statement is executed, the implementation uses the information it stored to retrieve the parameter values and send them to the data source; if possible, the implementation 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 data sources today. Thus, implementations can implement it by executing SQL statements individually.

Before an application uses arrays of parameters, it must be sure that they are supported by the implementations used by the application. There are two ways to do this:

- Use only connections to data sources over which arrays of parameters are known to be implemented. The application can hard-code parameters for such connections or the user can be instructed to specify only such connections. Custom applications and vertical applications commonly use a limited set of data sources.

- Check for support of arrays of parameters at run time. An implementation 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 application can determine the status of other implementation-defined options, relating to arrays of parameter values, by calling *SQLGetInfo*() with the following options:

- The SQL_PARAM_ARRAY_ROW_COUNTS option indicates whether individual row counts (one for each parameter set) are available (SQL_PARC_BATCH), or rows counts are rolled up into one (SQL_PARC_NO_BATCH).

- 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 implementation does not allow a result-set-generating statement to be executed with an array of parameters, SQL_PARAM_ARRAY_SELECTS returns SQL_PAS_NO_SELECT.

It is undefined if arrays of parameters can be used in other contexts in SQL, but this usage presupposes the use of vendor extensions to X/Open SQL grammar.

Error Processing

If an error occurs while executing the statements, 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 statements except the statement returning the row are executed, or all statements before (but not after) the statement returning the row are executed. The implementation sets SQL_ATTR_PARAMS_PROCESSED_PTR to the number of the row currently being processed. If all statements except the statement returning the row are executed, the implementation sets SQL_ATTR_PARAMS_PROCESSED_PTR 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 executed SQL statement. The parameter status array is allocated by the application and populated by the implementation. Its elements indicate whether the SQL statement was executed successfully for the set of parameters, or whether an error occurred while the statement was executed. If an error is encountered, the implementation sets the corresponding value in the parameter status array to SQL_PARAM_ERROR, continues processing statements, 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.

The application can determine whether the implementation fills in the status array and the parameter status array by calling *SQLGetInfo*() with the SQL_PARAM_ARRAY_ROW_COUNTS and SQL_PARAM_ARRAY_SELECTS options.

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 implementation. If fetched rows are used as input parameters by calling *SQLCopyDesc*(), the values of the row status array can be used in the parameter operation array, provided **#define**s have been set up to map the status values such that successfully fetched values are processed as parameters and unsuccessfully fetched values are ignored.

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 points are notable:

- When the implementation 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 implementation requests parameter values. If an error occurs in the execution of a data-at-execution parameter, it is implementation-defined whether SQL_ATTR_PARAMS_PROCESSED_PTR is reset to the lower row number. (The address of the row number variable is not updated if *SQLExecute*(), *SQLExecDirect*(), or *SQLParamData*() return SQL_STILL_EXECUTING.)
- Since the implementation doesn't interpret the value in *ParameterValuePtr* 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 the implementation is requesting on this occasion, is always the address of the array.

9.5 Asynchronous Execution

This section discusses asynchronous execution.

By default, XDBC functions execute **synchronously**— that is, an XDBC function does not return control to its caller until the requested operation is complete.

However, a function executed in the optional **asynchronous** mode may return promptly to the caller with the return value [SQL_STILL_EXECUTING], indicating that the requested operation is not yet complete. This return value indicates neither success nor failure. A function executed asynchronously may instead return any of the other defined return values and they retain their usual meaning.

The application polls the implementation by periodically calling the same function again. The return value [SQL_STILL_EXECUTING] may recur, indicating that the operation is not yet complete; or the return value may indicate success or failure. The application is free to perform other work between polling calls to the XDBC function. The application can cancel the requested operation by calling *SQLCancel*().

Asynchrony occurs only when all the following are true:

- The implementation provides some level of support, as described in Section 9.5.1 on page 116
- Asynchrony is permitted for the XDBC function in question, as defined by Table 9-1 on page 117
- The application has enabled asynchrony on the relevant connection or statement handle, as described in Section 9.5.2 on page 118.

9.5.1 Levels of Asynchronous Support

Implementations may support asynchronous execution at three levels:

- **No support**

  All XDBC functions execute synchronously; no XDBC function returns to its caller until it can report either success or failure.

- **Connection-level support**

  For every connection handle, either all associated statement handles are enabled for asynchrony or none are. No connection handle has some statement handles in synchronous mode and others in asynchronous mode.

- **Statement-level support**

  Any connection handle can have some associated statement handles that are in asynchronous mode and other statement handles in synchronous mode.

Determining the Support Level

The *SQLGetInfo*() function's SQL_ASYNC_MODE option indicates which level of support for asynchrony the implementation provides. *SQLGetInfo*() returns SQL_AM_CONNECTION if connection-level asynchronous execution is supported, SQL_AM_STATEMENT if statement-level asynchronous execution is supported, or SQL_AM_NONE if the implementation does not support asynchronous execution.

An implementation may limit the number of concurrent asynchronoous statements. The application can determine any limit on a specified connection by calling *SQLGetInfo*() with the SQL_MAX_ASYNC_CONCURRENT_STATEMENTS option.

Functions That Can Execute Asynchronously

Asynchrony is permitted only for functions in the following table. No other XDBC function ever returns [SQL_STILL_EXECUTING].

SQLBulkOperations	*SQLForeignKeys*	*SQLPrepare*
SQLColAttribute	*SQLGetData*	*SQLPrimaryKeys*
SQLColumnPrivileges	*SQLGetDescField**	*SQLProcedureColumns*
SQLColumns	*SQLGetDescRec**	*SQLProcedures*
SQLCopyDesc	*SQLGetDiagField*	*SQLPutData*
SQLDescribeCol	*SQLGetDiagRec*	*SQLSetPos*
SQLDescribeParam	*SQLGetTypeInfo*	*SQLSpecialColumns*
SQLExecDirect	*SQLMoreResults*	*SQLStatistics*
SQLExecute	*SQLNumParams*	*SQLTablePrivileges*
SQLFetch	*SQLNumResultCols*	*SQLTables*
SQLFetchScroll	*SQLParamData*	

Table **9-1.** Functions for which Asynchrony is Permitted

The above functions are those that may either submit requests to, or retrieve data from, the data source; and hence may require extensive processing.

On multithread operating systems, executing functions on separate threads may be a useful alternative to executing them asynchronously on the same thread. The performance effects of using either technique are undefined.

Implementation Methods

An implementation may support asynchrony using any of the following methods:

- **Activity in Parallel**

  The common meaning of the [SQL_STILL_EXECUTING] return value is that the initial call to the XDBC function has initiated activity that will operate in parallel to the calling process (for example, has submitted the request to a server) to complete the requested operation.

- **Time-slicing**

  An acceptable alternative implementation is that each subsequent call to the XDBC function performs another part of the operation originally requested. The function returns [SQL_STILL_EXECUTING] to pass control back to the application with a notification that it needs to gain control one or more times in the future to complete the operation.

- **No effect**

  Even when the above rules permit an XDBC function to use asynchrony, the function may in fact not return to its caller until it can report success or failure. No XDBC function is required to return [SQL_STILL_EXECUTING] in any situation.

---

* These functions can execute asynchronously only if the descriptor is an implementation descriptor, not an application descriptor.

In a situation where the application is expecting asynchrony, the implementation may impair application performance if it provides anything other than parallel execution.[15]

9.5.2 Enabling Asynchrony

When any connection handle or statement handle is allocated, asynchrony is initially disabled on that handle. This means no XDBC function ever returns [SQL_STILL_EXECUTING] unless the application takes explicit action to enable asynchrony.

On all implementations that provide some level of support for asynchrony, the application can enable asynchrony throughout a connection by setting the SQL_ATTR_ASYNC_ENABLE attribute of a connection handle. This enables asynchrony for all of the following:

- The specified connection handle
- All statement handles subsequently associated with that connection handle.

It is implementation-defined whether enabling asynchrony on a connection handle enables asynchrony on statement handles already associated with that connection handle.

There is also a statement attribute named SQL_ATTR_ASYNC_ENABLE. Its use depends on the level of support for asynchrony in the implementation:

- **On implementations that provide connection-level support of asynchrony:**

  The SQL_ATTR_ASYNC_ENABLE attribute of statement handles is a read-only attribute by which an application can determine if asynchrony has been enabled for the connection with which the statement handle is associated.

- **On implementations that provide statement-level support of asynchrony:**

  The SQL_ATTR_ASYNC_ENABLE attribute of statement handles is a settable attribute. Its initial value when the statement handle is allocated is the value of the SQL_ATTR_ASYNC_ENABLE attribute of the associated connection handle. This indicates whether asynchrony has been enabled or disabled for that connection.

  By setting the SQL_ATTR_ASYNC_ENABLE statement attribute, the application can override that decision for the specific statement handle.

9.5.3 Steps in Asynchronous Execution

Terminology

Of the functions for which asynchrony is defined (see Table 9-1 on page 117), some specify their scope of operation using a connection handle; others use a statement handle. In the remainder of this section, the term *original function* means the combination of an XDBC function and either a connection handle or statement handle (whichever that function takes).

When no incomplete asynchronous operation is outstanding, the first call to the original function that returns [SQL_STILL_EXECUTING] is called the *initial call*. The term *subsequent call* refers to subsequent repeat calls to the original function, until and including the call that does not return [SQL_STILL_EXECUTING] but instead reports success or failure.

---

15. X/Open believes that the degree of any impairment will be measured by the marketplace and regarded as part of the quality of implementation. X/Open declines to specify limits on such impairment or mandate that any specific XDBC function call operate using asynchrony.

Sequence

An application achieves asynchrony by performing the following steps:

1. The application either relies on private information on the level of support for asynchrony in the associated XDBC implementation, or calls *SQLGetInfo*() to determine the level of support.

2. The application enables asynchrony by setting the connection handle attribute (or, on some applications, setting the statement handle attribute). See Section 9.5.2 on page 118.

3. The application makes an applicable *initial call*. (Applicable means it calls an XDBC function for which asynchrony is defined, with a handle on which asynchrony is enabled.)

   If the return value is [SQL_STILL_EXECUTING], the requested operation uses asynchrony. Otherwise, the requested operation is complete and the sequence ends.

4. The application may perform other operations, subject to the restrictions specified in **Restrictions on Operations during Asynchrony** on page 122. In particular, the application may cancel the asynchronous operation, as described in Section 9.5.4 on page 119.

5. The application makes a *subsequent call* to determine whether the operation requested by the *initial call* is complete.

   Although this subsequent call uses the same syntax as the initial call, its only purpose is to refer back to the initial call to poll whether the operation it requested is complete. Therefore, the connection or statement handle on all subsequent calls must match that used on the initial call. However, the implementation ignores all other input arguments.

   For example, suppose an application calls *SQLExecDirect*() to execute a SELECT statement asynchronously. On each *subsequent call* to *SQLExecDirect*(), the return value indicates the status of the SELECT statement, even if the *StatementText* argument then contains an INSERT statement.

   If the subsequent call returns [SQL_STILL_EXECUTING], the operation is still not complete and the application returns to Step 4. Otherwise, the operation is complete.

9.5.4 Cancelling an Asynchronously-executing Function

An application can use *SQLCancel*() to request the cancellation of an asynchronously-executing function. Since, on some implementations, asynchrony may involve parallel processing, the success of this request may be subject to race conditions. To avoid ambiguity, applications should follow precisely the sequence described in this section and illustrated in Figure 9-2 on page 121.

The application can request cancellation at any time that the most recent call to the original function returned [SQL_STILL_EXECUTING].

To cancel an asynchronous operation, the application performs the following steps:

- Call *SQLCancel*(). When and if the original function is cancelled is implementation-dependent.

- Perform *subsequent calls* to the original function until such a call returns a value other than [SQL_STILL_EXECUTING].

  That is, after requesting cancellation, the application must continue polling the original function, as in the normal asynchrony sequence, in order to detect the conclusion of the operation. The effect is undefined if the application makes more than one attempt to cancel the operation.

On the subsequent call that reports completion of the operation, the return value and the diagnostic area indicate the operation's status:

- A return value of [SQL_ERROR] and a SQLSTATE of HY008 (Operation canceled) indicate that the application successfully cancelled the operation.
- Any other return value, or a return value of [SQL_ERROR] and any other SQLSTATE, indicates that the application failed to cancel the operation. The operation is complete. The return value and diagnostics area reflect the outcome of the operation.

9.5.5 Example Asynchronous Control Flow

Figure 9-2 on page 121 illustrates the sequence of asynchronous processing for an XDBC function *f*(), including the method of attempting to cancel an asynchronous operation. The figure does not illustrate the step of detecting the level of support for asynchrony in the implementation, nor the step of enabling asynchrony on the relevant handles.

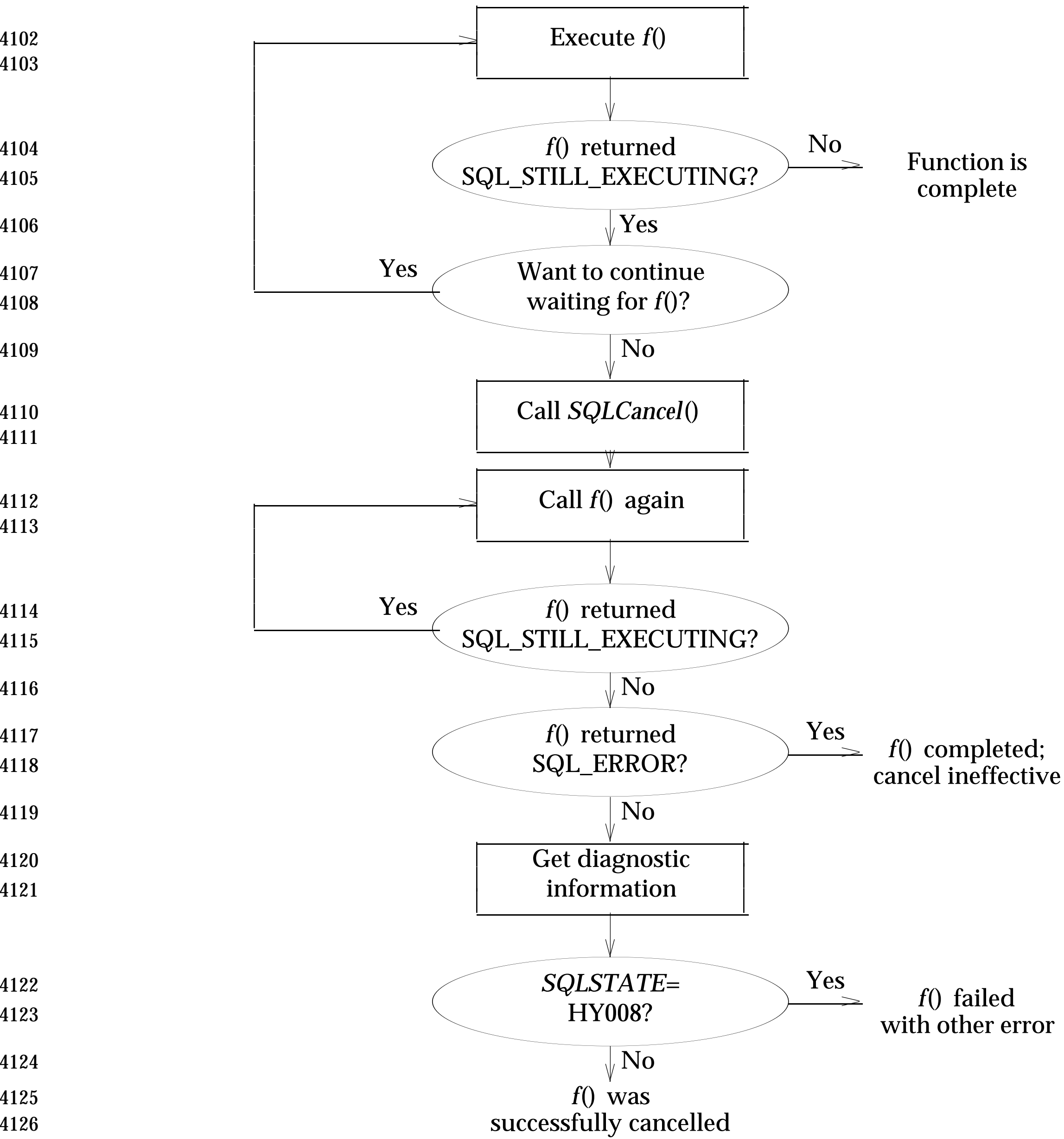


Figure **9-2.** Example Control Flow for Asynchrony

9.5.6 Asynchrony Combined with Other XDBC Features

Data-at-execute Dialogue

Section 9.4.3 on page 105 tells how, at the application's request, *SQLExecDirect*() and *SQLExecute*() may request the values of certain bound parameters at the time the function is executed. A call to one of these functions initiates a sequence of calls to *SQLParamData*() and *SQLPutData*() by which the application provides the values.

The time-consuming portion of *SQLExecDirect*() and *SQLExecute*() (the work for which asynchronous execution may be necessary) is presumed to entirely precede the data-at-execute dialogue. Therefore, a call to *SQLExecDirect*() or *SQLExecute*() can only return [SQL_STILL_EXECUTING] before the start of the data-at-execute dialogue. As usual, completion of the asynchronous operation is indicated by a return value other than [SQL_STILL_EXECUTING], which reflects the status of the original function. If this return value is [SQL_NEED_DATA], then any asynchronous operation is complete and the data-at-execute dialogue begins.

Restrictions on Operations during Asynchrony

When an asynchronous operation is outstanding, the other work the application can initiate is subject to the following limits:

- The only functions the application can call using a statement handle involved in an asynchronous operation are: the original function, *SQLCancel*(), *SQLGetDiagField*() and *SQLGetDiagRec*().
- The only functions the application can call using a connection handle involved in an asynchronous operation are: *SQLAllocHandle*() to allocate a statement handle, *SQLAllocStmt*(), *SQLGetFunctions*(), *SQLGetDiagField*() and *SQLGetDiagRec*().

Calling any other function on these handles returns SQLSTATE HY010 (Function sequence error). The application can call any function using handles other than the original statement handle and the original connection handle.

Diagnostics Area during Asynchrony

When an asynchronous operation is outstanding, the diagnostics area associated with the statement handle has the following contents:

- The SQL_DIAG_RETURNCODE field in the header record contains [SQL_STILL_EXECUTING].
- There are 0 status records.

If the application calls *SQLCancel*() to try to cancel an asynchronous operation and *SQLCancel*() returns [SQL_ERROR], the diagnostics area contains information pertaining to the failed call to *SQLCancel*().

After then *SQLCancel*() attempt, the application is required to make a *subsequent call* to poll the asynchronous operation for completion. After any such call, the diagnostics area contains information pertaining to that call.

Ordering Not Guaranteed

On implementations that support multiple concurrent asynchronous operations, no type of ordering of XDBC operations is guaranteed (either among asynchronous operations or between any asynchronous operation and an operation performed synchronously). It is undefined which operation finishes first.

9.5.7 Limits on Concurrency

Some implementations may impose a numerical limit on the number of active asynchronous operations on a connection. For example, if the limit is 1, an application cannot execute any XDBC function asynchronously on a second statement handle until it has verified that an asynchronous operation on a first statement handle either has completed or has been cancelled successfully.

An application can find out how many concurrent asynchronous operations the implementation allows by calling *SQLGetInfo*() to find the value of SQL_MAXIMUM_ASYNC_CONCURRENT_STATEMENTS.

Any limit on the number of concurrent asynchronous operations is independent of the limit on the number of statement handles that can simultaneously interact with the server. It is also independent of the level of support for asynchrony in the implementation.

If an application initiates an asynchronous operation so as to exceed the implementation's limit on the number of concurrent asynchronous operations on a connection, the effect is implementation-defined.

9.5.8 Example Asynchrony Code

The following is an example of asynchronous execution of an SQL statement:

```
SQLHDBC     hdbc1, hdbc2;
SQLHSTMT    hstmt1, hstmt2, hstmt3;
SQLCHAR     *SQLStatement = 'SELECT * FROM Orders';
SQLUINTEGER 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 S1010 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, SQL, 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
}
```

4211 ## 9.6 Freeing a Statement Handle

4212 As mentioned earlier, it's more efficient to reuse statements than drop them and allocate new
4213 ones. Before executing a new SQL statement on a statement, applications should be sure that the
4214 current statement settings are appropriate. These include statement attributes, parameter
4215 bindings, and result set bindings. Generally, parameters and result sets for the old SQL
4216 statement need to be unbound (by calling *SQLFreeStmt*() with the SQL_RESET_PARAMS and
4217 SQL_UNBIND options) and rebound for the new SQL statement.

4218 When the application has finished using the statement, it calls *SQLFreeHandle*() to free the
4219 statement. After freeing the statement, it's an application programming error to use the
4220 statement's handle in a call to an XDBC function; doing so has undefined but probably fatal
4221 consequences.

4222 When *SQLFreeHandle*() is called, the implementation releases the structure used to store
4223 information about the statement. *SQLDisconnect*() automatically frees all statements on a
4224 connection.

Retrieving Results (Basic)

A *result set* is the conceptual table that the implementation makes available to the application based on a query. SELECT statements, catalog functions, and some procedures create result sets. For example, the first SQL statement below 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 multi-row fetch and scrollable cursors, see Chapter 11. For information about updating, deleting, and inserting rows, see Chapter 12.

4245 ## 10.1 Was a Result Set Created?

4246 In most situations, application programmers know whether or not the statements their
4247 application executes will create a result set. This is the case if the application uses hard-coded
4248 SQL statements written by the programmer. It's usually the case when the application constructs
4249 SQL statements at run time: The programmer can easily include code that flags whether a
4250 SELECT statement or an INSERT statement is being constructed.

4251 In a few situations, the programmer cannot know whether a statement will create a result set.
4252 This is true if the application provides a way for the user to enter and execute an SQL statement.
4253 It's also true when the application constructs a statement at run time to execute a procedure.

4254 In such cases, the application can call *SQLNumResultCols*() to determine the number of columns
4255 in the result set. If this is 0, the statement didn't create a result set; if it's any other number, the
4256 statement did create a result set.

4257 The application can call *SQLNumResultCols*() at any time after the statement is prepared or
4258 executed. (But see **Performance Note** on page 279.)

4259 To determine the number of rows that an SQL statement returns in a result set, the application
4260 may be able to call *SQLRowCount*(). The application can call *SQLGetInfo*() to determine the
4261 meaning of the row count, as described in **Detecting Cursor Capabilities with SQLGetInfo()** on
4262 page 402:

4263 • The SQL_CA2_CRC_EXACT bitmask indicates that the row count is exact. The
4264 SQL_CA2_CRC_APPROXIMATE bitmask indicates that the row count is approximate. If
4265 neither bit is set, the data source does not provide a row count at all.

4266 • For static and keyset-driven cursors, the application can determine the effect on the row
4267 count of changes made through *SQLBulkOperations*(), *SQLSetPos*(), or by positioned UPDATE
4268 or DELETE statements.

4269 ## 10.2 Result Set Metadata

4270 *Metadata* is data that describes other data. For example, result set metadata describes the result
4271 set, such as the number of columns in the result set, the data types of those columns, their
4272 names, precision, nullability, and so on.

4273 Interoperable applications should check the metadata of the columns of a result set, because this
4274 metadata might not be the same as the metadata for the corresponding column of the underlying
4275 table (if indeed the column is based on a single column of an underlying table). For example, on
4276 some implementations, a column of a result set created by joining two tables is sometimes not
4277 updatable even when the underlying columns are updatable. Even data types cannot be
4278 assumed to be the same, as the data source might promote the data type in creating the result
4279 set.

4280 ### 10.2.1 How is Metadata Used?

4281 Applications require metadata for most result set operations. For example, the application uses
4282 the data type of a column to determine what kind of variable to bind to that column. It uses the
4283 octet length of a character column to determine how much space it needs to display data from
4284 that column. How an application determines the metadata for a column depends on the type of
4285 the application.

4286 Vertical applications work with predefined tables and perform predefined operations on those
4287 tables. Because the result set metadata for such applications is defined before the application is
4288 even written and is controlled by the application developer, it can be hard-coded into the
4289 application. For example, if an order ID column is defined as a 4-octet integer in the data source,
4290 the application can always bind a 4-octet integer to that column. When metadata is hard-coded
4291 in the application, a change to the tables used by the application generally implies a change to
4292 the application code. This is rarely a problem, as such changes are generally made as part of a
4293 new release of the application.

4294 Like vertical applications, custom applications generally work with predefined tables and
4295 perform predefined operations on those tables. For example, an application might be written to
4296 transfer data among three different data sources; the data to be transferred is generally known
4297 when the application is written. Thus, custom applications also tend to have hard-coded
4298 metadata.

4299 Generic applications, especially applications that support ad-hoc queries, almost never know the
4300 metadata of the result sets they create. Therefore, they must discover the metadata at run time
4301 using the functions *SQLNumResultCols*(), *SQLDescribeCols*(), and *SQLColAttribute*(), which are
4302 described in the next section.

4303 All applications, regardless of their type, can hard code metadata for the result sets returned by
4304 the catalog functions. These result sets are defined in the reference section of this manual.

4305 ### 10.2.2 SQLDescribeCol() and SQLColAttribute()

4306 *SQLDescribeCol*() and *SQLColAttribute*() are used to retrieve result set metadata. The difference
4307 between these two functions is that *SQLDescribeCol*() always returns the same five pieces of
4308 information (a column's name, data type, precision, scale, and nullability), while
4309 *SQLColAttribute*() returns a single piece of information requested by the application. However,
4310 *SQLColAttribute*() can return a much richer selection of metadata, including a column's case
4311 sensitivity, display size, updatability, and searchability.

4312 Many applications, especially ones that only display data, only require the metadata returned by
4313 *SQLDescribeCol*(). For these applications, it's faster to use *SQLDescribeCol*() than
4314 *SQLColAttribute*() because the information is returned in a single call. Other applications,
4315 especially ones that update data, require the additional metadata returned by *SQLColAttribute*()

4316 and so use both functions.

4317 An application can retrieve result set metadata at any time after a statement has been prepared
4318 or executed and before the cursor over the result set is closed. (Applications may degrade
4319 performance by asking for metadata before the statement is executed; see **Performance Note** on
4320 page 279.)

4321 It is often costly to retrieve metadata from the data source. Because of this, implementations
4322 should cache any metadata they retrieve from the data source and hold it for as long as the
4323 cursor over the result set is open. Also, applications should request only the metadata they
4324 absolutely need.

4325 ## 10.3 Binding Result Set Columns

4326 Data fetched from the data source is returned to the application in variables that the application
4327 has allocated for this purpose. Before this can be done, the application must associate, or bind,
4328 these variables to the columns of the result set; conceptually, this process is the same as binding
4329 application variables to statement parameters. When the application binds a variable to a result
4330 set column, it describes that variable — address, data type, and so on — to the implementation.
4331 The implementation stores this information in the structure it maintains for that statement and
4332 uses the information to return the value from the column when the row is fetched.

4333 ### 10.3.1 Overview

4334 Applications can bind as many or as few columns of the result set as they choose, including
4335 binding no columns at all. When a row of data is fetched, the implementation returns the data
4336 for the bound columns to the application. Whether the application binds all of the columns in the
4337 result set depends on the application. For example, applications that generate reports usually
4338 have a fixed format; such applications create a result set containing all of the columns used in
4339 the report, then bind and retrieve the data for all of these columns. Applications that display
4340 screens full of data sometimes allow the user to decide which columns to display; such
4341 applications create a result set containing all columns the user might want, but bind and retrieve
4342 the data only for those columns chosen by the user.

4343 Data can be retrieved from unbound columns by calling *SQLGetData*(). This is commonly called
4344 to retrieve long data, which often exceeds the length of a single buffer and must be retrieved in
4345 parts (see Section 10.4.4 on page 135).

4346 Columns can be bound at any time, even after rows have been fetched. However, the new
4347 bindings don't take effect until the next time a row is fetched; they aren't applied to data from
4348 rows already fetched.

4349 A variable remains bound to a column until the application calls *SQLBindCol*() to specify a
4350 different variable to the column (or to specify a null pointer, which unbinds the column). In
4351 addition, all columns are unbound by calling *SQLFreeStmt*() with the SQL_UNBIND option, and
4352 all columns are unbound when the statement is released. The application must ensure that all
4353 bound variables remain valid as long as they are bound. For more information, see Section 4.3.2
4354 on page 39.

4355 Because column bindings are just information associated with the statement structure, they can
4356 be set in any order. They are also independent of the result set. For example, suppose an
4357 application binds the columns of the result set generated by the following SQL statement:

```
4358 SELECT * FROM Orders
```

4359 If the application then executes the SQL statement:

```
4360 SELECT * FROM Lines
```

4361 on the same statement handle, the column bindings for the first result set are still associated with
4362 the statement structure. In most cases, this is a poor programming practice and should be
4363 avoided. Instead, the application should call *SQLFreeStmt*() with the SQL_UNBIND option to
4364 unbind all the old columns and then bind new ones.

4365 ### 10.3.2 Using SQLBindCol()

4366 The application binds columns by calling *SQLBindCol*(). This function binds one column at a
4367 time. With it, the application specifies:

4368 • The column number. Column 0 is the bookmark column; this column isn't included in some
4369 result sets. All other columns are numbered starting with the number 1. It's an error to bind a
4370 higher numbered column than there are columns in the result set; this error cannot be
4371 detected until the result set has been created, so it's returned by *SQLFetch*(), not
4372 *SQLBindCol*().

4373 • The C data type, address, and octet length of the variable bound to the column. It's an error
4374 to specify a C data type to which the SQL data type of the column cannot be converted; this
4375 error might not be detected until the result set has been created, so it's returned by
4376 *SQLFetch*(), not *SQLBindCol*(). For a list of supported conversions, see Appendix D. For
4377 information about the octet length, see **Data Buffer Length** on page 42.

4378 • The address of a length/indicator buffer. The length/indicator buffer is optional. It's used to
4379 return the octet length of binary or character data or return SQL_NULL_DATA if the data is
4380 NULL. For more information, see Section 4.3.5 on page 42.

4381 When *SQLBindCol*() is called, the implementation associates this information with the statement.
4382 When each row of data is fetched, it uses the information to place the data for each column in the
4383 bound application variables.

4384 For example, the following code binds variables to the SalesPerson and CustID columns. Data
4385 for the columns will be returned in SalesPerson and CustID. Because SalesPerson is a character
4386 buffer, the application specifies its octet length (11) so the implementation can determine
4387 whether to truncate the data. The octet length of the returned title, or whether it's NULL, will be
4388 returned in SalesPersonLenOrInd.

4389 Because CustID is an integer variable and has fixed length, there is no need to specify its octet
4390 length; the implementation assumes it's **sizeof(SQLUINTEGER)**. The octet length of the
4391 returned customer ID data, or whether it's NULL, will be returned in CustIDInd. Note that the
4392 application is only interested in whether the salary is NULL, as the octet length is always
4393 **sizeof(SQLUINTEGER)**.

```
4394 SQLCHAR     SalesPerson[11];
4395 SQLUINTEGER CustID;
4396 SQLINTEGER  SalesPersonLenOrInd, CustIDInd;
4397 SQLRETURN   rc;
4398 // Bind SalesPerson to the SalesPerson column and CustID to the CustID column.
4399 SQLBindCol(hstmt, 1, SQL_C_CHAR, SalesPerson, sizeof(SalesPerson),
4400            &SalesPersonLenOrInd);
4401 SQLBindCol(hstmt, 2, SQL_C_FLOAT, &CustID, 0, &CustIDInd);
4402 // Execute a statement to get the sales person/customer of all orders.
4403 SQLExecDirect(hstmt, 'SELECT SalesPerson, CustID FROM Orders ORDER BY SalesPerson',
4404               SQL_NTS);
4405 // Fetch and print the data.  Print 'NULL' if the data is NULL.  Code to check if rc
4406 // equals SQL_ERROR or SQL_SUCCESS_WITH_INFO not shown.
4407 while ((rc = SQLFetch(hstmt)) != SQL_NO_DATA) {
4408    if (SalesPersonLenOrInd == SQL_NULL_DATA) printf('NULL        ')
4409    else printf('%10s  ', SalesPerson);
4410    if (CustIDInd == SQL_NULL_DATA) printf('NULL\n')
4411    else printf('%d\n', CustID);
4412 }
4413 // Close the cursor.
4414 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 implementation's point of view, the address of this memory is indistinguishable from the address of a variable bound in the previous example.

```
// 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 SQLINTEGER 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;

// 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,
// octet length, and buffer offset to the data.
SQLNumResultCols(hstmt1, &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, i + 1, SQL_DESC_TYPE, NULL, 0, NULL, (SQLPOINTER) &SQLType);
   CTypeArray[i] = GetDefaultCType(SQLType);

   // Determine the column's octet length. Calculate the offset in the buffer to the
   // data as the offset to the previous column, plus the octet length of the previous
   // column, plus the octet length of the previous column's length/indicator buffer.
   // Note that the octet 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 octet 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, i+1, SQL_DESC_OCTET_LENGTH, NULL, 0, NULL, &ColLenArray[i]);
   ColLen[i]Array = ALIGNBUF(ColLenArray[i]);
   if (i)
      OffsetArray[i] = OffsetArray[i-1]+ColLenArray[i-1]+ALIGNBUF(sizeof(SQLINTEGER));
}
```

```
4475        // Allocate the data buffer. The size of the buffer is equal to the offset to the data
4476        // buffer for the final column, plus the octet length of the data buffer and
4477        // length/indicator buffer for the last column.
4478        void *DataPtr = malloc(OffsetArray[NumCols - 1] +
4479                            ColLenArray[NumCols - 1] + ALIGNBUF(sizeof(SQLINTEGER)));
4480        // For each column, bind the address in the buffer at the start of the memory allocated
4481        // for that column's data and the address at the start of the memory allocated for that
4482        // column's length/indicator buffer.
4483        for (i = 0; i < NumCols; i++)
4484           SQLBindCol(hstmt,
4485                      i + 1,
4486                      CTypeArray[i],
4487                      (SQLPOINTER)((SQLCHAR *)DataPtr + OffsetArray[i]),
4488                      ColLenArray[i],
4489                      (SQLINTEGER *)((SQLCHAR *)DataPtr + OffsetArray[i] + ColLenArray[i]));
4490        // Retrieve and print each row. PrintData accepts a pointer to the data, its C type,
4491        // and its octet length/indicator. It contains a switch statement that casts and prints
4492        // the data according to its type. Code to check if rc equals SQL_ERROR or
4493        // SQL_SUCCESS_WITH_INFO not shown.
4494        while ((rc = SQLFetch(hstmt)) != SQL_NO_DATA) {
4495           for (i = 0; i < NumCols; i++) {
4496              PrintData(DataPtr[OffsetArray[i]], CTypeArray[i],
4497                        (SQLINTEGER)*DataPtr[OffsetArray[i] + ColLenArray[i]]);
4498           }
4499        }
4500        // Close the cursor.
4501        SQLCloseCursor(hstmt);
```

4502 ## 10.4 Fetching Data

4503 The process of retrieving rows from the result set and returning them to the application is called
4504 *fetching*. This section describes how to fetch data.

4505 ### 10.4.1 Cursors

4506 A cursor is a movable pointer into a result set. The cursor indicates the current position in the
4507 result set.

4508 Cursors in XDBC are based on the cursor model in X/Open SQL. One notable difference
4509 between these models is the way cursors are opened. In SQL, the application must explicitly
4510 declare and open a cursor before using it. When the XDBC implementation executes a statement
4511 that creates a result set, it implicitly opens a cursor and positions it before the first row of the
4512 result set. In both SQL and XDBC, a cursor must be closed after the application has finished
4513 using it.

4514 The remainder of this chapter discusses the default cursor type of XDBC: the *forward-only cursor*
4515 when used to fetch one row of data at a time. A forward-only cursor can only move forward
4516 through the result set. To return to a previous row, the application must close and reopen the
4517 cursor, then read rows from the beginning of the result set until it reaches the row. Forward-only
4518 cursors provide a fast way to make a single pass through a result set.

4519 Forward-only cursors are less useful for screen-based applications in which the user scrolls
4520 backward and forward through the data. Advanced cursor types are discussed in Chapter 11.

4521 **Transaction completion may have side-effects on cursors. See Section 14.1.3 on page 184.**

4522 ### 10.4.2 Fetching a Row of Data

4523 To fetch a row of data in the forward direction, an application calls *SQLFetch*(). *SQLFetch*()
4524 advances the cursor to the next row and returns the data for any columns that were bound with
4525 calls to *SQLBindCol*(). When the cursor reaches the end of the result set, *SQLFetch*() returns
4526 SQL_NO_DATA. The examples in Section 10.3.2 on page 130 show the use of *SQLFetch*().

4527 *SQLFetch*() retrieves the data for any bound columns from the data source (or returns an error if
4528 it cannot), converts it according to the types of the bound variables, and places the converted
4529 data in those variables. The application can continue fetching rows, but the data for the current
4530 row is lost. For unbound columns, the implementation may retrieve and discard it or not retrieve
4531 it at all.

4532 The implementation also sets the values of any length/indicator buffers that have been bound. If
4533 the data value for a column is NULL, the implementation sets the corresponding
4534 length/indicator buffer to SQL_NULL_DATA. If the data value isn't NULL, the implementation
4535 sets the length/indicator buffer to the octet length of the data after conversion. If this length
4536 cannot be determined, as is sometimes the case when long data is retrieved in pieces, the
4537 implementation sets the length/indicator buffer to SQL_NO_TOTAL. (For fixed-length data
4538 types, such as integers and date structures, the octet length is the size of the data type.)

4539 For variable-length data, such as character and binary data, the implementation checks the octet
4540 length of the converted data against the octet length of the buffer bound to the column; the
4541 buffer's length is specified in the *BufferLength* argument in *SQLBindCol*(). If the octet length of
4542 the converted data is greater than the octet length of the buffer, the implementation truncates the
4543 data to fit in the buffer, returns the untruncated length in the length/indicator buffer, returns
4544 SQL_SUCCESS_WITH_INFO, and places SQLSTATE01004 (Data truncated) in the diagnostics.

4545 Values of fixed-length application data types is never truncated; the implementation assumes
4546 that the size of the bound buffer is the size of the data type. The application can avoid truncation
4547 by determining the data length from the metadata and binding the column to a buffer of

4548 adequate length. However, the application might explicitly bind a buffer it knows to be too
4549 small, such as to retrieve and display just the start of a long text column.

4550 If the SQL_ATTR_OUTPUT_NTS environment attribute is SQL_TRUE, then the implementation
4551 null-terminates character data before returning it to the application, even if the implementation
4552 truncated it. The null terminator isn't included in the returned octet length, but does require
4553 space in the bound buffer. For example, suppose an application uses a character set in which
4554 each character occupies one octet, an implementation has 50 characters of data to return, and the
4555 application's buffer is 25 octets long. In the application's buffer, the implementation returns the
4556 first 24 characters followed by a null terminator. In the length/indicator buffer, it returns a octet
4557 length of 50.

4558 The application can restrict the number of rows in the result set by setting the
4559 SQL_ATTR_MAX_ROWS statement attribute before executing the statement that creates the
4560 result set. For example, the preview mode in an application used to format reports only needs
4561 enough data to display the first page of the report. By restricting the size of the result set, such a
4562 feature would run faster. This statement attribute is intended to reduce network traffic and
4563 might not be supported by all implementations.

4564 10.4.3 Row Status

4565 The application can set the SQL_ATTR_ROW_STATUS_PTR to the address of an application
4566 variable. In this case, after a fetch, the implementation places one of the following values into
4567 the application variable:

4568 SQL_ROW_SUCCESS
4569 The row was successfully fetched and has not changed since it was last fetched.

4570 SQL_ROW_SUCCESS_WITH_INFO
4571 The row was successfully fetched and has not changed since it was last fetched. However, a
4572 warning was returned about the row.

4573 SQL_ROW_ERROR
4574 An error occurred while fetching the row.

4575 SQL_ROW_UPDATED
4576 The row was successfully fetched and has been updated since it was last fetched. If the row
4577 is fetched again, or refreshed by *SQLSetPos*(), its status is changed to the new status.

4578 SQL_ROW_DELETED
4579 The row has been deleted since it was last fetched.

4580 SQL_ROW_ADDED
4581 The row was inserted by *SQLBulkOperations*(). If the row is fetched again, its status is
4582 SQL_ROW_SUCCESS.

4583 SQL_ROW_NOROW
4584 The row-set overlapped the end of the result set and no row was returned that
4585 corresponded to this element of the row status array.

4586 This same information is available after a fetch in the SQL_DESC_ARRAY_STATUS_PTRfield of
4587 the implementation row descriptor.

10.4.4 Getting Long Data

Section 9.4.3 on page 105 discussed cases in which a *long data* value must be sent to the data source in pieces. The application can likewise retrieve a long data value in parts by calling *SQLGetData*() after fetching the other data in the row.

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's much easier to bind a buffer to the column and let the implementation 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 octet 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 implementation 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*() only lasts 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 octet length of the data in the length/indicator buffer. For more information on how *SQLFetch*() returns data, see Section 10.4.2 on page 133.

SQLGetData() differs from *SQLFetch*() in one important respect. If it's called more than once in succession for the same column, each call returns a successive part of the data. 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 value returned in the length/indicator buffer decreases in each call by the number of octets returned in the previous call. (If the implementation cannot determine the amount of available data, it returns an octet length of SQL_NO_TOTAL.) For example:

```
// Declare a binary buffer to retrieve 5000 octets of data at a time.
SQLCHAR     BinaryPtr[5000];
SQLUINTEGER PartID;
SQLINTEGER  PartIDInd, BinaryLenOrInd, NumOctets;
SQLRETURN   rc;

// 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 octets in
   // each part to a function that displays it. The number of octets is always 5000 if
   // there were more than 5000 octets 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) {
      NumOctets = (BinaryLenOrInd > 5000) || (BinaryLenOrInd == SQL_NO_TOTAL) ?
                  5000 : BinaryLenOrInd;
      DisplayNextPictPart(BinaryPtr, NumOctets);
   }
```

```
4641        }
4642        // Close the cursor.
4643        SQLCloseCursor(hstmt);
```

4644 Portable applications should either assume the implementation enforces the following
4645 restrictions on access to columns using *SQLGetData*(), or should call *SQLGetInfo*() with the
4646 SQL_GETDATA_EXTENSIONS option to determine if the current data source enforces the
4647 restrictions:

4648 • Columns must be accessed in order of increasing column number. For example, it's an error
4649 to call *SQLGetData*() for column 5 and then call it for column 4.

4650 • *SQLGetData*() cannot be used to gain access to bound columns.

4651 • The column requested must have a higher column number than the last bound column. For
4652 example, if the last bound column is column 3, it's an error to call *SQLGetData*() for column 2.
4653 For this reason, applications should place long data columns at the end of the select list.

4654 • *SQLGetData*() cannot be used if a multi-row fetch was performed (see Section 11.1.3 on page
4655 145).

4656 If the application doesn't need all of the data in a character or binary data column, it can reduce
4657 network traffic in data-source-based implementations by setting the
4658 SQL_ATTR_MAX_LENGTH statement attribute before executing the statement. This restricts
4659 the number of octets of data that will be returned for any character or binary column. For
4660 example, suppose a column contains long text documents. An application that browses the table
4661 containing this column might only need to display the first page of each document. Although
4662 this statement attribute can be simulated in the implementation, there is no reason to do so. In
4663 particular, if an application wants to truncate character or binary data, it should bind a small
4664 buffer to the column with *SQLBindCol*() and let the implementation truncate the data.

4665 ## 10.5 Closing the Cursor

4666 When an application has finished using a cursor, it calls *SQLCloseCursor*() to close the cursor. For
4667 example:

```
4668 SQLCloseCursor(hstmt);
```

4669 Until the application closes the cursor, the statement on which the cursor is opened cannot be
4670 used for most other operations, such as executing another SQL statement. For a complete list of
4671 functions that can be called while a cursor is open, see Appendix B.

4672 Cursors remain open until they are explicitly closed. In particular, reaching the end of the result
4673 set, when *SQLFetch*() returns SQL_NO_DATA, doesn't close a cursor. Even cursors on empty
4674 result sets (result sets created when a statement executed successfully but which returned no
4675 rows) must be explicitly closed.

4676 (On some data sources, completing a transaction implicitly closes all cursors on the connection.
4677 The application can determine whether this is the case by calling *SQLGetInfo*() with the
4678 SQL_CURSOR_COMMIT_BEHAVIOR or SQL_CURSOR_ROLLBACK_BEHAVIOR options.)

4679 *Chapter 11*

4680
Retrieving Results (Advanced)

4681 The simple model of forward-only cursors that retrieve a single row at a time is not optimal in
4682 two broad areas:

4683 • **Multi-row fetch**

4684 Network traffic and certain other overhead can be reduced by requesting more than one row
4685 of a result set with a single fetch. (See Section 11.1 on page 140.)

4686 • **Scrolling**

4687 Some screen-based applications let the user scroll backward and forward through the data.
4688 The use of forward-only cursors prevents the user from scrolling backward. A *scrollable*
4689 *cursor* can move backward and forward in the result set. (See Section 11.2 on page 147.)

4690 (Alternatives to scrollable cursors, such as closing and reopening the cursor, then fetching
4691 forward until the cursor reaches the target row, or cacheing some or all rows and
4692 implementing scrolling in the application, are less advantageous, especially as the size of the
4693 result set increases.)

11.1 Multi-row Fetch

An application declares that each call to *SQLFetch*() and *SQLFetchScroll*() should fetch multiple rows, and that certain operations of *SQLSetPos*() should affect multiple rows, by setting the SQL_ATTR_ROW_ARRAY_SIZE statement attribute (or, equivalently, setting the SQL_DESC_ARRAY_SIZE field of the application row descriptor).

The application always has this option. On data sources from which only one row at a time can be fetched, the XDBC implementation simulates the feature; for example, by translating the application's request for multiple rows into multiple requests from the data source. As more data sources implement multi-row fetch natively, applications written to request multiple rows will get faster.

Terminology

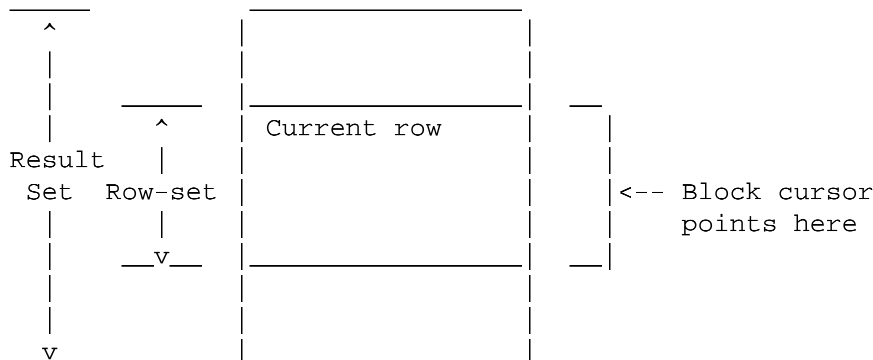
When multiple rows are fetched, this set of fetched rows is called the *row-set*. The result set is maintained at the data source, while the row-set is a movable window into the result set whose data are in application buffers.

To perform XDBC operations that operate on a single row when multiple rows have been fetched, the application must first indicate which row is the *current row*. These single-row operations are calls to *SQLGetData*() and positioned UPDATE and DELETE statements. When a row-set is fetched, the current row is the first row of the row-set. To change the current row, the application calls *SQLSetPos*(). For more information, see Section 11.1.3 on page 145 and Section 12.3 on page 163. Use of multi-row fetch does not require use of a scrollable cursor (see Section 11.2 on page 147). For example, a report generator can perform multi-row fetches to reduce network traffic, but it does not need a scrollable cursor if it can do its work through forward-only access to the result set.

Block Cursor

A multi-row fetch can be described in terms of the cursor and the current row-set size. However, some data sources may implement a multi-row fetch using an operation that treats every row in the current (fetched) row-set as active. The full implications of this are implementation-defined. For example, isolation and cursor sensitivity may be defined based on this row-set. Such effects at the data source make useful the concept of a *block cursor* or *fat cursor* — that the execution of a multi-row fetch has widened the cursor so that it effectively points to all the rows of the current row-set simultaneously.

The block cursor typically points to a row-set. The row-set can be empty, full, or partial. The block cursor can instead be positioned before the start or after the end of the result set. In these cases, or if the result set is empty, the block cursor points to an empty row-set.



Results for Each Row Fetched

For each column in a fetched row, the following information can be returned into variables the application binds:

- The data value.
- The length/indicator information.
- If desired, indicator information can be returned separately from length information.

In a multi-row fetch (that is, whenever the value of SQL_ATTR_ROW_ARRAY_SIZE is greater than 1), each of these pieces of information is bound not to a single variable but to an array that contains as many elements as there are rows in the row-set.

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.

11.1.1 Binding Styles

There are two binding styles:

- **Column-wise Binding**

  The application binds an array to each column.

- **Row-wise Binding**

  The application defines a structure to hold the data for an entire row. It binds an array of these structures.

The application sets the SQL_ATTR_ROW_BIND_TYPE statement attribute to specify whether it is using column-wise or row-wise binding. Row-wise binding often corresponds more closely to the application's layout of data in processor memory.

Column-wise Binding

With column-wise binding, the application binds arrays to hold the information described in **Results for Each Row Fetched** on page 141. The application binds to each column an array of variables instead of a single variable. These arrays are called the row-set buffers. The implementation returns the data for each row in successive rows of each array. The following diagram shows how column-wise binding works.

Column A		Column B		Column C	
Value array	Len/Ind array	Value array	Len/Ind array	Value array	Len/Ind array

Figure **11-1.** Application Buffer for Column-wise Binding

For example, the following code binds 10-element arrays to the OrderID, SalesPerson, and Status columns.

```
SQLUINTEGER  OrderIDArray[10], NumRowsFetched;
SQLCHAR      SalesPersonArray[10][11], StatusArray[10][7];
SQLINTEGER   OrderIDIndArray[10], SalesPersonLenOrIndArray[10],
```

```
                StatusLenOrIndArray[10];
SQLUSMALLINT RowStatusArray[10], i;
SQLRETURN    rc;

// Set the SQL_ATTR_ROW_BIND_TYPE statement attribute to use column-wise binding.
// Declare the row-set 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, 10, 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 10 rows at a time. Print the actual number of rows fetched; this number
// is returned in NumRowsFetched. Check the row status array to only print 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', OrderIDArray[i]);
         if (SalesPersonLenOrIndArray[i] == SQL_NULL_DATA) printf(' NULL       ')
         else printf('%s', 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 fields for the information described in **Results for Each Row Fetched** on page 141, repeated for each column to be fetched. The application then allocates an array of these structures, which contains at least as many elements as there are rows in the row-set.

The application, by setting the SQL_ATTR_ROW_BIND_TYPE statement attribute to a positive value, not only selects row-wise binding but informs the implementation of the length of the application's structure. The address that the application binds as the pointer to the column data is the address of the member that represents that column in the first element of the application's array. Using this information, the implementation can calculate the address of the data for a particular row and column as:

$$Address = Bound\ Address + ((Row\ Number - 1) * Structure\ Size)$$

The above subtraction relates the numbering of rows (which begins with 1) to the numbering of array elements in the C language (which begins with 0).

Since the application binds each column separately, and in this way reports to the implementation the location of column data within the application structure, the columns need

not appear in the structure in sequence, and the structure can also contain fields for the application's own use.

Generally, the application includes in the structure only the columns it will bind. The following diagram illustrates an application array for use with row-wise binding:

Column A		Column B		Column C		
Value elem.	Len/Ind elem.	Value elem.	Len/Ind elem.	Value elem.	Len/Ind elem.	
						← array[0]
						← array[1]
						← array[2]
						← array[3]

Figure **11-2.** Application Buffer for Row-wise Binding

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 the ORDERINFO struct and allocate an array of 10 structs.
typedef struct {
   SQLUINTEGER OrderID;
   SQLINTEGER  OrderIDInd;
   SQLCHAR     SalesPerson[11];
   SQLINTEGER  SalesPersonLenOrInd;
   SQLCHAR     Status[7];
   SQLINTEGER  StatusLenOrInd;
} ORDERINFO;
ORDERINFO OrderInfoArray[10];

SQLUINTEGER  NumRowsFetched;
SQLUSMALLINT RowStatusArray[10], i;
HRESULT      rc;

// 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 row-set 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, 10, 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);
```

```
4879    SQLBindCol(hstmt, 3, SQL_C_CHAR, OrderInfoArray[0].Status,
4880               sizeof(OrderInfoArray[0].Status),
4881               &OrderInfoArray[0].StatusLenOrInd);
4882    // Execute a statement to retrieve rows from the Orders table.
4883    SQLExecDirect(hstmt, 'SELECT OrderID, SalesPerson, Status FROM Orders',
4884                  SQL_NTS);
4885    // Fetch up to 10 rows at a time. Print the actual number of rows
4886    // fetched; this number is returned in NumRowsFetched. Check the row
4887    // status array to only print those rows successfully fetched. Code to
4888    // check if rc equals SQL_SUCCESS_WITH_INFO or SQL_ERROR not shown.
4889    while ((rc = SQLFetchScroll(hstmt,SQL_FETCH_NEXT,0)) != SQL_NO_DATA) {
4890       for (i = 0; i < NumRowsFetched; i++) {
4891          if (RowStatusArray[i] == SQL_ROW_SUCCESS||SQL_ROW_SUCCESS_WITH_INFO) {
4892             if (OrderInfoArray[i].OrderIDInd == SQL_NULL_DATA)
4893                printf(' NULL      ')
4894             else
4895                printf('%d', OrderInfoArray[i].OrderID);
4896             if (OrderInfoArray[i].SalesPersonLenOrInd == SQL_NULL_DATA)
4897                printf(' NULL       ')
4898             else
4899                printf('%s', OrderInfoArray[i].SalesPerson);
4900             if (OrderInfoArray[i].StatusLenOrInd == SQL_NULL_DATA)
4901                printf(' NULL\n')
4902             else
4903                printf('%s\n', OrderInfoArray[i].Status);
4904          }
4905       }
4906    }
4907    // Close the cursor.
4908    SQLCloseCursor(hstmt);
```

4909 **Bind Offsets**

4910 When using row-wise binding, an application can specify that an offset be added to buffer
4911 addresses that are bound to column data. The fetch functions (*SQLFetch*(), *SQLFetchScroll*(), and
4912 *SQLSetPos*()) add this offset to bound buffer addresses to obtain the effective address. A bind
4913 offset is measured in terms of octets.

4914 A bind offset is not added to a pointer whose value is 0. These pointers are not bound. If a bind
4915 offset is used, the pointers do not have to contain valid addresses, but the sum of the pointer and
4916 the offset must be a valid address at the time the fetch function is called.

4917 To specify a bind offset, the application sets the SQL_ATTR_ROW_BIND_OFFSET_PTR
4918 statement attribute to the address of an application variable of type SQLINTEGER. Before the
4919 application calls a fetch function, it places an offset in this variable.

4920 Bind offsets let an application change bindings without calling *SQLBindCol*() again. The
4921 application can change the bind offset at any time; all subsequent fetches use the new bind
4922 offset.

4923 ### 11.1.2 Additional Result Information

4924 The application can bind two additional statement attributes to application variables to obtain
4925 more information about a multi-row fetch:

4926 SQL_ATTR_ROWS_FETCHED_PTR
4927 If the application sets this attribute to point to an application variable of type
4928 SQLUINTEGER, a fetch function sets the variable to the number of rows in the row-set that
4929 it fetched, *SQLSetPos*() sets the variable to the number of rows that the operation affected,
4930 and *SQLBulkOperations*() sets the variable to the number of rows fetched if *Operation* is
4931 SQL_FETCH_BY_BOOKMARK. In all cases, this is the same value as the
4932 SQL_DESC_ROWS_PROCESSED_PTR field of the implementation descriptor.

4933 SQL_ATTR_ROW_STATUS_PTR
4934 If the row-set size is greater than 1, then if the application chooses to set this statement
4935 attribute, it points it not to a scalar variable but to an array of elements of type
4936 SQLUINTEGER. A fetch function uses this array as the *row status array*, in which each
4937 element provides the status of one fetched row, using the values listed in Section 10.4.3 on
4938 page 134.

4939 In cases where an error fetching a single row does not prevent the implementation from
4940 fetching subsequent rows, the fetch function returns SQL_SUCCESS_WITH_INFO and the
4941 application uses the row status array to identify the row that produced the error.

4942 Both of the buffers pointed to by these fields are allocated by the application and populated by
4943 the implementation. As with other bound variables, the application must ensure that the buffers
4944 remain allocated as long as the cursor is open.

4945 ### 11.1.3 Using Multi-row Fetch

4946 To perform a multi-row fetch, the application sets the row-set size to a number greater than 1,
4947 selects the binding style, binds application variables to hold the results, may set the
4948 SQL_ATTR_ROWS_FETCHED_PTR and SQL_ATTR_ROW_STATUS_PTR statement attributes,
4949 and calls *SQLFetch*() or *SQLFetchScroll*().

4950 **Choice of Row-set Size**

4951 The application typically selects a row-set size based on one of the following:

4952 • Screen-based applications commonly set the row-set size to the number of rows displayed on
4953 the screen. In this case, the various values of the *FetchOrientation* argument of
4954 *SQLFetchScroll*() map directly to typical keystroke operations, such as requests to display the
4955 first, previous, or next screenful.

4956 • Setting the row-set size to the largest value the application can reasonably handle effects the
4957 maximum reduction of network traffic and overhead. The optimal value depends on the size
4958 of each row and the amount of available memory.

4959 **Changes to Row-set Size**

4960 The application can change the row-set size and/or bind new row-set buffers (by calling
4961 *SQLBindCol*() or specifying a bind offset) even after rows have been fetched. The implications of
4962 changing the row-set size depend on the function:

4963 • *SQLFetch*() and *SQLFetchScroll*() use the row-set size at the time of the call to determine how
4964 many rows to fetch. (But *SQLFetchScroll*() with SQL_FETCH_NEXT increments the cursor
4965 based on the row-set of the previous fetch, then fetches a row-set based on the current row-
4966 set size.)

- *SQLSetPos*() uses the row-set size in effect as of the preceding call to *SQLFetch*() or *SQLFetchScroll*(), because *SQLSetPos*() operates on a row-set that has already been set.
- *SQLBulkOperations*() uses the row-set size at the time of the call, since it performs operations on a table independent of any fetched row-set.

SQLGetData() and Multi-row Fetch

SQLGetData() operates on a single column in order to obtain long data in parts. On some implementations, *SQLGetData*() can be used when more than one row was fetched, but the application must first call *SQLSetPos*() to position the cursor on a single row. It then calls *SQLGetData*() for a column in that row. To determine if an implementation supports the use of *SQLGetData*() after a multi-row fetch, an application calls *SQLGetInfo*() with the SQL_GETDATA_EXTENSIONS option (SQL_GD_BLOCK bit).

11.2 Scrollable Cursors

A *scrollable cursor* is a cursor that can move backward and forward over the result set. These are common in 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 won't do the job, as scrollable cursors are generally more expensive than forward-only cursors.

To cover the needs of different applications, XDBC defines four different types of scrollable cursors. The four types of scrollable cursors are: static, dynamic, keyset-driven, and mixed. These cursors vary both in expense and in their ability to detect changes to the result set.

Detecting Changes to Tables

The ability to move backward, and to re-read rows of a table that may be subject to change from multiple sources raises the question of whether the cursor ''sees'' these changes.[16] When fetching a row previously fetched, should a scrollable cursor fetch the same values it fetched before, or should it fetch the most current values? Different types of scrollable cursor answer this question in different ways.

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 reservations system needs a cursor that shows the latest changes to the data; without such a cursor, it must continually re-query the database to accurately show available seating.

11.2.1 Scrollable Cursor Types

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, using any technique in XDBC.
- **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.[17]
- **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.

---

16. This discussion only concerns the information fetched when an application re-fetches rows. This specification does not envisage any technique by which the implementation alerts the application to changes to tables at other times, such as changes to the rows currently fetched.

17. Visibility of changes made outside the transaction also depends on the transaction isolation level. See Section 14.2.4 on page 188.

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, in spite of 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. An application can determine whether static cursors detect these changes as described in **Detecting Cursor Capabilities with SQLGetInfo()** on page 402. 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.

Dynamic Cursors

A dynamic cursor can detect any changes made to the membership, order, and values of the result set after the cursor is opened. For example, suppose a dynamic cursor fetches two rows and another application then updates one of those rows and deletes the other. If the dynamic cursor then attempts to refetch those rows, it won't find the deleted row, but will return the new values for the updated row.

Dynamic cursors detect all updates, deletes, and inserts, both their own and those made by others. The row status array specified by the SQL_ATTR_ROW_STATUS_PTR statement attribute reflects these changes and can contain SQL_ROW_SUCCESS, SQL_ROW_SUCCESS_WITH_INFO, SQL_ROW_ERROR, SQL_ROW_UPDATED, and SQL_ROW_ADDED. It cannot return SQL_ROW_DELETED because it does not return deleted rows.

Dynamic cursors can be simulated by requiring the result set to be ordered by a unique key. With such a restriction, fetches are made by executing a SELECT statement each time the cursor fetches rows. For example, suppose the result set is defined by the statement:

```
SELECT * FROM Customers ORDER BY Name, CustID
```

To fetch the next row-set in this result set, the simulated cursor sets the parameters in the following SELECT statement to the values in the last row of the current row-set, then executes it:

```
SELECT * FROM Customers WHERE (Name > ?) AND (CustID > ?)
   ORDER BY Name, CustID
```

This statement creates a second result set, the first row-set of which is the next row-set in the original result set — in this case, the set of rows in the Customers table. The cursor returns this row-set to the application.

It is interesting to note that a dynamic cursor implemented in this manner actually creates many result sets, which allows it to detect changes to the original result set. The application never

5060 learns of the existence of these auxilliary result sets; it simply appears as if the cursor is able to
5061 detect changes to the original result set.

5062 **Keyset-Driven Cursors**

5063 A keyset-driven cursor lies between a static and a dynamic cursor in its ability to do detect
5064 changes. Like a static cursor, it does not always detect changes to the membership and order of
5065 the result set. Like a dynamic cursor, it does detect changes to the values of rows in the result set.

5066 When a keyset-driven cursor is opened, it saves the keys for the entire result set; this fixes the
5067 apparent membership and order of the result set. As the cursor scrolls through the result set, it
5068 uses the keys in this *keyset* to retrieve the current data values for each row. For example, suppose
5069 a keyset-driven cursor fetches a row and another application then updates that row. If the cursor
5070 refetches the row, the values it sees are the new ones, because it refetched the row using its key.
5071 Because of this, the keyset-driven cursors always detect changes made by themselves and others.

5072 When the cursor attempts to retrieve a row that has been deleted, this row appears as a *hole* in
5073 the result set: the key for the row exists in the keyset but the row no longer exists in the result
5074 set. If the key values in a row are updated, the row is considered to have been deleted and then
5075 inserted, so such rows also appear as holes in the result set. While a keyset-driven cursor can
5076 always detect rows deleted by others, it can optionally remove the keys for rows it deletes itself
5077 from the keyset. Keyset-driven cursors that do this cannot detect their own deletes.

5078 Rows inserted by others are never visible to a keyset-driven cursor because no keys for these
5079 rows exist in the keyset. However, a keyset-driven cursor can optionally add the keys for rows it
5080 inserts itself to the keyset. Keyset-driven cursors that do this can detect their own inserts.

5081 An application can determine whether keyset-driven cursors detect their own inserts and deletes
5082 as described in **Detecting Cursor Capabilities with SQLGetInfo()** on page 402.

5083 The row status array specified by the SQL_ATTR_ROW_STATUS_PTR statement attribute can
5084 contain SQL_ROW_SUCCESS, SQL_ROW_SUCCESS_WITH_INFO, or SQL_ROW_ERROR for
5085 any row. It returns SQL_ROW_UPDATED, SQL_ROW_DELETED, or SQL_ROW_ADDED for
5086 rows it detects as updated, deleted, or inserted.

5087 Keyset-driven cursors are commonly implemented by creating a temporary table that contains
5088 the keys for each row in the result set. Because the cursor must also determine if rows have been
5089 updated, this table also commonly contains a column with row versioning information.

5090 To scroll over the original result set, the keyset-driven cursor opens a static cursor over the
5091 temporary table. To retrieve a row in the original result set, the cursor first retrieves the
5092 appropriate key from the temporary table, then retrieves the current values for the row. On a
5093 multi-row fetch, the cursor must retrieve multiple keys and rows.

5094 **Mixed Cursors**

5095 A mixed cursor is a combination of a keyset-driven cursor and a dynamic cursor. It is used when
5096 the result set is too large to reasonably save keys for the entire result set. Mixed cursors are
5097 implemented by creating a keyset that is smaller than the entire result set but larger than the
5098 row-set.

5099 As long as the application scrolls within the keyset, the behavior is keyset-driven. When the
5100 application scrolls outside the keyset, the behavior is dynamic: the cursor fetches the requested
5101 rows and creates a new keyset. Note that after the new keyset is created, the behavior reverts to
5102 keyset-driven within that keyset.

5103 For example, suppose a result set has 1000 rows and used a mixed cursor with a keyset size of
5104 100 and a row-set size of 10. When the first row-set is fetched, the cursor creates a keyset
5105 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.

If a mixed cursor had a keyset size of 1, it would be a dynamic cursor. If a mixed cursor had a keyset equal to the entire result set, it would be a keyset-driven cursor.

11.2.2 Using Scrollable Cursors

There are three steps to using a scrollable cursor:

- Determine the cursor capabilities.
- Set up the cursor.
- Scroll and fetch rows.

Determining Cursor Capabilities

An application can call *SQLGetInfo*() to determine the cursor capabilities supported through a connection.

- The application specifies an *InfoItem* of SQL_SCROLL_OPTIONS to determine the supported cursor types (forward-only, static, keyset-driven, dynamic, or mixed). All data sources must support forward-only cursors.

- The application can determine what operations are valid on various types of cursor, and to determine other attributes of various types of cursor, as described in **Detecting Cursor Capabilities with SQLGetInfo()** on page 402.

Generic applications typically determine cursor capabilities at run time by calling *SQLGetInfo*(). Vertical and custom applications may determine cursor capabilities during development and assumptions about them may be coded into the application.

Setting Up the Cursor

The application can specify the cursor type before executing a statement that creates a result set by setting the SQL_ATTR_CURSOR_TYPE statement attribute. If the application does not explicitly specify a type, a forward-only cursor is 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 row-set size; for more information, see Section 11.1.3 on page 145.

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 row-set to fetch, as shown in the following diagrams.

Row-set returned with *FetchOrientation* = SQL_FETCH_FIRST
Row-set returned with *FetchOrientation* = SQL_FETCH_PRIOR
Current row-set
Row-set returned with *FetchOrientation* = SQL_FETCH_NEXT
Row-set returned with *FetchOrientation* = SQL_FETCH_LAST

Figure **11-3.** Fetching next, prior, first, and last row-sets

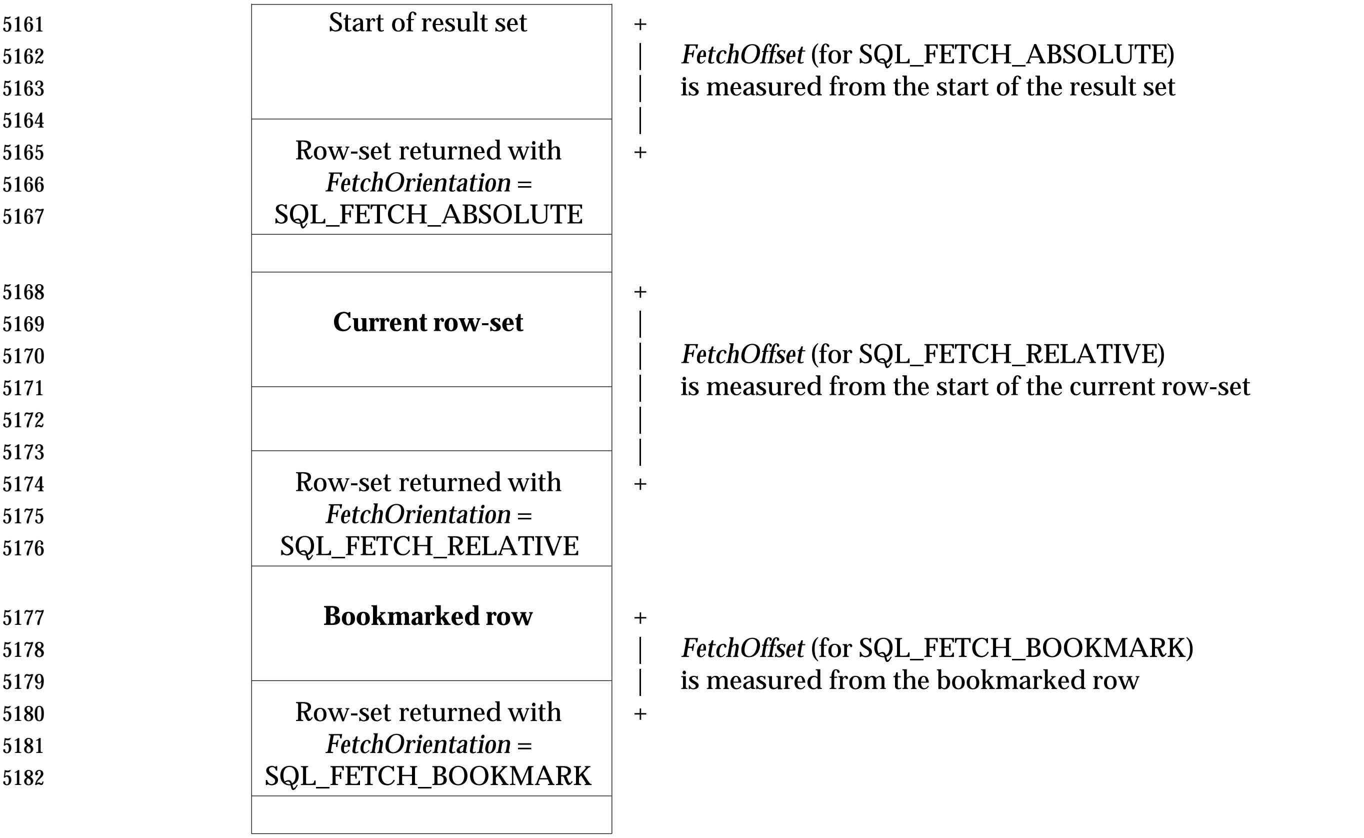


Figure **11-4.** Fetching absolute, relative, and bookmarked row-sets

SQLFetchScroll() positions the cursor to the specified row and returns the rows in the row-set starting with that row. If there are fewer rows remaining in the result set than the row-set size, *SQLFetchScroll*() returns a partial row-set. If a cursor position is specified that is before the start of the result set, *SQLFetchScroll*() moves to the first row of the result set.

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 row-set has been retrieved, the application can call *SQLSetPos*() to position to a particular row in the row-set or refresh rows in the row-set. For more information on using *SQLSetPos*(), see Chapter 12.

In addition to data, *SQLFetchScroll*() can return row status, as described in Section 10.4.3 on page 134.

5198 ### 11.2.3 Relative and Absolute Scrolling

5199 The *FetchOrientation* argument to *SQLFetchScroll*() selects an addressing mode for the new cursor
5200 position. *SQLFetchScroll*() supports *relative scrolling* (addressing modes that depend on the
5201 current cursor position) and *absolute scrolling* (addressing modes that do not depend on the
5202 current cursor position).

5203 Relative scrolling means a call to *SQLFetchScroll*() to fetch the row-set *n* rows from the start of
5204 the current row-set, where a negative value of *n* specifies movement toward the start of the
5205 result set. The function also supports a fetch of the next and prior row-sets. These operations
5206 are also relative to the current row-set.

5207 Absolute scrolling includes calls to *SQLFetchScroll*() to fetch the first row-set and the row-set
5208 specified absolutely as starting at row *n*. The function also supports a fetch of the last row-set,
5209 and uses negative values of *n* to specify rows counting back from the last row in the result set.

5210 Absolute scrolling modes do not make sense when applied to dynamic cursors. Dynamic
5211 cursors do not treat rows as occupying a fixed, numbered position, but detect rows inserted into
5212 and deleted from the result set. Therefore, dynamic cursors cannot retrieve the row at a
5213 particular number except by reading from the start of the result set, which is likely to be slow.
5214 Furthermore, absolute fetching is not very useful in dynamic cursors because row numbers
5215 change as rows are inserted and deleted; thus, successively fetching the same row number can
5216 yield different rows.

5217 Applications that use *SQLFetchScroll*() only for its multi-row fetch capabilities, such as report
5218 generators, are likely to pass through the result set a single time, using only the option to fetch
5219 the next row-set. Screen-based applications, on the other hand, can take advantage of all of the
5220 capabilities of *SQLFetchScroll*(). If the application sets the row-set size to the number of rows
5221 displayed on the screen and binds the screen buffers to the result set, it can translate scroll bar
5222 operations directly to calls to *SQLFetchScroll*():

	Scroll bar operation	SQLFetchScroll() scrolling operation
5223		
5224	Page up	SQL_FETCH_PRIOR
5225	Page down	SQL_FETCH_NEXT
5226	Line up	SQL_FETCH_RELATIVE with *FetchOffset* = −1
5227	Line down	SQL_FETCH_RELATIVE with *FetchOffset* = 1
5228	Scroll box at top	SQL_FETCH_FIRST
5229	Scroll box at bottom	SQL_FETCH_LAST
5230	Arbitrary scroll box position	SQL_FETCH_ABSOLUTE

5231 Such applications also need to position the scroll box after a scrolling operation, which requires
5232 the current row number and the number of rows. For the current row number, applications can
5233 either keep track of the current row number or call *SQLGetStmtAttr*() with the
5234 SQL_ATTR_ROW_NUMBER attribute to retrieve it.

5235 The number of rows fetched, which is the size of the current row-set, is available as the
5236 SQL_DIAG_CURSOR_ROW_COUNT field of the diagnostic header. It is implementation-
5237 defined whether row counts are available for various cursor types; the application can determine
5238 the level of support as described in **Detecting Cursor Capabilities with SQLGetInfo()** on page
5239 402. Applications can also determine the number of rows affected by the fetch operation by
5240 calling *SQLRowCount*() (see Section 12.2 on page 162).

11.2.4 Bookmarks

A bookmark is a value used to identify a row of data. The meaning of the bookmark value is known only to the implementation or data source. The value is sufficient to enable the implementation or data source to directly move to the associated row.

To determine whether bookmarks are supported for a given cursor type, see **Detecting Cursor Capabilities with SQLGetInfo()** on page 402. To determine the persistence of bookmarks, call *SQLGetInfo*() with the SQL_BOOKMARK_PERSISTENCE option.

Bookmark Types

Bookmarks are variable-length data structures. A bookmark could be based on a primary key or a unique index associated with a table or could be a 32-bit value. To specify that a bookmark is used with a cursor, the application sets the SQL_ATTR_USE_BOOKMARK statement attribute to SQL_UB_VARIABLE.

The SQL_DESC_OCTET_LENGTH field of record 0 of the IRD contains the maximum length of a bookmark. An application can call *SQLColAttribute*() or *SQLGetDescField*() with a *FieldIdentifier* of SQL_DESC_OCTET_LENGTH to obtain the length of the bookmark. (Describing a bookmark column between the preparation and the execution of an SQL statement has performance implications; see **Performance Note** on page 279.) Since a 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 row-set.

Retrieving Bookmarks

The application must set the SQL_ATTR_USE_BOOKMARKS statement attribute before preparing or executing a statement that uses bookmarks. The default is to not use bookmarks because building and maintaining bookmarks can be costly.

Bookmarks are returned as column 0 of the result set. The application can retrieve them in any of the following ways:

- Bind column 0 of the result set. *SQLFetch*() or *SQLFetchScroll*() returns the bookmarks for each row in the row-set along with the data for other bound columns.
- Call *SQLSetPos*() to position to a row in the row-set, then call *SQLGetData*() for column 0. (If an implementation supports bookmarks, it must always support the ability to call *SQLGetData*() for column 0, even if it does not let applications call *SQLGetData*() for other columns before the last bound column.)
- Call *SQLBulkOperations*() with an *Operation* of SQL_ADD to return the bookmark of an inserted row, if column 0 is bound.

Scrolling by Bookmark

When fetching rows with *SQLFetchScroll*(), the 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 row-set starting with the row identified by that bookmark. An application can specify an offset for this operation in *FetchOffset*. When an offset is specified, the first row of the returned row-set is determined by adding *FetchOffset* to the number of the row identified by the bookmark.

5283 **Comparing Bookmarks**

5284 Bookmarks can be compared for equality or inequality by treating each bookmark as an array of
5285 octets and comparing two bookmarks octet-by-octet. Since bookmarks are guaranteed to be
5286 distinct only within a result set, it makes no sense to compare bookmarks obtained from
5287 different result sets.

11.3 Multiple Results

A batch (see Section 9.3.4 on page 99) can generate multiple results (result sets and/or row counts).

To determine the implementation's level of support for multiple results, the application can call *SQLGetInfo*() with the SQL_MULT_RESULT_SETS option. This provides a rough indication on whether multiple results are supported.

The application can get more detailed information on the level of support for various types of batch by calling *SQLGetInfo*() as follows:

For explicit batches and procedures:
  Explicit batches and procedures always return multiple result sets when they include multiple result-set-generating statements. The SQL_BATCH_SUPPORT option of *SQLGetInfo*() indicates whether row-count-generating statements are allowed in batches; the SQL_BATCH_ROW_COUNT option indicates whether these row counts are returned to the application.

For arrays of parameters:
  The SQL_PARAM_ARRAY_SELECTS option of *SQLGetInfo*() indicates whether result sets are returned. The SQL_PARAM_ARRAY_ROW_COUNTS option indicates whether row counts are returned.

These options indicate whether the data source returns a total row count for the batch or individual row counts for each statement in the batch; and, in the case of a result-set-generating statement executed with an array of parameters, whether the data source returns a single result set for all sets of parameters or individual result sets for each set of parameters.

To process multiple results, an application calls *SQLMoreResults*(). This function discards any 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 discard the result set and 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.

Because this example was coded as a batch, the caller cannot first inspect the list of parts and then decide whether to update them. The entire batch statement is executed before any results are available; *SQLMoreResults*() simply makes each result available in turn.

If one of the statements in a batch fails, *SQLMoreResults*() returns one of the following:

SQL_ERROR
  if the batch was aborted when the statement failed, or if the statement that failed was the last statement in the batch.

SQL_SUCCESS_WITH_INFO
  if a statement before the last statement failed and execution of the batch continued.

SQL_SUCCESS_WITH_INFO indicates that at least one result set or count was generated.

Chapter 12

Updating Data

Applications can update data either by executing the UPDATE, DELETE, and INSERT statements of SQL (see Section 12.1 on page 158) or by calling *SQLBulkOperations*() or *SQLSetPos*() (see Section 12.3 on page 163).

Searched UPDATE, DELETE, and INSERT statements contain a specification of the rows to change and and are usually supported. 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 do not detect changes. For information about whether scrollable cursors can detect changes, see Section 11.2 on page 147.

12.1 UPDATE, DELETE, and INSERT Statements

SQL-based applications make changes to tables by executing the UPDATE, DELETE, and INSERT statements. For the general syntax definition of these statements, see the X/Open **SQL** specification. Searched UPDATE and DELETE specify the rows to update or delete. Positioned UPDATE and DELETE rely on a cursor (see Section 12.1).

Use of Parameters

Like other SQL statements, UPDATE, DELETE, and INSERT 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 Section 9.4 on page 102.

12.1.1 Positioned UPDATE and DELETE

Applications can update or delete the current row in a result set with a positioned UPDATE or DELETE statement. The X/Open **SQL** specification defines the syntax of these statements.

Not all data sources support these statements. To determine whether a data source supports them for various types of cursors, see **Detecting Cursor Capabilities with SQLGetInfo()** on page 402 (the SQL_CA1_POSITIONED_UPDATE and SQL_CA1_POSITIONED_DELETE bitmasks).

To use a positioned UPDATE or DELETE statement, the application must create a result set with a SELECT FOR UPDATE statement. The application then positions the cursor on the row to be updated or deleted. It can do this by calling *SQLFetchScroll*() to retrieve a row-set containing the row it requires and calling *SQLSetPos*() to select a current row from the row-set. The application then executes the positioned UPDATE or DELETE statement, using a different statement handle from the one it used to generate the result set.

The UPDATE and DELETE statements require a cursor name. The application can either specify a cursor name with *SQLSetCursorName*() before executing the statement that creates the result set or it 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*().

12.1.2 Code Example

For example, the following code lets a user scroll through the Customers table and deletes customer records or update their address and phone number. 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 row-set buffers and binds the elements of these buffers. This keeps the row-set buffers synchronized with the updated data.

```
#define POSITIONED_UPDATE 100
#define POSITIONED_DELETE 101

SQLUINTEGER  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;
```

```
5389        SQLHSTMT     hstmtCust, hstmtUpdate, hstmtDelete;
5390        // Set the SQL_ATTR_BIND_TYPE statement attribute to use column-wise binding. Declare
5391        // the row-set size with the SQL_ATTR_ROW_ARRAY_SIZE statement attribute. Set the
5392        // SQL_ATTR_ROW_STATUS_PTR statement attribute to point to the row status array.
5393        SQLSetStmtAttr(hstmtCust, SQL_ATTR_BIND_TYPE, SQL_BIND_BY_COLUMN, 0);
5394        SQLSetStmtAttr(hstmtCust, SQL_ATTR_ROW_ARRAY_SIZE, 10, 0);
5395        SQLSetStmtAttr(hstmtCust, SQL_ATTR_ROW_STATUS_PTR, RowStatusArray, SQL_IS_POINTER);
5396        // Bind arrays to the CustID, Name, Address, and Phone columns.
5397        SQLBindCol(hstmtCust, 1, SQL_C_ULONG, CustIDArray, 0, CustIDIndArray);
5398        SQLBindCol(hstmtCust, 2, SQL_C_CHAR, NameArray, sizeof(NameArray[0]),
5399                   NameLenOrIndArray);
5400        SQLBindCol(hstmtCust, 3, SQL_C_CHAR, AddressArray, sizeof(AddressArray[0]),
5401                   AddressLenOrIndArray);
5402        SQLBindCol(hstmtCust, 4, SQL_C_CHAR, PhoneArray, sizeof(PhoneArray[0]),
5403                   PhoneLenOrIndArray);
5404        // Set the cursor name to Cust.
5405        SQLSetCursorName(hstmtCust, 'Cust', SQL_NTS);
5406        // Prepare positioned UPDATE and DELETE statements.
5407        SQLPrepare(hstmtUpdate,
5408           'UPDATE Customers SET Address = ?, Phone = ? WHERE CURRENT OF Cust',
5409           SQL_NTS);
5410        SQLPrepare(hstmtDelete, 'DELETE FROM Customers WHERE CURRENT OF Cust', SQL_NTS);
5411        // Execute a statement to retrieve rows from the Customers table.
5412        SQLExecDirect(hstmtCust,
5413           'SELECT CustID, Name, Address, Phone FROM Customers FOR UPDATE OF Address, Phone',
5414           SQL_NTS);
5415        // Fetch and display the first 10 rows.
5416        SQLFetchScroll(hstmtCust, SQL_FETCH_NEXT, 0);
5417        DisplayData(CustIDArray, CustIDIndArray, NameArray, NameLenOrIndArray, AddressArray,
5418                    AddressLenOrIndArray, PhoneArray, PhoneLenOrIndArray, RowStatusArray);
5419        // Call GetAction to get an action and a row number from the user.
5420        while (GetAction(&Action, &RowNum)) {
5421           switch (Action) {
5422              case SQL_FETCH_NEXT:
5423              case SQL_FETCH_PRIOR:
5424              case SQL_FETCH_FIRST:
5425              case SQL_FETCH_LAST:
5426              case SQL_FETCH_ABSOLUTE:
5427              case SQL_FETCH_RELATIVE:
5428                 // Fetch and display the requested data.
5429                 SQLFetchScroll(hstmtCust, Action, RowNum);
5430                 DisplayData(CustIDArray, CustIDIndArray, NameArray, NameLenOrIndArray,
5431                             AddressArray, AddressLenOrIndArray, PhoneArray,
5432                             PhoneLenOrIndArray, RowStatusArray);
5433                 break;
5434              case POSITIONED_UPDATE:
5435                 // Get the new data and place it in the row-set buffers.
5436                 GetNewData(AddressArray[RowNum - 1], &AddressLenOrIndArray[RowNum - 1],
5437                            PhoneArray[RowNum - 1], &PhoneLenOrIndArray[RowNum - 1]);
5438                 // Bind the elements of the arrays at position RowNum-1 to the parameters
5439                 // of the positioned UPDATE statement.
5440                 SQLBindParameter(hstmtUpdate, 1, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR,
5441                                  50, 0, AddressArray[RowNum - 1], sizeof(AddressArray[0]),
5442                                  &AddressLenOrIndArray[RowNum - 1]);
5443                 SQLBindParameter(hstmtUpdate, 2, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR,
5444                                  10, 0, PhoneArray[RowNum - 1], sizeof(PhoneArray[0]),
5445                                  &PhoneLenOrIndArray[RowNum - 1]);
5446                 // Define RowNum as the current row of the row-set.
5447                 SQLSetPos(hstmtCust, RowNum, SQL_POSITION, SQL_LOCK_NO_CHANGE);
5448                 // Execute the positioned UPDATE statement to update the row.
5449                 SQLExecute(hstmtUpdate);
5450                 break;
```

```
            case POSITIONED_DELETE:
               // Define RowNum as the current row of the row-set.
               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);
```

12.1.3 Simulating Positioned UPDATE and DELETE

If the data source does not support positioned UPDATE and DELETE statements, the implementation may simulate them by converting positioned statements to searched ones. It would replace the WHERE CURRENT OF clause with a searched WHERE clause that identifies the current row.

For example, if the implementation determines that the value of the CustID column uniquely identifies each row in the Customers table, it might convert the following positioned DELETE statement:

```
DELETE FROM Customers WHERE CURRENT OF CustCursor
```

to the following:

```
DELETE FROM Customers WHERE (CustID = ?)
```

The implementation may use one of the following *row identifiers* in such a WHERE clause:

- Columns whose values serve to uniquely identify every row in the table. For example, calling *SQLSpecialColumns*() with SQL_BEST_ROWID returns the optimal columns 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

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. Using all the columns in the result set is usually 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 doesn't contain all the columns that exist in the underlying table.

If the data source supports both strategies, applications can choose one with the SQL_ATTR_SIMULATE_CURSOR statement attribute. The application removes the risk that a simulated operation will affect multiple rows by ensuring that the columns in the result set uniquely identify each row in the result set. This keeps the implementation from having to generate a row identifier.

If the implementation 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 implementation adds the necessary columns to the end of the select list. (Some data sources have a single column that always uniquely identifies each row. Otherwise, the implementation uses the information available to the application through *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 data source identifies rows, the implementation 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 only used with positioned UPDATE and DELETE statements and 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 implementation 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 implementation for each column it uses in the WHERE clause. After the implementation 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 implementation calls *SQLSpecialColumns*() for the Customers table, discovers that CustID is the key to the Customers table, 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 implementation only strips 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 requested a guarantee of uniqueness, the implementation 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 implementation 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 = ?)
```

5531 ## 12.2 Determining the Number of Affected Rows

5532 After an application updates, deletes, or inserts rows, it can call *SQLRowCount*() to determine
5533 how many rows were affected. *SQLRowCount*() returns this value regardless of whether the
5534 rows were updated, deleted, or inserted by executing an UPDATE, DELETE, or INSERT
5535 statement, by executing a positioned UPDATE or DELETE statement, or by calling
5536 *SQLBulkOperations*() or *SQLSetPos*().

5537 If a batch of SQL statements is executed (see Section 9.3.4 on page 99), the count of affected rows
5538 might be a total count for all statements in the batch or individual counts for each statement in
5539 the batch. For more information, see Section 11.3 on page 156.

5540 The number of affected rows is also returned in the SQL_DIAG_ROW_COUNT header field in
5541 the diagnostic area associated with the statement handle. However, this field is reset after every
5542 function call on the same statement handle, whereas the value returned by *SQLRowCount*()
5543 remains the same until a call to *SQLBulkOperations*(), *SQLExecDirect*(), *SQLExecute*(),
5544 *SQLPrepare*(), or *SQLSetPos*().

12.3 Using SQLSetPos()

Applications can update or delete any row in the row-set or insert new rows with *SQLSetPos*(). Calling *SQLSetPos*() is a convenient alternative to constructing and executing an SQL statement. It lets an XDBC implementation support positioned updates even when the data source doesn't support positioned SQL statements. It is part of the paradigm of achieving complete database access by means of function calls.

An application can determine which *SQLSetPos*() operations are supported for various cursor types, as described in **Detecting Cursor Capabilities with SQLGetInfo()** on page 402.

On SQL-based data sources, a call to *SQLSetPos*() may be implemented by constructing and executing an UPDATE or DELETE statement.

Row Addressing

SQLSetPos() operates within the current row-set and can be used only after a call to *SQLFetchScroll*(). The application specifies the number of the row to update, delete, or insert, using the *RowNumber* argument, and the implementation retrieves the new data for that row from the row-set buffers. *SQLSetPos*() can also be used to re-fetch a specified row of the row-set from the data source or to designate a specified row as the current row.

The first row in the row-set is row number 1. *RowNumber* must identify a row in the row-set — 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 row-set size), inclusive; except that setting *RowNumber* to 0 has special meaning for some values of *Operation*.

SQLSetPos() ignores any changes made to the row-set size since the rows were fetched, because it operates on the rows in the fetched row-set.

12.3.1 Updating Rows 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 (except when the value in the length/indicator buffer is SQL_COLUMN_IGNORE). Unbound columns are not updated.

To update rows with *SQLSetPos*(), the application:

- Places the new data values in the row-set buffers. For information on how to send long data with *SQLSetPos*(), see Section 12.4.4 on page 167.
- Sets the value in the length/indicator buffer of each column as necessary. This is the octet length of the data or SQL_NTS for columns bound to string buffers, the octet length of the data for columns bound to binary buffers, and SQL_NULL_DATA for any columns to be set to NULL.
- 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.
- 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 row-set are updated.

The update operation of *SQLSetPos*() does not affect which row of the row-set is the current row.

When updating all rows of the row-set (*RowNumber* = 0), an application can disable the update of certain rows by setting the corresponding elements in the row operation array (pointed to by the SQL_ATTR_ROW_OPERATION_PTR statement attribute) to SQL_ROW_IGNORE.

5588 The row operation array corresponds in size and number of elements to the row status array
5589 (pointed to by the SQL_ATTR_ROW_STATUS_ARRAY statement attribute). To update only
5590 those rows in the result set that were successfully fetched and have not been deleted from the
5591 row-set, the application uses the row status array from the function that fetched the row-set as
5592 the row operation array to *SQLSetPos*().[18]

5593 For every row that is sent to the data source as an update, the application buffers should have
5594 valid row data. If the application buffers were filled by fetching and if a row status array has
5595 been maintained, its value at each of these row positions should not be SQL_ROW_DELETED,
5596 SQL_ROW_ERROR, or SQL_ROW_NOROW.

5597 ### 12.3.2 Deleting Rows with SQLSetPos()

5598 The delete operation of *SQLSetPos*() makes the data source delete one or more selected rows of a
5599 table.

5600 To delete rows with *SQLSetPos*(), the application calls *SQLSetPos*() with *Operation* set to
5601 SQL_DELETE and *RowNumber* set to the number of the row to delete. If *RowNumber* is 0, all
5602 rows in the row-set are deleted.

5603 After *SQLSetPos*() returns, the deleted row is the current row, and its status is
5604 SQL_ROW_DELETED. The row cannot be used in any further positioned operations, such as
5605 calls to *SQLGetData*() or *SQLSetPos*().

5606 When deleting all rows of the row-set (*RowNumber* = 0), the application can prevent the
5607 implementation from deleting certain rows by using the row operation array, in the same way as
5608 for the update operation of *SQLSetPos*() (see Section 12.3.1 on page 163).

5609 Every row that is deleted should be a row that exists in the result set. If the application buffers
5610 were filled by fetching and if a row status array has been maintained, its value at each of these
5611 row positions should not be SQL_ROW_DELETED, SQL_ROW_ERROR, or
5612 SQL_ROW_NOROW.

5613 ---

5614 18. Values of the row status array that indicate a successfully fetched row that is still present (SQL_SUCCESS, SQL_UPDATED, etc.)
are equivalent to SQL_ROW_PROCEED, while values of the row status array that indicate rows that were not successfully
5615 fetched or are no longer present (SQL_ERROR or SQL_DELETED) are equivalent to SQL_ROW_IGNORE.

12.4 Using SQLBulkOperations()

The *SQLBulkOperations*() functions performs operations on database tables. *SQLBulkOperations*() requires a result set and a cursor, but uses them solely to specify the underlying table on which to operate. *SQLBulkOperations*() does not base its operations on, nor does it change, the position of any cursor, the selected row, or the current row-set.

12.4.1 Updating Rows by Bookmark with SQLBulkOperations()

When updating 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 is SQL_COLUMN_IGNORE). Unbound columns are not updated.

To update by bookmark with *SQLBulkOperations*(), the application:

- 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.
- 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 buffers containing the bookmark values, to column 0.
- Places the new data values in the row-set buffers. For information on how to send long data with *SQLBulkOperations*(), see Section 12.4.4 on page 167.
- Sets the value in the length/indicator buffer of each column as necessary. This is the octet length of the data or SQL_NTS for columns bound to string buffers, the octet length of the data for columns bound to binary buffers, and SQL_NULL_DATA for any columns to be set to NULL.
- 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.
- Calls *SQLBulkOperations*() with *Operation* set to SQL_UPDATE_BY_BOOKMARK.

The application can prevent the implementation from updating certain rows by using the row operation array, in the same way as for the update operation of *SQLSetPos*() (see Section 12.3.1 on page 163).

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 value at each of these row positions should not be SQL_ROW_DELETED, SQL_ROW_ERROR, or SQL_ROW_NOROW.

12.4.2 Deleting Rows by Bookmark with SQLBulkOperations()

When deleting 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 by bookmark with *SQLBulkOperations*(), the application:

- 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.
- 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 buffers containing the bookmark values, to column 0.
- Calls *SQLBulkOperations*() with *Operation* set to SQL_DELETE_BY_BOOKMARK.

After *SQLBulkOperations*() returns, the status of all deleted rows is SQL_ROW_DELETED. These rows cannot be used in any further positioned operations, such as calls to *SQLGetData*() or *SQLSetPos*().

The application can prevent the implementation from updating certain rows by using the row operation array, in the same way as for the update operation of *SQLSetPos*() (see Section 12.3.1 on page 163).

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 value at each of these row positions should not be SQL_ROW_DELETED, SQL_ROW_ERROR, or SQL_ROW_NOROW.

12.4.3 Inserting Rows with SQLBulkOperations()

Inserting data with *SQLBulkOperations*() is similar to updating data with *SQLBulkOperations*(), as it uses data from application buffers.

To insert rows with *SQLBulkOperations*(), the application:

- Sets the SQL_ATTR_ROW_ARRAY_SIZE statement attribute to the number of rows to insert, and places the new data values in the row-set buffers. For information on how to send long data with *SQLBulkOperations*(), see Section 12.4.4 on page 167.
- Sets the length/indicator buffer of each column as follows:
  - — For columns bound to string buffers: The octet length of the data or SQL_NTS.
  - — For columns bound to binary buffers: The octet length of the data.
  - — For columns to be set to NULL: SQL_NULL_DATA.
  - — For columns to be set to their default value (or to the null value if no default is defined for the column): SQL_IGNORE. In this case, the column must either have a defined default value or must allow the null value.

  So that each column in the new row has a value, all bound columns with a length/indicator value of SQL_IGNORE and all unbound columns must either accept NULL values or have a default.
- Calls *SQLBulkOperations*() with *Operation* set to SQL_ADD.

The application can prevent the implementation from inserting certain rows by using the row operation array, in the same way as for the update operation of *SQLSetPos*() (see Section 12.3.1

on page 163).

12.4.4 Long Data and SQLBulkOperations()/SQLSetPos()

Long data can be sent in parts when updating or inserting rows with *SQLBulkOperations*() or *SQLSetPos*(), in the same way as long parameters are sent in parts (see Section 9.4.3 on page 105).

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*.

SQLBulkOperations() and *SQLSetPos*() operate only on bound columns. An application must bind the affected columns in order to use one of these functions. The application can unbind the column after calling the function so that it can call *SQLGetData*() to retrieve data from the column.

To send data at execution time, the application:

1. Places a 32-bit value in the row-set 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 implementation 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 octets 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 implementations must support the SQL_LEN_DATA_AT_EXEC(length) macro; if the data source does not require the octet length, the implementation can ignore it.

3. Calls *SQLBulkOperations*() or *SQLSetPos*(). The implementation 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. In the buffer pointed to by *ValuePtr*, the implementation returns the value the application placed in the row-set 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 implementation 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 implementation. 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, and the implementation reassembles the data. If the application passes null-terminated string data, the implementation removes the null terminator 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 implementation returns SQL_NEED_DATA and 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*() can then 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. While a statement is in a Need Data state, the application can call only *SQLPutData*(), *SQLParamData*(), or *SQLCancel*(); 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.

5743 ## 12.4.5 Code Example

5744 The following code lets a user scroll through the Customers table and update, delete, or add new
5745 rows. It places the new data in the row-set buffers before calling *SQLSetPos*() to update or add
5746 new rows. An extra row is allocated at the end of the row-set buffers to hold new rows; this
5747 prevents existing data from being overwritten when data for a new row is placed in the buffers.

```
5748 #define UPDATE_ROW 100
5749 #define DELETE_ROW 101
5750 #define ADD_ROW    102
5751 SQLUINTEGER  CustIDArray[11];
5752 SQLCHAR      NameArray[11][51], AddressArray[11][51], PhoneArray[11][11];
5753 SQLINTEGER   CustIDIndArray[11], NameLenOrIndArray[11], AddressLenOrIndArray[11],
5754              PhoneLenOrIndArray[11];
5755 SQLUSMALLINT RowStatusArray[10], Action, RowNum;
5756 // Set the SQL_ATTR_BIND_TYPE statement attribute to use column-wise binding. Declare
5757 // the row-set size with the SQL_ATTR_ROW_ARRAY_SIZE statement attribute. Set the
5758 // SQL_ATTR_ROW_STATUS_PTR statement attribute to point to the row status array.
5759 SQLSetStmtAttr(hstmt, SQL_ATTR_BIND_TYPE, SQL_BIND_BY_COLUMN, 0);
5760 SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_ARRAY_SIZE, 10, 0);
5761 SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_STATUS_PTR, RowStatusArray, 0);
5762 // Bind arrays to the CustID, Name, Address, and Phone columns.
5763 SQLBindCol(hstmt, 1, SQL_C_ULONG, CustIDArray, 0, CustIDIndArray);
5764 SQLBindCol(hstmt, 2, SQL_C_CHAR, NameArray, sizeof(NameArray[0]), NameLenOrIndArray);
5765 SQLBindCol(hstmt, 3, SQL_C_CHAR, AddressArray, sizeof(AddressArray[0]),
5766            AddressLenOrIndArray);
5767 SQLBindCol(hstmt, 4, SQL_C_CHAR, PhoneArray, sizeof(PhoneArray[0]),
5768            PhoneLenOrIndArray);
5769 // Execute a statement to retrieve rows from the Customers table.
5770 SQLExecDirect(hstmt, 'SELECT CustID, Name, Address, Phone FROM Customers', SQL_NTS);
5771 // Fetch and display the first 10 rows.
5772 rc = FetchScroll(hstmt, SQL_FETCH_NEXT, 0);
5773 DisplayData(CustIDArray, CustIDIndArray, NameArray, NameLenOrIndArray, AddressArray,
5774             AddressLenOrIndArray, PhoneArray, PhoneLenOrIndArray, RowStatusArray);
5775 // Call GetAction to get an action and a row number from the user.
5776 while (GetAction(&Action, &RowNum)) {
5777    switch (Action) {
5778       case SQL_FETCH_NEXT:
5779       case SQL_FETCH_PRIOR:
5780       case SQL_FETCH_FIRST:
5781       case SQL_FETCH_LAST:
5782       case SQL_FETCH_ABSOLUTE:
5783       case SQL_FETCH_RELATIVE:
5784          // Fetch and display the requested data.
5785          SQLFetchScroll(hstmt, Action, RowNum);
5786          DisplayData(CustIDArray, CustIDIndArray,
5787                      NameArray, NameLenOrIndArray,
5788                      AddressArray, AddressLenOrIndArray,
5789                      PhoneArray, PhoneLenOrIndArray, RowStatusArray);
5790          break;
5791       case UPDATE_ROW:
5792          // Place the new data in the row-set buffers and update the specified row.
5793          GetNewData(&CustIDArray[RowNum - 1], &CustIDIndArray[RowNum - 1],
5794                     NameArray[RowNum - 1], &NameLenOrIndArray[RowNum - 1],
5795                     AddressArray[RowNum - 1], &AddressLenOrIndArray[RowNum - 1],
5796                     PhoneArray[RowNum - 1], &PhoneLenOrIndArray[RowNum - 1]);
5797          SQLSetPos(hstmt, RowNum, SQL_UPDATE, SQL_LOCK_NO_CHANGE);
5798          break;
5799       case DELETE_ROW:
5800          // Delete the specified row.
5801          SQLSetPos(hstmt, RowNum, SQL_DELETE, SQL_LOCK_NO_CHANGE);
```

```
5802                 break;
5803              case ADD_ROW:
5804                 // Place the new data in the row-set buffers at index 10. This is an extra
5805                 // element for new rows so row-set data is not over-written. Insert the new
5806                 // row. Row 11 corresponds to index 10.
5807                 GetNewData(&CustIDArray[10], &CustIDIndArray[10],
5808                            NameArray[10], &NameLenOrIndArray[10],
5809                            AddressArray[10], &AddressLenOrIndArray[10],
5810                            PhoneArray[10], &PhoneLenOrIndArray[10]);
5811                 SQLBulkOperations(hstmt, 11, SQL_ADD);
5812                 break;
5813           }
5814        }
5815        // Close the cursor.
5816        SQLCloseCursor(hstmt);
```

5817 *Chapter 13*

5818
Descriptors

5819 A descriptor handle refers to a data structure that holds information about either columns or
5820 dynamic parameters. A XDBC descriptor is analogous to the SQL descriptor area. ‘‘Descriptor’’
5821 in this document means the XDBC data structure, not the data structure from SQL.

5822 XDBC functions that operate on column and parameter data implicitly get and set descriptor
5823 fields. For instance, when *SQLBindCol*() is called to bind column data, it sets descriptors fields
5824 that completely describe the binding. When *SQLColAttribute*() is called to describe column data,
5825 it returns data stored in descriptor fields.

5826 An application calling these XDBC functions need not concern itself with descriptors. No
5827 database operation requires that the application gain direct[19] access to descriptors. However, for
5828 some applications, gaining direct access to descriptors streamlines many operations. For
5829 example, direct access to descriptors provides a way to rebind column data that may be more
5830 efficient than calling *SQLBindCol*() again.

5831 ________________

5832 19. XDBC does not define the physical representation of a descriptor. Applications gain ‘‘direct’’ access to the descriptor only by manipulating its fields using XDBC functions and the descriptor handle.

13.1 Types of Descriptor

A descriptor is used to describe one of the following:

1. 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 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.

2. A single row of database data. A row descriptor can be used for:

   — the *implementation row buffer*, which contains the row from the database*

   — the *application row buffer*, which contains the row, following any data conversion the application may specify, in which form the data is presented to the application.

   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 following table summarises the descriptor types:

	Rows	**Dynamic Parameters**
Application Buffer	Application Row Descriptor	Application Parameter Descriptor
Implementation Buffer	Implementation Row Descriptor	Implementation Parameter Descriptor

Table **13-1.** The Four Types of Descriptor

For either the parameter or row buffers, if the application specifies different data types in corresponding records of the implementation and application descriptors, the XDBC implementation performs data conversion when it uses the descriptors. For example, it may convert numeric and date/time values to character-string format. For valid combinations and their effects, see Section D.6 on page 576 and Section D.7 on page 587.

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.[20]

---

* The implementation buffers are conceptually the data as written to, or read from, the database. However, X/Open does not specify the stored form of database data, and a data source could perform additional conversion on the data from its form in the implementation buffer.

20. By reusing a row descriptor that contains a fetched row of a table as a parameter descriptor of an INSERT statement, an application could copy rows between tables without specifying copying of the data at the application level. However, an application can copy rows between different databases in this way only if the implementation supports simultaneous access to multiple connections, because the descriptor is valid only while connected.

5875 It is always known whether a given descriptor is an application descriptor or an implementation
5876 descriptor, even if the descriptor has not yet been used in a database operation. For the
5877 descriptors that the implementation implicitly allocates, the implementation records the
5878 predefined role relative to the statement handle. Any descriptor the application allocates using
5879 *SQLAllocHandle*() is an application descriptor.

13.2 Descriptor Fields

The fields of a descriptor are listed in this section, and described completely in the reference manual entry for *SQLSetDescField*().

Header Fields

A descriptor contains a single copy of the following fields:

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	

Table **13-2.** List of Descriptor Header Fields

For more information on each field, see **Fields of the Descriptor Header** on page 472.

Record Fields

A descriptor contains zero or more descriptor records[21] each containing 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	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

Table **13-3.** List of Descriptor Record Fields

For more information on each field, see **Fields of Each Descriptor Record** on page 476.

Fields that Relate to Statement Attributes

Many statement attributes correspond to the header field of a descriptor. Setting such an attribute by calling *SQLSetStmtAttr*() and setting the corresponding descriptor header field by calling *SQLSetDescField*() have the same effect. Likewise, the same value can be obtained by calling *SQLGetStmtAttr*() as by calling *SQLSetDescField*() for the corresponding descriptor header field. Calling the statement functions instead of the descriptor functions has the advantage that a descriptor handle does not have to be retrieved.

The following descriptor header fields can be set by setting statement attributes:

---

21. These records correspond to the *item descriptor areas* in the SQL descriptor area of SQL.

SQL_DESC_ARRAY_SIZE	SQL_DESC_BIND_TYPE
SQL_DESC_ARRAY_STATUS_PTR	SQL_DESC_ROWS_PROCESSED_PTR
SQL_DESC_BIND_OFFSET_PTR	

Table **13-4.** Descriptor Fields that Relate to Statement Attributes

13.2.1 Count of Records

The SQL_DESC_COUNT header field of a descriptor indicates the number of records that are present in that descriptor. It can range from 0 up to and including an implementation-defined maximum. When a descriptor is allocated, the initial value of SQL_DESC_COUNT is 0.

The XDBC implementation takes any necessary action 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 implementation automatically increments SQL_DESC_COUNT. When the application unbinds the highest-numbered descriptor record (see Section 13.2.2), the implementation automatically decrements SQL_DESC_COUNT to contain the number of the highest remaining bound record.

13.2.2 Bound Descriptor Records

When the application sets the SQL_DESC_DATA_PTR field of a descriptor record, so that it no longer contains the null value, the record is said to be *bound*.

If the descriptor is an application parameter descriptor, then each bound record constitutes a *bound parameter*.

- For input dynamic parameters, the application must bind a parameter for each dynamic parameter marker in the SQL statement before executing the statement (see Section 9.4 on page 102).
- For output dynamic parameters, the application need not bind the parameter. The application retrieves data from bound and unbound output dynamic parameters using different methods (see Section 9.4.3 on page 105).

If the descriptor is an application row descriptor, which describes a row of database data, then each bound record constitutes a *bound column*. The application retrieves data from bound and unbound columns using different methods (see Section 10.4 on page 133). (For methods of retrieving data from a row-set after a multi-row fetch, see Section 11.3 on page 156.)

5952 ## 13.3 Operations on Descriptors

5953 **Implicit Allocation/Freeing**

5954 When an application allocates a statement handle, the implementation implicitly allocates one
5955 set of four descriptors.[22] The application can obtain the handles of these implicitly-allocated
5956 descriptors as attributes of the statement handle. When the application frees the statement
5957 handle, the implementation frees all implicitly-allocated descriptors on that handle.

5958 **Explicit Allocation/Freeing**

5959 The application can explicitly allocate an application descriptor on a connection at any time it is
5960 actually connected to a database. By specifying that descriptor handle as an attribute of a
5961 statement handle using *SQLSetStmtAttr*(), the application directs the implementation to use that
5962 descriptor in place of the respective implicitly-allocated application descriptor. (The application
5963 cannot specify alternative implementation descriptors.)

5964 The application can associate an explicitly-allocated descriptor with more than one statement.
5965 The application can free such a descriptor explicitly, or implicitly by freeing its connection.

5966 **Obtaining a Descriptor Handle**

5967 The application obtains the handle of any explicitly-allocated descriptor as an output argument
5968 of the call to *SQLAllocHandle*(). The handle of an implicitly-allocated descriptor is available by
5969 calling *SQLGetStmtAttr*().

5970 **Initialisation of Fields**

5971 When an application row descriptor record is allocated, its fields receive initial values as
5972 specified in **Initialization of Descriptor Fields** on page 467. The initial value of the
5973 SQL_DESC_TYPE field is SQL_DEFAULT. This provides for a standard treatment of database
5974 data for presentation to the application (see **Cautions Regarding SQL_DEFAULT** on page 219).
5975 The application may specify different treatment of the data by setting fields of the descriptor
5976 record.

5977 The initial value of SQL_DESC_ARRAY_SIZE in the descriptor header is 1. The application can
5978 modify this field to enable multi-row fetch (see Section 11.1 on page 140.

5979 **Access to Fields**

5980 The application can call *SQLGetDescField*() to obtain a single field of a descriptor record.
5981 *SQLGetDescField*() gives the application access to all the descriptor fields defined in the X/Open
5982 **SQL** specification, and to other fields as well. *SQLGetDescField*() returns one field per call. The
5983 function is extensible, using additional argument values, to return future or implementation-
5984 defined fields.

5985 To modify fields of a descriptor, the application can call *SQLSetDescField*(), an extensible
5986 function that sets a single descriptor field per call. Some fields are read-only and cannot be set
5987 by *SQLSetDescField*(); refer to the table in the reference manual entry for *SQLSetDescField*().

5988 When setting fields individually, the application should follow the sequence defined in *[X-ref*
5989 *err*—**setdescfield**]. Setting some fields causes the XDBC implementation to set other fields.

5990 ________________

5991 22. The implementation has the option of deferring allocation of any descriptor until the point at which it is actually used.

5992 These cases, directly analogous to cases defined in the X/Open **SQL** specification, ensure that a
5993 descriptor is always ready to use once the application has specified a data type. When the
5994 application sets the SQL_DESC_DATA_PTR field, the implementation checks that other fields
5995 that specify the type are valid and consistent (see **Consistency Checks** on page 486).

5996 **Copying Descriptors**

5997 The *SQLCopyDesc*() function copies the fields of one descriptor to another descriptor. Fields can
5998 only be copied to an application descriptor or an implementation parameter descriptor, but not
5999 to an implementation row descriptor. Fields can be copied from either an application or an
6000 implementation descriptor. Only those fields that are defined for both the source and target
6001 descriptors are copied. *SQLCopyDesc*() does not copy the SQL_DESC_ALLOC_TYPE field,
6002 because a descriptor's allocation type cannot be changed. Copied values overwrite the existing
6003 values.

6004 An ARD on one statement handle can serve as the APD on another statement handle. This lets
6005 an application copy rows between tables without copying data at the application level. To do
6006 this, a row descriptor that describes a fetched row of a table is reused as a parameter descriptor
6007 for a parameter in an INSERT statement. The SQL_MAX_CONCURRENT_ACTIVITIES
6008 information item must be greater than 1 for this operation to succeed.

6009 **Freeing Handles**

6010 Explicitly allocated descriptors can be freed either explicitly by calling *SQLFreeHandle*() with a
6011 *HandleType* of SQL_HANDLE_DESC and the appropriate *Handle*, or implicitly when the
6012 connection handle is freed. When an explicitly-allocated descriptor is freed, all statement
6013 handles to which the freed descriptor applied automatically revert to the implicitly-allocated
6014 descriptors.

6015 Implicitly-allocated descriptors can only be freed by calling *SQLDisconnect*(), which drops any
6016 statements or descriptors open on the connection, or by calling *SQLFreeHandle*() with a
6017 *HandleType* of SQL_HANDLE_STMT to free a statement handle and all the implicitly-allocated
6018 descriptors associated with the statement. Implicitly-allocated descriptor handles cannot be
6019 freed by calling *SQLFreeHandle*() with a *HandleType* of SQL_HANDLE_DESC.

6020 13.3.1 Concise Functions

6021 Some XDBC functions gain implicit access to descriptors. Application writers may find them
6022 more convenient than calling *SQLSetDescField*() and *SQLGetDescField*(). Concise functions can
6023 be called without first retrieving a descriptor handle for use as an argument. The functions
6024 imply one or more descriptors based on a statement handle used as an argument.

6025 Some concise functions let an application set or retrieve several related descriptor fields in a
6026 single function call. Some concise functions perform more tasks than simply setting descriptor
6027 fields.

6028 The concise functions *SQLBindCol*() and *SQLBindParameter*() bind a column or parameter,
6029 respectively, by setting the descriptor fields that correspond to their arguments. These functions
6030 performs more tasks than simply setting descriptors. (The reference manual entries for these
6031 functions specify sequences of XDBC calls that are conceptually equivalent to calling
6032 *SQLBindCol*() and *SQLBindParameter*().) These functions completely specify the binding of a
6033 data column or dynamic parameter. However, an application can change individual details of a
6034 binding by calling *SQLSetDescField*() or *SQLSetDescRec*(), and can completely bind a column or
6035 parameter by making a series of suitable calls to these functions.

6036 The concise functions *SQLColAttribute*(), *SQLDescribeCol*(), *SQLDescribeParam*(),
6037 *SQLNumParams*(), and *SQLNumResultCols*() retrieve values in descriptor fields.

6038 *SQLSetDescRec*() and *SQLGetDescRec*() are concise functions that set or get multiple descriptor
6039 fields with one call. *SQLSetStmtAttr*() and *SQLGetStmtAttr*() serve as concise functions in some
6040 cases (see **Fields that Relate to Statement Attributes** on page 173).

13.4 Deferred Fields

The following fields 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 multi-row fetch, the SQL_DESC_ARRAY_STATUS_PTR, SQL_DESC_DIAG_INDEX_PTR and SQL_DESC_ROWS_PROCESSED_PTR fields of a descriptor header.

When the application sets these fields by calling *SQLSetDescField*(), the implementation does not use the current value of the application variables, but saves the addresses of the variables in the descriptor for a deferred effect as follows:

- For an application parameter descriptor, the implementation uses the contents of the variables at the time of the call to *SQLExecDirect*() or *SQLExecute*().
- For an application row descriptor, the application can set the SQL_DESC_ARRAY_STATUS_PTR, SQL_DESC_DIAG_INDEX_PTR and SQL_DESC_ROWS_PROCESSED_PTR fields in preparation for a multi-row fetch. The implementation assigns values to the SQL_DESC_DATA_PTR, SQL_DESC_INDICATOR_PTR and SQL_DESC_OCTET_LENGTH_PTR variables at the time of the fetch.

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 the null value, a multi-row fetch fails to return this component of the per-row diagnostic information (see **Diagnostic Messages** on page 201).
- If SQL_DESC_DATA_PTR has the null value, the record is unbound.
- If SQL_DESC_DIAG_INDEX_PTR has the null value, a multi-row fetch fails to return this component of the per-row diagnostic information (see **Diagnostic Messages** on page 201).
- If SQL_DESC_INDICATOR_PTR has the null value:
  - — For an application parameter descriptor, there is no indicator information for the descriptor record. For dynamic arguments, this prevents the application from using the buffer to specify null input dynamic arguments, and prevents the implementation from returning indicator information for output arguments.
  - — For an application row descriptor, a null SQL_DESC_INDICATOR_PTR prevents the implementation from returning indicator information for that column. (As in SQL, the implementation needs an indicator to report the fetch of a null value; in this case, failure to bind SQL_DESC_INDICATOR_PTR is an error.)
- If SQL_DESC_OCTET_LENGTH_PTR has the null value:
  - — For an application parameter descriptor, SQL_DESC_OCTET_LENGTH_PTR indicates the length in octets of character-string dynamic arguments. For input dynamic arguments, a null value directs the implementation to assume the string is null-terminated. For output dynamic arguments, a null value prevents the implementation from returning length information. (If TYPE does not indicate a character-string dynamic argument or character-string stored routine argument, SQL_DESC_OCTET_LENGTH_PTR is ignored.)
  - — For an application row descriptor, the implementation does not return length information for that column.

6085 The application can obtain the value of a deferred field by calling *SQLGetDescField*(). Such a call
6086 returns not the actual data but a pointer to the associated application variable.[23] A routine that
6087 takes as an argument a descriptor handle could call *SQLGetDescField*() to obtain pointers to the
6088 data, indicator or length of any descriptor record.

6089 Once the application has associated a deferred field with an application pointer, it can specify a
6090 different application pointer, or specify the null pointer to return the deferred field to the initial,
6091 unbound state. To reuse the same application descriptor with a different number and position of
6092 bound records, an application can free the descriptor and allocate a new one, or overwrite the
6093 previous bindings and change the SQL_DESC_COUNT field.

6094 The application must not deallocate or discard variables used for deferred fields between the
6095 time it associates them with the fields and the time the XDBC implementation reads or writes
6096 them.

6097 ———

6098 23. The call returns the null pointer if the field is not associated with an application variable. In the case of multi-row fetches,
SQL_DESC_DATA_PTR, SQL_DESC_INDICATOR_PTR and SQL_DESC_OCTET_LENGTH_PTR each point to an array whose
6099 cardinality is the value of SQL_DESC_ARRAY_SIZE (see Section 11.1 on page 140).

Chapter 14

Transactions

A *transaction* is a unit of work that is done as an single, atomic operation; that is, the operation succeeds or fails as a whole.

For example, consider a banking application that transfers money from one bank account to another. This involves two steps: withdrawing the money from the first account and depositing it in the second. The application requires 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* either by being *committed* or by 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 their state when the transaction began. 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. If either statement fails for any reason, the application rolls back the transaction. In both cases, the database is in a consistent state at the end of the transaction.

A single transaction can encompass multiple database operations, which 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. This is discussed in Section 14.3 on page 191.

14.1 Transaction Support in XDBC

Transactions in XDBC 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.

14.1.1 Determining Level of Support

The degree of support for transactions is implementation-defined. XDBC is designed to be implementable on a single-user or desktop database which 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.

If the implementation does not support transactions, but the application has the ability (using an API other than XDBC) to lock and unlock data, applications can achieve transaction isolation 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.

Some data sources that support transactions do not support more than one transaction at a time within an 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.

14.1.2 Commit Mode and Transaction Completion

An XDBC connection can be in either auto-commit mode or manual-commit mode.

Auto-commit Mode

In auto-commit mode, every database operation[24] 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 auto-commit mode, there is no way to specify that work be rolled back.

If the data source does not support auto-commit mode, the implementation can emulate it by explicitly committing each SQL statement as it is executed.

---

24. SELECT statements do not make any changes to the database and it is meaningless to commit them. SELECT statements open a cursor, through which operations such as DELETE, INSERT, and UPDATE can be performed. In auto-commit mode, these operations are the auto-committing transactions.

Auto-committing a Batch

When a batch is executed in auto-commit mode, it is implementation-defined which of the following is true:

- The entire batch is treated as an auto-commitable unit
- Each statement in a batch is treated as an auto-commitable unit.

Some data sources may support both these behaviors and may provide a way of selecting one or the other.

In particular, if an error occurs in the middle of the batch, it is implementation-defined whether statements already executed 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, the application must explicitly complete transactions by calling *SQLEndTran*(). Manual-commit mode is the usual method of working with most relational databases.

XDBC follows the model used in X/Open SQL in which the application does not explicitly initiate a transaction. Instead, a transaction begins implicitly whenever the application starts operating on the database.

If the data source requires explicit transaction initiation, the XDBC implementation must provide it whenever the application executes a statement requiring a transaction and there is no current transaction.

To achieve atomic completion encompassing XDBC database operations and other operations, on an implementation that complies both with XDBC and with the X/Open **TX** specification, the application delimits transactions by preceding all work with a call to *tx_begin*() and following it with a call to *tx_end*() (see the X/Open **TX** specification).

Setting the Commit Mode

Applications specify the transaction mode with the SQL_ATTR_AUTOCOMMIT connection attribute. By default, XDBC transactions are in auto-commit mode.[25] It is implementation-defined whether switching from manual-commit mode to auto-commit mode 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*().

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. They should instead call *SQLEndTran*().

If an application passes the environment handle to *SQLEndTran*() but does not pass a connection handle, the implementation conceptually calls *SQLEndTran*() for each active connection in the

---

25. An implementation's default for SQL_ATTR_AUTOCOMMIT may be incompatible with implementations complying with the March 1995 issue, because that issue did not specify a default.

6207 specified environment. **This calling mode does not imply the use of two-phase commit[26] to**
6208 **ensure atomicity across connections; it is merely a convenient alternative to calling**
6209 ***SQLEndTran*() once for each connections in the environment.**

6210 14.1.3 Side-effects of Transaction Completion

6211 It is implementation-defined which of the following is the case when a transaction is completed
6212 (committed or rolled back):

6213 • Cursors are closed and access plans for prepared statements are deleted.

6214 • Cursors are closed and access plans for prepared statements remain intact.

6215 • Cursors remain open and access plans for prepared statements remain intact.

6216 For example, suppose a data source exhibits the first behavior in the list above, and that an
6217 application does the following:

6218 1. Sets the commit mode to manual commit.

6219 2. Creates a result set of sales orders on statement 1.

6220 3. Creates a result set of the lines in a sales order on statement 2 when the user highlights that
6221 order.

6222 4. Calls *SQLExecute*() to execute a positioned UPDATE statement that has been prepared on
6223 statement 3 when the user updates a line.

6224 5. Calls *SQLEndTran*() to commit the positioned UPDATE statement.

6225 Because of the data source's behavior, the call to *SQLEndTran*() in step 5 causes it to close the
6226 cursors on statements 1 and 2 and to delete the access plan on statement 3. The application must
6227 reexecute statements 1 and 2 to recreate the result sets and reprepare the statement on statement
6228 3.

6229 In auto-commit mode, functions other than *SQLEndTran*() commit transactions:

6230 • *SQLExecute*() or *SQLExecDirect*(). In the previous example, the call to *SQLExecute*() in step 4
6231 commits a transaction. This causes the data source to close the cursors on statements 1 and 2
6232 and delete the access plan on statement 3.

6233 • *SQLBulkOperations*() or *SQLSetPos*(). In the previous example, suppose that in step 4 the
6234 application calls *SQLSetPos*() with the SQL_UPDATE option on statement 2 instead of
6235 executing a positioned update statement on statement 3. This commits a transaction and
6236 causes the data source to close the cursors on statements 1 and 2.

6237 • *SQLCloseCursor*(). In the previous example, suppose that, when the user highlights a
6238 different sales order, the application calls *SQLCloseCursor*() on statement 2 before creating a
6239 result of the lines for the new sales order. The call to *SQLCloseCursor*() commits the SELECT
6240 statement that created the result set of lines and causes the data source to close the cursor on
6241 statement 1.

6242 Applications, especially screen-based applications in which the user scrolls around the result set
6243 and updates or deletes rows, must be careful to code around this rather surprising behavior.

6244 ________________

6245 26. 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. If all data sources respond affirmatively to
6246 the poll, then the second phase commits the transaction on all data sources. If any data source responds negatively to the poll,
then the second phase rolls back the transaction on all data sources.

6247 To determine how a data source behaves when a transaction is commited or rolled back, an
6248 application calls *SQLGetInfo*() with the SQL_CURSOR_COMMIT_BEHAVIOR and
6249 SQL_CURSOR_ROLLBACK_BEHAVIOR options.

14.2 Transaction Isolation

Transaction isolation refers to the degree of interaction between multiple concurrent transactions. To see why this is important, we will first look at the idea of serializability.

14.2.1 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 in some serial order. 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 doubles a number and transaction B adds 1 to it. 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 the transaction A is run first or 2 and 22 if the 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.

14.2.2 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 (either through an update or 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 ISO SQL standard defines four transaction isolation levels 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 describes simple ways that a data source might implement the transaction isolation levels. (Most data sources use more complex schemes than these in order to increase concurrency. These examples are provided for illustrative purposes only. In particular, it is undefined how a particular data source isolates transactions from each other.)

Read Uncommitted

Transactions are not isolated from each other. If the data source supports other transaction isolation levels, it ignores whatever mechanism it uses to implement those levels. So that they don't adversely affect other transactions, transactions running at the Read Uncommitted level are usually read only.

Read Committed

The transaction waits until rows write-locked by other transactions are unlocked; this prevents it from reading any "dirty" data.

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.

Repeatable Read

The transaction waits until rows write-locked by other transactions are unlocked; this prevents it from reading any "dirty" data.

The transaction holds read locks on all rows it returns to the applicationand write locks on all rows it inserts, updates, or deletes. For example, if the transaction includes the SQL statement `SELECT * FROM Orders`, the transaction read-locks rows as the application fetches them. If the transaction includes the SQL statement `DELETE FROM Orders WHERE Status = 'CLOSED'`, the transaction write-locks rows as it deletes them.

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.

Serializable

The transaction waits until rows write-locked by other transactions are unlocked; this prevents it from reading any "dirty" data. 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 `SELECT * FROM Orders`, 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 `DELETE FROM Orders WHERE Status = 'CLOSED'`, the range is all rows with a Status of CLOSED; the transaction write-locks all rows in the Orders table with a Status of CLOSED

6343 and does not allow any rows to be inserted or updated such that the resulting row has a
6344 Status of CLOSED. Because other transactions cannot update or delete the rows in the
6345 range, the current transaction avoids any nonrepeatable reads.

6346 Because other transactions cannot insert any rows in the range, the current transaction
6347 avoids any phantoms. The transaction releases its lock when it is committed or rolled back.

6348 Transaction isolation never prevents a transaction from seeing its own changes. For example, a
6349 transaction might consist of two UPDATE statements, the first of which raises the pay of all
6350 employees by ten percent and the second of which sets the pay of any employees over some
6351 maximum amount to that amount. This succeeds as a single transaction only because the second
6352 UPDATE statement can see the results of the first.

6353 14.2.3 Setting the Transaction Isolation Level

6354 To set the transaction isolation level, an application uses the SQL_ATTR_TXN_ISOLATION
6355 connection attribute. If the data source does not support the requested isolation level, it can set a
6356 higher level. To determine what transaction isolation levels a data source supports and what the
6357 default isolation level is, an application calls *SQLGetInfo*() with the
6358 SQL_TXN_ISOLATION_OPTION and SQL_DEFAULT_TXN_ISOLATION options, respectively.

6359 Higher levels of transaction isolation offer the most protection for the integrity of database data.
6360 Serializable transactions are guaranteed to be unaffected by other transactions and therefore
6361 guaranteed to maintain database integrity.

6362 However, a higher level of transaction isolation can cause slower performance because it
6363 increases the chances that the application will have to wait for locks on data to be released. An
6364 application may specify a lower level of isolation in order to increase performance in the
6365 following cases:

6366 • When it can be guaranteed that no other transactions exist that might interfere with an
6367 application's transactions. This situation occurs only in limited circumstances, such as when
6368 one person in a small company maintains files and does not share them.

6369 • When speed is more critical than accuracy and any errors are likely to be inconsequential.
6370 For example, suppose that a company makes many small sales and that large sales are rare. A
6371 transaction that estimates the total value of all open sales might safely use the Read
6372 Uncommitted isolation level. Although the transaction would include orders in the process of
6373 being opened or closed that are subsequently rolled back, these would tend to cancel each
6374 other out and the transaction would be faster because it is not blocked each time it
6375 encounters such an order.

6376 See also Section 14.3.2 on page 192.

6377 14.2.4 Scrollable Cursors and Transaction Isolation

6378 One of the distinguishing characteristics of a certain type of scrollable cursor — static, keyset-
6379 driven, or dynamic — is its ability to detect changes made by other operations in the same
6380 transaction and by other transactions. Because the transaction isolation level also determines
6381 what changes are visible to the cursor, it seems fair to ask what the relationship is between these
6382 two.

6383 The answer is simple. The transaction isolation level dictates what changes in other transactions
6384 might be visible to the cursor while the cursor type dictates which of those changes are actually
6385 visible. For example, suppose the transaction containing the cursor is running at the Read
6386 Committed isolation level: Committed changes made by other transactions are visible to the
6387 cursor's transaction. However, the cursor sees these only if it is a keyset-driven or dynamic
6388 cursor; if it is a static cursor, it can't see any changes at all.

Note that the transaction isolation level does not affect a cursor's ability to see its own changes — those made with positioned UPDATE or DELETE statements or through *SQLSetPos*() — or those made by other operations in the same transaction. Whether the cursor can see its own changes depends on the cursor type and how it is implemented. Whether the cursor can see changes made by other operations in the same transaction depends on the cursor type. For more information, see Section 11.2 on page 147.

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

This is shown in the following diagram:

Cursor's Transaction

Visibility of changes made here is governed by cursor type.

Cursor

Visibility of changes made here is governed by the cursor type and the implementation of the cursor.

Other Transactions

Visibility of changes made here is governed by the cursor type and the transaction isolation level.

Transaction isolation works at this boundary.

Depending on the application, certain combinations of cursor type and transaction isolation level do not make sense. For example, suppose an online telephone book uses a dynamic cursor to read and display telephone numbers and that a separate application is used to maintain the database of telephone numbers. To be effective, the cursor used to read telephone numbers needs to detect all committed changes to the database. If the transaction containing this cursor is run at the Repeatable Read or Serializable isolation level, the cursor will detect few or no changes and is essentially a slow, expensive static cursor. Instead, the transaction containing the cursor should be run at the Read Committed isolation level.

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.

		Own	**Other transactions** [a]			
	Self	**Transaction**	**Read Unc.**	**Read Com.**	**Repeatable**	**Serializable**

Static	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 legends here indicate the four transaction isolation levels; see Section 14.2.2 on page 186.

[b] It depends on how the cursor is implemented. The application can determine whether various types of cursor can detect such changes by calling *SQLGetInfo*() as described in **Detecting Cursor Capabilities with SQLGetInfo()** on page 402.

14.3 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 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.

This is a problem even if the cursor is read-only and the application only lets the user read existing orders. In this case, the application commits the transaction — and releases locks — when it calls *SQLCloseCursor*() (in manual commit mode) or *SQLEndTran*() (in auto-commit mode).

14.3.1 Concurrency Types

To solve the problem of reduced concurrency in cursors, XDBC exposes four different types of cursor concurrency:

- **Read only**

  The cursor can only read data but cannot update or delete data. This is the default concurrency type. Although the data source 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 ensure that 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.

An application specifies what type of concurrency it wants the cursor to use with the SQL_ATTR_CONCURRENCY statement attribute.

14.3.2 Optimistic Concurrency

Optimistic concurrency derives its name from the optimistic assumption that collisions between transactions 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, which uses the assumption 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 a row version or the values of each column in the row. Some data sources do not support row versions.

Optimistic concurrency can be implemented by the XDBC implementation or by 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 XDBC implementation, 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 implementation 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 been 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

Diagnostics

Functions in XDBC return diagnostic information in the following ways:

- The return code (see Section 15.1 on page 194) indicates the overall success or failure of the function
- Diagnostic records (see Section 15.2 on page 195 provide detailed information about the function. The diagnostics area may contain information about multiple diagnostic events associated with a function invocation.
- SQLSTATE (see Section 15.3 on page 196 is a five-character standardized error code.

Section 15.4 on page 200 provides information on how applications use the above diagnostic information.

Diagnostic information is used at development time to catch programming errors such as invalid handles. It is used at run time to catch run time errors and warnings such as data truncation, access violations, and errors in the execution of SQL statements.

6532 ## 15.1 Return Codes

6533 Each function in XDBC returns a code, known as its *return code*, that indicates the overall success
6534 or failure of the function.

6535 XDBC defines the following return codes:

6536 SQL_SUCCESS
6537 Function completed successfully. The application can call *SQLGetDiagField*() to retrieve
6538 additional information from the header record.

6539 SQL_SUCCESS_WITH_INFO
6540 Function completed successfully, possibly with a nonfatal error (warning). The application
6541 can call *SQLGetDiagRec*() or *SQLGetDiagField*() to retrieve additional information.

6542 SQL_ERROR
6543 Function failed. The application can call *SQLGetDiagRec*() or *SQLGetDiagField*() to retrieve
6544 additionalinformation.

6545 SQL_INVALID_HANDLE
6546 Function failed due to an invalid environment, connection, statement, or descriptor handle.
6547 This indicates a programming error. No additional information is available from
6548 *SQLGetDiagRec*() or *SQLGetDiagField*(). This code is only returned when the handle is a null
6549 pointer or is the wrong type, such as when a statement handle is passed for an argument
6550 that requires a connection handle.

6551 SQL_NO_DATA
6552 No more data was available. The application can call *SQLGetDiagRec*() or *SQLGetDiagField*()
6553 to retrieve additional information. One or more implementation-defined status records in
6554 class 02xxx may be returned.

6555 SQL_NEED_DATA
6556 More data is needed, such as when parameter data is sent at execution time or additional
6557 connection information is required. The application can call *SQLGetDiagRec*() or
6558 *SQLGetDiagField*() to retrieve additional information, if any.

6559 SQL_STILL_EXECUTING
6560 A function that was started asynchronously is still executing. The application can call
6561 *SQLGetDiagRec*() or *SQLGetDiagField*() to retrieve additional information, if any.

6562 The return code SQL_INVALID_HANDLE always indicates a programming error and should
6563 never be encountered at run time. All other return codes provide run-time information,
6564 although SQL_ERROR may indicate a programming error.

15.2 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 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. These fields are different for the header record and the status records. In addition, XDBC components can define their own diagnostic record fields.

The stored format of the diagnostic data structure is undefined.

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 *SQLGetDiagField*().

Status Records

The fields in the status records contain information about specific errors or warnings, 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, or SQL_NEED_DATA. For a complete list of fields in the status records, see *SQLGetDiagField*().

15.3 SQLSTATE

The SQLSTATE code is a five-character, standardized diagnostic code. that provides detailed information about the cause of a warning or error.

The first two characters are the class and the final three characters are the subclass. In many cases, applications need only consider the class code and do not need the specific information provided by the subclass. (See Appendix A for a list of XSQL SQLSTATE values with cross-references. For conditions under which a specific XDBC function may return a SQLSTATE value, see the **DIAGNOSTICS** section of the reference manual pages.)

The format, values, and usage of SQLSTATE are the same as defined in the ISO SQL standard and the X/Open **SQL** specification, in diagnostic conditions that also occur in SQL.

The X/Open **SQL** specification (SQLSTATE Status Variable) reserves all class and subclass codes starting with 0-4 or A-H for definition by an international standard.

The X/Open **SQL** specification (SQLSTATE Values) is the authoritative reference for a description of each SQLSTATE code. Using XDBC to execute SQL text whose syntax or usage violates the X/Open **SQL** specification produces the error code specified by the X/Open **SQL** specification.

XDBC defines addition SQLSTATE values in two classes:

HY XDBC-specific codes.[27] Their definition in XDBC is consistent with that in the ISO CLI International Standard. Implementation-defined errors pertaining to the application's use of XDBC specify class HY and subclasses 500 to 9ZZ inclusive and I00 to ZZZ inclusive. (X/Open reserves subclass codes S00 to SZZ inclusive.)

IM XDBC-specific codes reporting errors specific to a data source.

Sequence of Status Records

Status records are first sorted by the SQL_DIAG_ROW_NUMBER diagnostic field, then sorted according to the ranking of the SQLSTATE code, as described below.

Sorting by Row Number

In diagnostics that pertain to a multi-row fetch, the sequence of the records is determined first by row number. The following rules determine the sequence of errors by row:

- Records for which the row number is unknown appear in front of all other records, because SQL_ROW_NUMBER_UNKNOWN is defined to be −1.
- Records that do not correspond to any row appear in front of records that correspond to a particular row, because SQL_NO_ROW_NUMBER is defined to be 0.
- For all records that pertain to specific rows, records are sorted by the value in the SQL_DIAG_ROW_NUMBER field. Diagnostics pertaining to the first row affected are listed, then diagnostics pertaining to the next row affected, and so on.

Ranking by Severity

Within a row, or for all those records that do not correspond to a row or for which the row number is unknown, or for diagnostics that do not apply to a multi-row fetch, the first record in the diagnostics area is the record with the highest rank according to the following rules:

27. Standards organizations have begun using the classes at the end of their reserved range for standards adopted since the adoption of the ISO SQL standard. The class HZ was assigned to remote database access errors.

- **Errors**

  Status records that describe errors have the highest rank. Among error records, the following rules are followed to sort errors:

  — Records that indicate or suggest a transaction failure outrank all other records.

  — If two or more records describe the same error condition, then SQLSTATEs defined in the ISO CLI International Standard (classes 03 through HZ) outrank XDBC- and implementation-defined SQLSTATEs.

- **Implementation-defined No Data values**

  Status records that describe implementation-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 SQLSTATEs defined in the ISO CLI International Standard outrank XDBC- and implementation-defined SQLSTATEs.

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, warnings may appear before errors. Applications should check all status records when a function returns a value other than SQL_SUCCESS.

The software component that generated a record is not relevant to its rank. If the XDBC implementation collects diagnostic information from several sources, it assembles the information so as to comply with the above rules.

Implementation Variability

Unlike return codes, the SQLSTATEs in this manual are guidelines; implementations are not required to return them. Thus, while implementations 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 diagnostics and their possible causes, it is not complete and probably never will be; implementations simply vary too much for this to ever occur. Thus, any given implementation probably won't 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 thousands of diagnostics. The implementations for such engines are unlikely to map all of these diagnostics 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, implementations should map as many diagnostics as seems reasonable and be sure to map those diagnostics on which application logic might be based, such as SQLSTATE 01004 (Data truncated).

Most applications react to an error by simply displaying the SQLSTATE, diagnostic message text, and the native error code. This is often sufficient; for example, when the application submits SQL statements typed by the user, a typical error based on SQL statement failure cannot be corrected by the application. Instead, the user must edit or re-type the SQL statement, assisted by the knowledge of which error occurred.

Any application that bases its logic on SQLSTATEs 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 implementations. However, a general guideline is that SQLSTATEs for errors that occur in the XDBC implementation, as opposed to the data source, are more likely to be returned reliably. For example, most implementations probably return SQLSTATE HYC00 (Optional feature not implemented) while fewer implementations probably return SQLSTATE 42S21 (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 implementations return them.

- 01004 (Data truncated)
- 01S02 (Attribute value changed)
- HY008 (Operation canceled)
- HY010 (Function sequence error)
- HYC00 (Optional feature not implemented)
- HYT00 (Timeout expired)

It is particularly desirable for an application to detect SQLSTATE HYC00 (Optional feature not implemented), because it is the only way the application can determine whether a data source supports a particular statement or connection attribute.

Diagnostic Messages

A diagnostic message is returned with each SQLSTATE.

The SQLSTATEs in this specification are accompanied by a sample diagnostic message. This text is not normative. Implementations are not required to return these messages. Implementations typically pass through to the application whatever message the data source provides.

Moreover, it is not mandatory that diagnostic messages be consistent within a given SQLSTATE value. For example, in the case of SQLSTATE 42000 (Syntax error or access violation), implementations are not required to return the diagnostic message in parentheses, and are more likely to return a variety of messages that are more specific.

Applications may 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. **Application logic should never be based on the specific text of a diagnostic message.**

Diagnostic messages come from data sources and other software components in an XDBC connection. Typically, data sources do not directly support XDBC. Consequently, if a component in an XDBC 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 a diagnostic is a component itself, the diagnostic message must explain this. Therefore, the text of messages has two different formats. Brackets ([ ]) in the following formats do not indicate optionality but must appear in the message.

Messages for diagnostics that do not occur in a data source use this format:

[*vendor-identifier*][*XDBC-component-identifier*]
component-supplied-text

Messages for diagnostics that occur in a data source use this format:

[*vendor-identifier*][*XDBC-component-identifier*]
[*data-source-identifier*]*data-source-supplied-text*

6719 The components in these messages are defined as follows:

6720 *vendor-identifier*
6721 Identifies the vendor of the component in which the error or warning occurred or that
6722 received the error or warning directly from the data source.

6723 *XDBC-component-identifier*
6724 Identifies the component in which the error or warning occurred or that received the error
6725 or warning directly from the data source.

6726 *data-source-identifier*
6727 Identifies the data source. For file-based data sources, this is typically a file format, such as
6728 Xbase.[28] For other data sources, this is the data source product.

6729 *component-supplied-text*
6730 Generated by the XDBC component.

6731 *data-source-supplied-text*
6732 Generated by the data source.

6733 ________________
6734 28. In this case, the driver is acting as both the driver and the data source.

6735 ## 15.4 Application Usage

6736 Program logic is generally based on return codes.

6737 For example, the following code calls *SQLFetch*() to retrieve the rows in a result set. It checks the
6738 return code of the function to determine if the end of the result set was reached
6739 (SQL_NO_DATA), if any warning information was returned (SQL_SUCCESS_WITH_INFO), or
6740 if an error occurred (SQL_ERROR).

```
6741 SQLRETURN rc;
6742 while ((rc=SQLFetch(hstmt) != SQL_NO_DATA) {
6743    if (rc == SQL_SUCCESS_WITH_INFO) {
6744       // Call function to display warning information.
6745    } else if (rc == SQL_ERROR) {
6746       // Call function to display error information.
6747       break;
6748    }
6749    // Process row.
6750 }
```

6751 Applications call *SQLGetDiagRec*() or *SQLGetDiagField*() to retrieve diagnostic information.
6752 These functions accept an environment, connection, statement, or descriptor handle and return
6753 diagnostics from the function that last used that handle. The diagnostics logged on a particular
6754 handle are discarded when a new function is called using that handle. If the function returned
6755 multiple diagnostic records, the application calls these functions multiple times; the total
6756 number of status records is retrieved by calling *SQLGetDiagField*() for the header record (record
6757 0) with the SQL_DIAG_NUMBER option.

6758 Applications retrieve individual diagnostic fields by calling *SQLGetDiagField*() and specifying
6759 the field to retrieve. Certain diagnostic fields do not have any meaning for certain types of
6760 handle; see *SQLGetDiagField*() for more information. For a list of diagnostic fields and their
6761 meaning, see *SQLGetDiagField*().

6762 Applications can retrieve the SQLSTATE, native error code, and diagnostic message in a single
6763 call by calling *SQLGetDiagRec*(). This function does not retrieve information from the header
6764 record.

6765 For example, the following code prompts the user for an SQL statement and executes it. If any
6766 diagnostic information was returned, it calls *SQLGetDiagField*() to get the number of status
6767 records and *SQLGetDiagRec*() to get the SQLSTATE, native error code, and diagnostic message
6768 from those records.

```
6769 SQLCHAR    SqlState[6], Msg[SQL_MAX_MESSAGE_LENGTH - 1];
6770 SQLINTEGER i, NativeError, MsgLen;
6771 SQLRETURN  rc1, rc2;

6772 // Prompt the user for an SQL statement.
6773 GetSQLStmt(SQLStmt);

6774 // Execute the SQL statement and return any errors or warnings.
6775 rc1 = SQLExecDirect(hstmt, SQLStmt, SQL_NTS);
6776 if ((rc1 == SQL_SUCCESS_WITH_INFO) || (rc1 == SQL_ERROR)) {

6777    // Get the status records.
6778    i = 1;
6779    while ((rc2 = SQLGetDiagRec(SQL_HANDLE_STMT, hstmt, i, SqlState,
6780                                &NativeError, Msg, sizeof(Msg),
6781                                &MsgLen)) != SQL_NO_DATA) {
6782       DisplayError(SqlState,NativeError,Msg,MsgLen);
```

```
            i++;
        }
    }
    if ((rc1 == SQL_SUCCESS) || (rc1 == SQL_SUCCESS_WITH_INFO)) {
        // Process statement results, if any.
    }
```

15.4.1 Per-row Diagnostics

A multi-row fetch (see Section 11.1 on page 140) uses two arrays to disclose to applications the diagnostic status of individual rows. The implementation row descriptor header has two deferred fields, SQL_DESC_ARRAY_STATUS_PTRand SQL_DESC_DIAG_INDEX_PTR, that the application can bind to arrays it has allocated.

A multi-row fetch describes the outcome of fetching each row of the row-set by setting the corresponding element of the arrays pointed to by SQL_DESC_ARRAY_STATUS_PTR and of SQL_DESC_DIAG_INDEX_PTR. The element of SQL_DESC_ARRAY_STATUS_PTR contains one of the following:

SQL_ROW_SUCCESS
: If the row was fetched and populated without errors or warnings.

SQL_ROW_SUCCESS_WITH_INFO
: If the row was fetched and populated but there is a warning associated with the row.

SQL_ROW_ERROR
: If there was an error fetching or populating the row. The contents of the corresponding row buffers is undefined.

SQL_ROW_NOROW
: If the row could not be fetched because it was before the start or after the end of a partial row-set.

The element of SQL_DESC_DIAG_INDEX_PTR contains the following:

- For any element of SQL_DESC_ARRAY_STATUS_PTRthat contains SQL_ROW_SUCCESS or SQL_ROW_NOROW, the corresponding element of SQL_DESC_DIAG_INDEX_PTR contains 0 to indicate that there is no per-row diagnostic information for this row.
- For any element of SQL_DESC_ARRAY_STATUS_PTR that contains SQL_ROW_SUCCESS_WITH_INFO or SQL_ROW_ERROR, the corresponding element of SQL_DESC_DIAG_INDEX_PTR contains the record number of the first diagnostic record that pertains to that row. Successive diagnostic records may also pertain to that row.

Every diagnostic record contains a field SQL_DIAG_ROW_NUMBER. For diagnostics produced during a multi-row fetch, this field specifies the row number to which the diagnostic pertains. The first row of the multi-row fetch is row number 1. The SQL_DIAG_ROW_NUMBER field is a cross-reference back to the two arrays defined above. It also lets the application obtain complete per-row diagnostic information using the following algorithm.

6821 **Example Application Algorithm**

6822 The application can obtain complete diagnostic information on a multi-row fetch using the
6823 following procedure, which depends on the return code of the fetch function:

6824 [SQL_SUCCESS]
6825 Inspect the value pointed to by the SQL_DESC_ROWS_PROCESSED_PTR field in the
6826 implementation row descriptor header to determine how many rows were fetched.

6827 [SQL_SUCCESS_WITH_INFO]
6828 Inspect each element in the array pointed to by the SQL_DESC_ARRAY_STATUS_PTRfield
6829 in the header of the implementation row descriptor. For any element that contains
6830 SQL_ROW_SUCCESS_WITH_INFO or SQL_ROW_ERROR:

6831 • Get the value of the corresponding element of the array pointed to by
6832 SQL_DESC_DIAG_INDEX_PTR. Let this value be *i*.

6833 • Call *GetDiagField*() with *RecNumber* = *i* to obtain information from the first status record
6834 that pertains to this row position.

6835 • Continue to call *GetDiagField*() for incremented values of *i* until either the
6836 SQL_DIAG_ROW_NUMBER diagnostic field indicates that the status record pertains to
6837 a different row, or *GetDiagField*() returns [SQL_NO_DATA], indicating that there are no
6838 more status records.

6839 Truncation can also be detected by examining the column's length or indicator information.

6840 [SQL_ERROR]
6841 Call *GetDiagField*() for more information on an error that pertains to the entire fetch.

6842 *Chapter 20*

6843
Interface Overview

6844 This chapter lists the XDBC functions and gives a section reference for specific overview
6845 information. For detailed information, see the appropriate reference manual page, which
6846 appears in alphabetical order in Chapter 21.

6847 Table **20-1.** XDBC Functions

XDBC Function	Description	Overview Section
Allocate and Deallocate		
SQLAllocHandle()	Allocate memory for environment, connection, statement, or descriptor handles.	Section 4.1 on page 34
SQLFreeHandle()	Free resources associated with a specific handle.	Section 4.1 on page 34
SQLFreeStmt()	Stop processing associated with a specific statement, close any open cursors associated with the statement, or discard pending results.	Section 9.6 on page 124
Get and Set Attributes		
SQLGetConnectAttr()	Return the current setting of a connection attribute.	Section 4.5 on page 49
SQLGetEnvAttr()	Return the current setting of an environment attribute.	Section 4.5 on page 49
SQLGetStmtAttr()	Return the current setting of a statement attribute.	Section 9.2 on page 93
SQLSetConnectAttr()	Set attributes that govern aspects of connections.	Section 4.5 on page 49
SQLSetEnvAttr()	Set attributes that govern aspects of environments.	Section 4.5 on page 49
SQLSetStmtAttr()	Set attributes related to a statement.	Section 9.2 on page 93
Connection		
SQLBrowseConnect()	Iterative method of discovering and enumerating the attributes and attribute values required to connect to a data source.	Section 6.4.5 on page 62
SQLConnect()	Establish connections to a data source.	Section 6.4.2 on page 61
SQLDriverConnect()	Connect to a data source using implementation-defined interaction with the user.	Section 6.4.4 on page 62
SQLDrivers() OP	List driver descriptions and driver attribute keywords.	Section I.2 on page 617
SQLDisconnect()	Close the connection associated with a specific connection handle.	Section 6.5 on page 64

XDBC Function	Description	Overview Section
Descriptor Access		
SQLColAttributes()	Return descriptor information for a column in a result set.	Section 13.3.1 on page 176
SQLCopyDesc()	Copy descriptor information from one descriptor handle to another.	Section 13.3.0 on page 176
SQLGetDescField()	Return the current settings of a single field of a descriptor record.	Chapter 13
SQLGetDescRec()	Return the current settings of multiple fields of a descriptor record.	Section 13.3.1 on page 176
SQLSetDescField()	Set the value of a single field of a descriptor record.	Chapter 13
SQLSetDescRec()	Set multiple descriptor fields.	Section 13.3.1 on page 176
Executing SQL Statements		
SQLBindParam() DE *SQLBindParameter*()	Bind a buffer to a parameter marker in an SQL statement.	Section 9.4.1 on page 102
SQLExecDirect()	Execute a preparable statement, using the current values of the parameter marker variables if any parameters exist in the statement.	Section 9.3 on page 94
SQLExecute()	Execute a prepared statement, using the current values of the parameter marker variables if any parameter markers exist in the statement.	Section 9.3 on page 94
SQLGetCursorName()	Return the cursor name associated with a specified statement.	Section 12.1 on page 158
SQLParamData()	Supply parameter data at statement execution time.	Section 9.4.3 on page 105
SQLPrepare()	Prepare an SQL statement for execution.	Section 9.3.2 on page 96
SQLPutData()	Supply data for a parameter or column at statement execution time.	Section 9.4.3 on page 105
SQLSetCursorName()	Set the name of a cursor.	Section 12.1 on page 158
Call-level Database Access		
SQLBulkOperations()	Perform bulk insertions and bulk bookmark operations, including update, delete, and fetch by bookmark. T}‘T{ Section 12.4 on page 165	
SQLSetPos()	Set the cursor position in a row-set and refresh, update, or delete data in the result set.	
Function Cancellation		
SQLCancel()	Cancel the processing of a statement.	

XDBC Function	Description	Overview Section
Receiving Results		
SQLBindCol()	Bind application data buffers to columns in the result set.	Section 10.3.2 on page 130
SQLCloseCursor()	Close a cursor that has been opened on a statement, discarding pending results.	Section 10.5 on page 137
SQLDescribeCol()	Return the result descriptor for one column in the result set.	Section 13.3.1 on page 176
SQLDescribeParam()	Return the description of a parameter marker associated with a prepared SQL statement.	Section 13.3.1 on page 176
SQLFetch()	Fetch the next row-set of data from the result set and return data for all bound columns.	Section 10.4.2 on page 133
SQLFetchScroll()	Fetch the specified row-set of data from the result set and return data for all bound columns.	Chapter 11
SQLGetData()	Retrieve data for a single column in the result set.	
SQLMoreResults()	Determine whether there are more results available on a statement containing SELECT, UPDATE, INSERT, or DELETE statements and, if so, initialize processing for those results.	Section 11.3 on page 156
SQLNumParams()	Return the number of parameters in an SQL statement.	Section 13.3.1 on page 176
SQLNumResultCols()	Return the number of columns in a result set.	Section 13.3.1 on page 176
SQLRowCount()	Return the number of rows affected by certain database operations.	Section 13.3.1 on page 176
Catalog Functions		
SQLColumnPrivileges()	Return a list of columns and associated privileges for the specified table as a result set.	Chapter 7
SQLColumns()	Return the list of column names in specified tables as a result set.	Chapter 7
SQLForeignKeys()	Return a list of foreign keys for a specified table.	Chapter 7
SQLPrimaryKeys()	Return as a result set the column names of the primary key of a table.	Chapter 7
SQLProcedureColumns()	Return as a result set the list of input and output parameters, and the columns of the result set, for the specified procedures.	Chapter 7
SQLProcedureColumns()	Return the list of procedure names stored in a specified data source.	Chapter 7

XDBC Function	Description	Overview Section
Catalog Functions (continued)		
SQLSpecialColumns()	Retrieve information about row-identifying columns of a table.	Chapter 7
SQLStatistics()	Retrieve as a result set a list of statistics about a single table and the indexes associated with it.	Chapter 7
SQLTablePrivileges()	Return as a result set a list of tables and the privileges associated with each table.	Chapter 7
SQLTables()	Return as a result set the list of table, catalog, or schema names, and table types, stored in a specified data source.	Chapter 7
Introspection		
SQLDataSources()	Return information about a data source.	
SQLGetFunctions()	Indicate the level of support for a specified XDBC function.	
SQLGetInfo()	Return general information about the data source and the connection to it.	
SQLGetTypeInfo()	Return information about data types supported by the data source.	Section 4.4.2 on page 46
SQLNativeSql()	Return the text of a specified SQL statement as modified by the implementation, without executing the statement.	Section 8.2 on page 80
Transaction Control		
SQLEndTran()	Request commit or rollback of all active operations on all statements associated with a connection, or for all connections associated with an environment.	Chapter 14
Diagnostic Information		
SQLGetDiagField()	Return the current value of a field of a diagnostic data structure that contains error, warning, and status information.	Chapter 15
SQLGetDiagRec()	Return the current values of multiple fields of a diagnostic record.	Chapter 15

Chapter 21

Reference Manual Pages

The following pages describe each XDBC function in alphabetic order. Each function is defined as a C programming language function. Descriptions include the following:

- **Source of the function**

  This text appears at the center of each page. The label is one or more of the following:

  ISO 92
  : The function appears here based on its definition in the ISO CLI International Standard.

  X/Open CLI
  : The function appears here based on its definition in the March 1995 issue of the X/Open **Call Level Interface (CLI)** specification.

  XDBC
  : The function is published here for the first time in any X/Open specification and is not in the ISO CLI International Standard.

- **NAME**

  The function name and a brief summary of its effects.

- **SYNOPSIS**

  A sample C-language declaration of the function. The parameter names used in this declaration are also used throughout the entry to refer to the respective parameters.

- **ARGUMENTS**

  The argument to be supplied for each function parameter.

- **RETURN VALUES**

  The values the XDBC function can return. The valid return values are listed and described in Section 15.1 on page 194.

- **DIAGNOSTICS**

  The entire list of possible XDBC-defined errors and warnings that the function can report, sorted by SQLSTATE values. For a cross-reference of all SQLSTATE values, listing the functions that return each, see Appendix A.

  For information on handling diagnostic information, see *SQLGetDiagField*(). The text associated with SQLSTATE values is included to provide a description of the condition, but is not intended to prescribe specific text.

- **COMMENTS**

  A description of the function, including comments about usage and implementation.

- **SEE ALSO**

  References to related functions.

7038 **NAME**

7039 SQLAllocHandle — Allocate memory for environment, connection, statement, or descriptor
7040 handles.

7041 **SYNOPSIS**

```
7042 SQLRETURN SQLAllocHandle(
7043   SQLSMALLINT HandleType,
7044   SQLHANDLE InputHandle,
7045   SQLHANDLE * OutputHandlePtr);
```

7046 **ARGUMENTS**

7047 *HandleType* [Input]
7048 The type of handle to be allocated by *SQLAllocHandle*(). Must be one of the following
7049 values:

7050 SQL_HANDLE_ENV
7051 SQL_HANDLE_DBC
7052 SQL_HANDLE_STMT
7053 SQL_HANDLE_DESC

7054 *InputHandle* [Input]
7055 The handle that describes the data structure in whose context the new data structure is to be
7056 allocated. If *HandleType* is SQL_HANDLE_ENV, this is SQL_NULL_HANDLE. If
7057 *HandleType* is SQL_HANDLE_DBC, this must be an environment handle, and if it is
7058 SQL_HANDLE_STMT or SQL_HANDLE_DESC, it must be a connection handle.

7059 *OutputHandlePtr* [Output]
7060 Pointer to a buffer in which to return the handle to the newly allocated data structure.

7061 **RETURN VALUE**

7062 SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_INVALID_HANDLE, or SQL_ERROR.

7063 When allocating a handle other than an environment handle, if *SQLAllocHandle*() returns
7064 SQL_ERROR, it sets *OutputHandlePtr* to SQL_NULL_HDBC, SQL_NULL_HSTMT, or
7065 SQL_NULL_HDESC, depending on the value of *HandleType*, unless the output argument is a
7066 null pointer. The application can then obtain additional information from the diagnostic data
7067 structure associated with the handle in *InputHandle*. **Environment Handle Allocation Errors** If
7068 the implementation cannot allocate memory for **OutputHandlePtr* when *SQLAllocHandle*() with
7069 a *HandleType* of SQL_HANDLE_ENV is called, or the application provides a null pointer for
7070 *OutputHandlePtr*, *SQLAllocHandle*() returns SQL_ERROR. The implementation sets
7071 **OutputHandlePtr* to SQL_NULL_HENV (unless the application provided a null pointer). There
7072 is no handle with which to associate additional diagnostic information.

7073 **DIAGNOSTICS**

7074 When *SQLAllocHandle*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated
7075 SQLSTATE value can be obtained by calling *SQLGetDiagRec*() with the appropriate *HandleType*
7076 and *Handle* set to the value of *InputHandle*. SQL_SUCCESS_WITH_INFO (but not SQL_ERROR)
7077 can be returned for *OutputHandle*. The following SQLSTATE values are commonly returned by
7078 *SQLAllocHandle*(). The return code associated with each SQLSTATE value is SQL_ERROR,
7079 except that for SQLSTATE values in class 01, the return code is SQL_SUCCESS_WITH_INFO.

7080 01000 — General warning
7081 Implementation-defined informational message.

7082 08003 — Connection does not exist
7083 *HandleType* was SQL_HANDLE_STMT or SQL_HANDLE_DESC, but the connection
7084 specified by *InputHandle* was not open. The connection process must be completed
7085 successfully (and the connection must be open) to allocate a statement or descriptor handle.

HY000 — General error
An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by *SQLGetDiagRec*() in the **MessageText* buffer describes the error and its cause.

HY001 — Memory allocation error
The implementation failed to allocate memory for the specified handle.

HY009 — Invalid use of null pointer
OutputHandlePtr was a null pointer.

HY013 — Memory management error
The HandleType argument was SQL_HANDLE_DBC, SQL_HANDLE_STMT, or SQL_HANDLE_DESC; and the function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.

HY014 — Limit on the number of handles exceeded
The implementation-defined limit for the number of handles that can be allocated for the type of handle indicated by *HandleType* has been reached.

HY092 — Invalid attribute identifier
HandleType was not: SQL_HANDLE_ENV, SQL_HANDLE_DBC, SQL_HANDLE_STMT, or SQL_HANDLE_DESC.

HYT01 — Connection timeout expired
The connection timeout period expired before the data source responded to the request. The connection timeout period is set through *SQLSetConnectAttr*(), SQL_ATTR_CONNECTION_TIMEOUT.

IM001 — Function not supported
The function is not supported on the current connection to the data source.

COMMENTS

SQLAllocHandle() allocates handles for environments, connections, statements, and descriptors.

It is implementation-defined how many environment, connection, and statement handles an application can allocate at a time. The application can determine these limits by calling *SQLGetInfo*() with one of the following options: SQL_ACTIVE_ENVIRONMENTS, SQL_MAX_DRIVER_CONNECTIONS, or SQL_MAX_CONCURRENT_ACTIVITIES (for statements). An attempt to allocate more than the supported number of environments, connections, or statements produces SQLSTATE HY014 (Limit on the number of handles exceeded). There is no limit on the number of descriptor handles that can be allocated.

If the application calls *SQLAllocHandle*() with **OutputHandlePtr* set to a handle already in use, the implementation typically overwrites information associated with the handle. The implementation need not check to see whether **OutputHandlePtr* is already in use, nor check the previous contents of a handle before overwriting them.

On operating systems that support multiple threads, applications can use the same environment, connection, statement, or descriptor handle on different threads. Implementations must therefore support safe, multithreaded access to this information, for example, through the use of a critical section or a semaphore.

7127 **Allocating an Environment Handle**

7128 An environment handle provides access to global information such as valid connection handles
7129 and active connection handles. To request an environment handle, an application calls
7130 *SQLAllocHandle*() with *HandleType* of SQL_HANDLE_ENV and *InputHandle* of
7131 SQL_NULL_HANDLE. The implementation allocates memory for the environment
7132 information, and passes the value of the associated handle back in **OutputHandlePtr*. The
7133 application uses *OutputHandle* in all subsequent calls that require an environment handle
7134 argument.

7135 **Allocating a Connection Handle**

7136 A connection handle provides access to information such as the valid statement and descriptor
7137 handles on the connection and whether a transaction is currently open. To request a connection
7138 handle, an application calls *SQLAllocHandle*() with *HandleType* of SQL_HANDLE_DBC. The
7139 *InputHandle* argument is set to an environment handle, returned by another call to
7140 *SQLAllocHandle*(), for the environment on which to allocate the connection handle.

7141 The implementation allocates memory for the connection information, and passes the value of
7142 the associated handle back in **OutputHandlePtr*. The application uses **OutputHandlePtr* in all
7143 subsequent calls that require a connection handle.

7144 **Allocating a Statement Handle**

7145 A statement handle provides access to statement information, such as error messages, the cursor
7146 name, and status information for SQL statement processing. To request a statement handle
7147 before submitting SQL statements, an application connects to a data source, and then calls
7148 *SQLAllocHandle*() with *HandleType* set to SQL_HANDLE_STMT and *InputHandle* set to the
7149 connection handle for the connection on which the statement handle is to be allocated.

7150 The implementation allocates memory for the statement information, associates the statement
7151 handle with the connection specified, and passes the value of the associated handle back in
7152 **OutputHandlePtr*. The application uses **OutputHandlePtr* in all subsequent calls that require a
7153 statement handle.

7154 **Allocating a Descriptor Handle**

7155 When the statement handle is allocated, the implementation automatically allocates a set of four
7156 descriptors, and assigns the handles for these descriptors to the SQL_ATTR_APP_ROW_DESC,
7157 SQL_ATTR_APP_PARAM_DESC, SQL_ATTR_IMP_ROW_DESC, and
7158 SQL_ATTR_IMP_PARAM_DESC statement attributes. Use of explicitly-allocated application
7159 descriptors instead of the automatically-allocated ones is discussed next.

7160 The application can call *SQLAllocHandle*() with a *HandleType* of SQL_HANDLE_DESC to allocate
7161 an application descriptor explicitly. The application can use such a descriptor in place of an
7162 automatically-allocated one by calling the *SQLSetStmtAttr*() function with *Attribute* set to
7163 SQL_ATTR_APP_ROW_DESC or SQL_ATTR_APP_PARAM_DESC.

7164 The application cannot use explicitly-allocated descriptor handles as the implementation
7165 descriptors, nor specify an implementation descriptor in a *SQLSetStmtAttr*() call.

7166 Explicitly-allocated descriptors are associated with a connection handle rather than a statement
7167 handle (as automatically allocated descriptors are). Descriptors can be associated with a
7168 connection handle only when an application is actually connected to the database. Since
7169 explicitly-allocated descriptors are associated with a connection handle, an application can
7170 explicitly associate an allocated descriptor with more than one statement within a connection.
7171 An automatically-allocated application descriptor, on the other hand, cannot be associated with
7172 more than one statement handle. Explicitly-allocated descriptor handles can either be freed

explicitly by the application, by calling *SQLFreeHandle*() with a *HandleType* of SQL_HANDLE_DESC, or freed implicitly when the connection handle is freed upon disconnect.

When the application associates an explicitly-allocated application descriptor with a statement, the automatically-allocated descriptor that is superseded remains associated with the connection handle. When the application frees the explicitly-allocated descriptor, the automatically-allocated descriptor once again takes effect, as though *SQLSetStmtAttr*() had been called to set SQL_ATTR_APP_ROW_DESC or SQL_ATTR_APP_PARAM_DESC to the automatically-allocated descriptor handle. This is true for all statements that were associated with the explicitly-allocated descriptor on the connection.

When a descriptor is first used, the initial value of its SQL_DESC_TYPE field is SQL_C_DEFAULT. DATA_PTR, INDICATOR_PTR, and OCTET_LENGTH_PTR are all initially set to null pointers. For the initial values of other fields, see *SQLSetDescField*().

SEE ALSO

For information about	See
Executing an SQL statement	*SQLExecDirect*()
Executing a prepared SQL statement	*SQLExecute*()
Freeing an environment, connection, statement, or descriptor handle	*SQLFreeHandle*()
Preparing a statement for execution	*SQLPrepare*()
Setting a connection attribute	*SQLSetConnectAttr*()
Setting a descriptor field; initial values of descriptor fields	*SQLSetDescField*()
Setting an environment attribute	*SQLSetEnvAttr*()
Setting a statement attribute	*SQLSetStmtAttr*()

CHANGE HISTORY

Version 2

Revised generally. See **Alignment with Popular Implementations** on page 2.

NAME

SQLBindCol — Bind application data buffers to columns in the result set.

SYNOPSIS

```
SQLRETURN SQLBindCol(
  SQLHSTMT StatementHandle,
  SQLUSMALLINT ColumnNumber,
  SQLSMALLINT TargetType,
  SQLPOINTER TargetValuePtr,
  SQLINTEGER BufferLength,
  SQLINTEGER * StrLen_or_IndPtr);
```

ARGUMENTS

StatementHandle [Input]
Statement handle.

ColumnNumber [Input]
Number of the result set column to bind. The first column is column 0, the bookmark column. If bookmarks are not used (if the SQL_ATTR_USE_BOOKMARKS statement attribute is SQL_UB_OFF) then the first column is column 1.

TargetType [Input]
The identifier of the C data type of the **TargetValuePtr* buffer. When retrieving data from the data source with *SQLFetch*(), *SQLFetchScroll*(), or *SQLSetPos*(), the implementation converts the data to to this type; when sending data to the data source with *SQLBulkOperations*() or *SQLSetPos*(), the implementation converts the data from this type. For a list of valid C data types and type identifiers, see Section D.2 on page 560. Appendix D gives details of data type conversion.

If *TargetType* is an interval data type, the default interval leading precision and default interval seconds precision (as set in the SQL_DESC_DATETIME_INTERVAL_PRECISION and SQL_DESC_PRECISION fields of the ARD, respectively) are used for the data. If *TargetType* is a SQL_C_NUMERIC data type, the default precision and default scale (as set in the SQL_DESC_PRECISION and SQL_DESC_SCALE fields of the ARD) are used for the data. If any default precision or scale is not appropriate, the application should explicitly set the descriptor field by a call to *SQLSetDescField*() or *SQLSetDescRec*().

TargetValuePtr [Deferred Input/Output]
Pointer to the data buffer to bind to the column. *SQLFetch*() and *SQLFetchScroll*() return data in this buffer. *SQLBulkOperations*() retrieves data from this buffer when *Operation* is SQL_ADD, SQL_UPDATE_BY_BOOKMARK, or SQL_DELETE_BY_BOOKMARK. *SQLSetPos*() returns data in this buffer when *Operation* is SQL_REFRESH; it retrieves data from this buffer when *Operation* is SQL_UPDATE.

If *TargetValuePtr* is a null pointer, the implementation unbinds the column. (An application can unbind all columns by calling *SQLFreeStmt*() with the SQL_UNBIND option.)

BufferLength [Input]
Length of the **TargetValuePtr* buffer in octets.

The implementation uses *BufferLength* to avoid writing past the end of the **TargetValuePtr* buffer when returning variable-length data, such as character or binary data. This value includes the null terminator. **TargetValuePtr* must therefore contain space for the null terminator or the implementation truncates the data.

When retrieving fixed-length data from the data source, such as an integer or a date structure, the implementation ignores *BufferLength* and assumes the buffer is large enough to hold the data. The application must allocate sufficient buffer space or the implementation

7247 writes past the end of the buffer.

7248 *SQLBindCol*() returns SQLSTATE HY090 (Invalid string or buffer length) when *BufferLength*
7249 is less than 0.[29]

7250 *StrLen_or_IndPtr* [Deferred Input/Output]
7251 Pointer to the length/indicator buffer to bind to the column. *SQLFetch*() and
7252 *SQLFetchScroll*() return a value in this buffer. *SQLBulkOperations*() retrieves a value from
7253 this buffer when *Operation* is SQL_ADD, SQL_UPDATE_BY_BOOKMARK, or
7254 SQL_DELETE_BY_BOOKMARK. *SQLSetPos*() returns a value in this buffer when *Operation*
7255 is SQL_REFRESH; it retrieves a value from this buffer when *Operation* is SQL_UPDATE.

7256 *SQLFetch*(), *SQLFetchScroll*(), and *SQLSetPos*() can return the following values in the
7257 length/indicator buffer:

7258 • The length of the data available to return
7259 • SQL_NO_TOTAL
7260 • SQL_NULL_DATA

7261 The application can place the following values in the length/indicator buffer for use with
7262 *SQLBulkOperations*() or *SQLSetPos*():

7263 • The length of the data being sent
7264 • SQL_NTS
7265 • SQL_NULL_DATA
7266 • SQL_DATA_AT_EXEC
7267 • The result of the SQL_LEN_DATA_AT_EXECmacro
7268 • SQL_COLUMN_IGNORE

7269 If *StrLen_or_IndPtr* is a null pointer, no length or indicator value is used. This is an error
7270 when fetching data and the data is NULL. It is also an error when sending character or
7271 binary data.

7272 For more information, see Section 4.3.5 on page 42.

7273 **RETURN VALUE**
7274 SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.

7275 **DIAGNOSTICS**
7276 When *SQLBindCol*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated
7277 SQLSTATE value can be obtained by calling *SQLGetDiagRec*() with a *HandleType* of
7278 SQL_HANDLE_STMT and a *Handle* of *StatementHandle*. The following SQLSTATE values are
7279 commonly returned by *SQLBindCol*(). The return code associated with each SQLSTATE value is
7280 SQL_ERROR, except that for SQLSTATE values in class 01, the return code is
7281 SQL_SUCCESS_WITH_INFO.

7282 01000 — General warning
7283 Implementation-defined informational message.

7284 07006 — Restricted data type attribute violation
7285 *ColumnNumber* was 0 and *TargetType* was not SQL_C_VARBOOKMARK.

7286 07009 — Invalid descriptor index
7287 The value specified for *ColumnNumber* exceeded the maximum number of columns in the

7288 ________________

7289 29. It is no longer an error to specify a *BufferLength* of 0, but it was an error (HY090) in the X/Open **CLI** specification (1995).
Applications should not specify a value of 0.

7290 result set.

7291 HY000 — General error
7292 An error occurred for which there was no specific SQLSTATE and for which no
7293 implementation-specific SQLSTATE was defined. The error message returned by
7294 *SQLGetDiagRec*() in the **MessageText* buffer describes the error and its cause.

7295 HY001 — Memory allocation error
7296 The implementation failed to allocate memory required to support execution or completion
7297 of the function.

7298 HY003 — Invalid application buffer type
7299 *TargetType* was neither a valid data type nor SQL_C_DEFAULT.

7300 HY010 — Function sequence error
7301 An asynchronously executing function was called for *StatementHandle* and was still
7302 executing when this function was called.

7303 *SQLBulkOperations*(), *SQLExecDirect*(), *SQLExecute*(), or *SQLSetPos*() was called for
7304 *StatementHandle* and returned SQL_NEED_DATA. This function was called before data was
7305 sent for all data-at-execution parameters or columns.

7306 HY021 — Inconsistent descriptor information
7307 The descriptor consistency check failed (see **Consistency Checks** on page 486).

7308 HY090 — Invalid string or buffer length
7309 *BufferLength* was less than 0.

7310 HYC00 — Optional feature not implemented
7311 The implementation does not support the conversion specified by the combination of
7312 *TargetType* and the SQL data type of the corresponding column. This error only applies
7313 when the SQL data type of the column was mapped to an implementation-defined SQL
7314 data type.

7315 *ColumnNumber* was 0 and the data source does not support bookmarks.

7316 HYT01 — Connection timeout expired
7317 The connection timeout period expired before the data source responded to the request. The
7318 connection timeout period is set through *SQLSetConnectAttr*(),
7319 SQL_ATTR_CONNECTION_TIMEOUT.

7320 IM001 — Function not supported
7321 The function is not supported on the current connection to the data source.

7322 **COMMENTS**

7323 **Overview**

7324 *SQLBindCol*() associates, or binds, columns in the result set to data buffers and length/indicator
7325 buffers in the application. When the application calls *SQLFetch*(), *SQLFetchScroll*(), or
7326 *SQLSetPos*() to fetch data, the implementation returns the data for the bound columns in the
7327 specified buffers. When the application calls *SQLBulkOperations*() to update or insert a row, or
7328 *SQLSetPos*() to update a row, the implementation retrieves the data for the bound columns from
7329 the specified buffers.

7330 Columns do not have to be bound to retrieve data from them. An application can bind some
7331 columns of a row and call *SQLGetData*() for others. Certain restrictions exist; see **Restrictions on**
7332 **Use of SQLGetData()** on page 347.

Binding, Unbinding, and Rebinding Columns

A column can be bound, unbound, or rebound at any time, even after data has been fetched from the result set. The new binding takes effect the next time a function that uses bindings is called. In particular, *SQLBindCol*() does not access the newly bound buffers. For example, suppose an application binds the columns in a result set and calls *SQLFetch*(). The data is returned in the bound buffers. Now suppose the application binds the columns to a different set of buffers. The data for the just-fetched row does not move to the newly-bound buffers. But subsequent calls to *SQLFetch*() place the data for subsequent rows in the newly-bound buffers.

Binding Columns

To bind a column, an application calls *SQLBindCol*() and passes the column number, the type, address, and length of a data buffer, and the address of a length/indicator buffer. For information on how these addresses are used, see **Buffer Addresses** on page 217.

The use of these buffers is deferred. That is, the application binds them in *SQLBindCol*() but the implementation uses their values only when retrieving data from the data source. The application must ensure that the pointers specified in *SQLBindCol*() remain valid as long as the binding remains in effect. If the application lets these pointers become invalid — for example, if it frees a buffer — and then calls a function that depends on their values, the consequences are undefined. For more information, see Section 4.3.1 on page 39.

The binding remains in effect until it is replaced by a new binding, the column is unbound, the statement is freed, or the SQL_DESC_COUNT field is set to 0 in the ARD.

Unbinding Columns

To unbind a single column, an application calls *SQLBindCol*() with *ColumnNumber* set to the number of that column and *TargetValuePtr* set to a null pointer. If *ColumnNumber* refers to an unbound column, *SQLBindCol*() still returns SQL_SUCCESS.

To unbind all columns, an application calls *SQLFreeStmt*() with *fOption* set to SQL_UNBIND.

The application can also unbind all columns (except any bookmark) by setting the SQL_DESC_COUNT field in the header record of the ARD to 0.

Rebinding Columns

An application can perform either of two operations to change a binding:

- Call *SQLBindCol*() to specify a new binding for a column that is already bound. The implementation overwrites the old binding with the new one.
- Specify an offset to be added to the buffer address that was specified by the binding call to *SQLBindCol*(). For more information, see **Bind Offsets**.

Binding Arrays

If the row-set size (the value of the SQL_ATTR_ROW_ARRAY_SIZE statement attribute) is greater than 1, the application binds arrays of buffers rather than single buffers. The application can either bind separate data and length/indicator arrays to each column of data (known as column-wise binding) or to each row of data (row-wise binding).

The application can bind arrays in two ways:

- 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.

Each array of buffers must have at least as many elements as the size of the row-set.

Note: An application must verify that alignment is valid.

Column-Wise Binding

In column-wise binding, the application binds separate data and length/indicator arrays to each column.

To use column-wise binding, the application first sets the SQL_ATTR_ROW_BIND_TYPE statement attribute to SQL_BIND_BY_COLUMN (this is the default). For each column to be bound, it:

- Allocates a data buffer array.
- Allocates an array of length/indicator buffers.
- Calls *SQLBindCol*():
  - — *TargetType* is the type of a single element in the data buffer array.
  - — *TargetValuePtr* is the address of the data buffer array.
  - — *BufferLength* is the size of a single element in the data buffer array. *BufferLength* is ignored when the data is fixed-length data.
  - — *StrLen_or_IndPtr* is the address of the length/indicator array.

For more information on how this information is used, see **Buffer Addresses** on page 217.

Row-Wise Binding

In row-wise binding, the application defines a structure containing data and length/indicator buffers for each column to be bound.

To use row-wise binding, the application:

- Defines a structure to hold a single row of data (including both data and length/indicator buffers) and allocates an array of these structures.
- Sets the SQL_ATTR_ROW_BIND_TYPE statement attribute to the size of the structure containing a single row of data, or to the size of an instance of a buffer into which the results columns will be bound. The length must include space for all of the bound columns, and any padding of the structure or buffer to ensure that when the address of a bound column is incremented with the specified length, the result will point to the beginning of the same column in the next row. When using the **sizeof** operator in ANSI C, this behavior is guaranteed.
- Calls *SQLBindCol*() for each column to be bound:
  - — *TargetType* is the type of the data buffer member to be bound to the column.
  - — *TargetValuePtr* is the address of the data buffer member in the first array element.
  - — *BufferLength* is the size of the data buffer member.
  - — *StrLen_or_IndPtr* is the address of the length/indicator member to be bound.

For more information on how this information is used, see **Buffer Addresses**.

7413 **Bind Offsets**

7414 A bind offset is a value that is added to the addresses of the data and length/indicator buffers
7415 (as specified in *TargetValuePtr* and *StrLen_or_IndPtr*) before they are dereferenced. When offsets
7416 are used, the bindings are a template of how the application's buffers are laid out and the
7417 application can move this template to different areas of memory by changing the offset. Because
7418 the same offset is added to each address in each binding, the relative offsets between buffers for
7419 different columns must be the same within each set of buffers. This is always true when row-
7420 wise binding is used; the application must carefully lay out its buffers for this to be true when
7421 column-wise binding is used.

7422 Using a binding offset has much the same effect as rebinding a column by calling *SQLBindCol*().
7423 The difference is that a new call to *SQLBindCol*() specifies new addresses for the data buffer and
7424 length/indicator buffer, while use of a bind offset does not change the addresses, but merely
7425 adds an offset to them. The application can specify a new offset whenever it wants and this
7426 offset is always added to the originally-bound addresses. In particular, if the offset is set to 0 or if
7427 the statement attribute is set to a null pointer, the implementation uses the originally-bound
7428 addresses.

7429 To specify a bind offset, the application sets the SQL_ATTR_ROW_BIND_OFFSET_PTR
7430 statement attribute to the address of an SQLINTEGER buffer. Before the application calls a
7431 function that uses bindings, it places an offset in octets in this buffer. To determine the address
7432 of the buffer to use, the implementation adds the offset to the address in the binding. The sum of
7433 the address and the offset must be a valid address, but the address to which the offset is added
7434 need not be a valid address. For more information on how bind offsets are used, see **Buffer**
7435 **Addresses** on page 217.

7436 **Buffer Addresses**

7437 The *buffer address* is the actual address of the data or length/indicator buffer. It is calculated from
7438 the following formula, which uses the addresses specified in the *TargetValuePtr* and
7439 *StrLen_or_IndPtr* arguments, the bind offset, and the row number:

7440 *Bound Address* + *Bind Offset* + ((*Row Number* - 1) x *Element Size*)

7441 where

7442 Variable	Description
7443 *Bound Address*	For data buffers, the address specified with *TargetValuePtr* in *SQLBindCol*(). 7444 For length/indicator buffers, the address specified with *StrLen_or_IndPtr* in 7445 *SQLBindCol*(). 7446 For more information, see **Additional Comments** on page 314. 7447 If the bound address is 0, no data value is returned, even if the address as 7448 calculated by the formula above is non-zero.
7449 *Bind Offset*	If row-wise binding is used, the value stored at the address specified with the 7450 SQL_ATTR_ROW_BIND_OFFSET_PTR statement attribute. If this attribute is a 7451 null pointer, *Bind Offset* is 0. 7452 If column-wise binding is used, *Bind Offset* is 0.
7453 *Row Number*	The 1-based number of the row in the row-set. For single-row fetches, which are 7454 the default, this is 1.

7455 *Element Size* The size of an element in the bound array.

7456 If column-wise binding is used, this is **sizeof(SQLINTEGER)** for
7457 length/indicator buffers. For data buffers, it is the value of the *BufferLength*
7458 argument in *SQLBindCol*() if the data type is variable length and the size of the
7459 data type if the data type is fixed length.

7460 If row-wise binding is used, this is the value of the
7461 SQL_ATTR_ROW_BIND_TYPE statement attribute for both data and
7462 length/indicator buffers.

7463 **Descriptors and SQLBindCol()**

7464 The following sections describe how *SQLBindCol*() interacts with descriptors.

7465 **Caution:** Calling *SQLBindCol*() for one statement affects other statements if the ARD associated
7466 with the statement is explicitly allocated and is also associated with other statements. Any
7467 modifications made to a descriptor with *SQLBindCol*() apply to all statements with which the
7468 descriptor is associated. To prevent this effect, the application must dissociate this descriptor
7469 from the other statements before calling *SQLBindCol*().

7470 **Argument Mappings**

7471 Conceptually, *SQLBindCol*() performs the following steps in sequence:

7472 • Calls *SQLGetStmtAttr*() to obtain the application row descriptor handle.

7473 • Calls *SQLGetDescField*() to get this descriptor's COUNT field, and if *ColumnNumber* exceeds
7474 the value of COUNT, calls *SQLSetDescField*() to increase the value of COUNT to
7475 *ColumnNumber*.

7476 • Calls *SQLSetDescField*() multiple times to assign values to the following fields of the
7477 application row descriptor:

7478 — sets TYPE to the value of *TargetType*

7479 — sets OCTET_LENGTH to the value of *BufferLength*

7480 — sets DATA_PTR to the value of *TargetValue*

7481 — sets INDICATOR_PTR to the value of *StrLen_or_Ind* (see below)

7482 — sets OCTET_LENGTH_PTR to the value of *StrLen_or_Ind* (see below).
7483 The variable that *StrLen_or_Ind* references is used for both indicator and length information.
7484 If a fetch encounters a null value for the column, it stores SQL_NULL_DATA in this variable;
7485 otherwise it stores the data length in this variable. Passing a null pointer as *StrLen_or_Ind*
7486 keeps the fetch operation from returning the data length, but makes the fetch fail if it
7487 encounters a null value and has no way to return SQL_NULL_DATA.

7488 If the call to *SQLBindCol*() fails, the content of the descriptor fields it would have set are
7489 undefined.

7490 **Implicit Resetting of COUNT Field**

7491 *SQLBindCol*() sets SQL_DESC_COUNT to *ColumnNumber* only when this would serve to
7492 increase the value of SQL_DESC_COUNT. If *TargetValuePtr* is a null pointer and *ColumnNumber*
7493 is equal to SQL_DESC_COUNT (that is, when unbinding the highest bound column), then
7494 SQL_DESC_COUNT is set to the number of the highest remaining bound column.

7495 **Cautions Regarding SQL_DEFAULT**

7496 To retrieve column data successfully, the application must determine correctly the length and
7497 starting point of the data in the application buffer. When the application specifies an explicit
7498 *TargetType*, application misconceptions are readily detected. However, when the application
7499 specifies a *TargetType* of SQL_DEFAULT, *SQLBindCol*() can be applied to a column of a different
7500 data type from the one intended by the application, either from changes to the metadata or by
7501 applying the code to a different column. In this case, the application may fail to determine the
7502 start or length of the fetched column data. This can lead to unreported data errors or memory
7503 violations.

7504 **Other Descriptor Fields**

7505 The SQL_DESC_BIND_OFFSET_PTR descriptor field is also related to binding columns. This
7506 header field in the ARD can be set through *SQLSetDescField*() or through the
7507 SQL_ATTR_ROW_BIND_OFFSET_PTR statement attribute.

7508 **SEE ALSO**

	For information about	See
7509		
7510	Returning information about a column in a result set	*SQLDescribeCol*()
7511	Fetching a block of data or scrolling through a result set	*SQLFetchScroll*()
7512	Fetching multiple rows of data	*SQLFetch*()
7513	Freeing a statement handle	*SQLFreeStmt*()
7514	Fetching part or all of a column of data	*SQLGetData*()
7515	Returning the number of result set columns	*SQLNumResultCols*()

7516 **CHANGE HISTORY**

7517 **Version 2**

7518 Revised generally. See **Alignment with Popular Implementations** on page 2.

7519 **NAME**

7520 SQLBindParam — Bind a dynamic parameter

7521 **SYNOPSIS**

```
7522 DE  SQLRETURN BindParam
7523       (SQLHSTMT StatementHandle,
7524        SQLUSMALLINT ParameterNumber,
7525        SQLSMALLINT ValueType,
7526        SQLSMALLINT ParameterType,
7527        SQLUINTEGER ColumnSize,
7528        SQLSMALLINT DecimalDigits,
7529        SQLPOINTER ParameterValue,
7530        SQLINTEGER * StrLen_or_Ind)
7531      RETURNS (SMALLINT)
```

7532 **DESCRIPTION**

7533 The *BindParam*() function is identical in effect to an equivalent call to *SQLBindParameter*().

7534 **NAME**

7535 SQLBindParameter — Bind a buffer to a parameter marker in an SQL statement.

7536 **SYNOPSIS**

```
7537 SQLRETURN SQLBindParameter(
7538   SQLHSTMT StatementHandle,
7539   SQLUSMALLINT ParameterNumber,
7540   SQLSMALLINT InputOutputType,
7541   SQLSMALLINT ValueType,
7542   SQLSMALLINT ParameterType,
7543   SQLUINTEGER ColumnSize,
7544   SQLSMALLINT DecimalDigits,
7545   SQLPOINTER ParameterValuePtr,
7546   SQLINTEGER BufferLength,
7547   SQLINTEGER * StrLen_or_IndPtr);
```

7548 **ARGUMENTS**

7549 *StatementHandle* [Input]

7550 Statement handle.

7551 *ParameterNumber* [Input]

7552 Parameter number, ordered sequentially left to right, starting at 1.

7553 *InputOutputType* [Input]

7554 The type of the parameter; see **InputOutputType Argument** on page 223.

7555 *ValueType*[Input]

7556 The C data type of the parameter; see **ValueTypeArgument** on page 224.

7557 *ParameterType*[Input]

7558 The SQL data type of the parameter; see **ParameterType Argument** on page 224.

7559 *ColumnSize* [Input]

7560 The size of the column or expression of the corresponding parameter marker; see
7561 **ColumnSize Argument** on page 225.

7562 *DecimalDigits* [Input]

7563 The decimal digits of the column or expression of the corresponding parameter marker; see
7564 Section D.3.2 on page 564.

7565 *ParameterValuePtr*[Deferred Input]

7566 A pointer to a buffer for the parameter's data; see **ParameterValuePtr Argument** on page
7567 225.

7568 *BufferLength* [Input]

7569 Length of the *ParameterValuePtr*buffer in octets; see **BufferLength Argument** on page 226.

7570 *StrLen_or_IndPtr* [Deferred Input]

7571 A pointer to a buffer for the parameter's length; see **StrLen_or_IndPtr Argument** on page
7572 226.

7573 **RETURN VALUE**

7574 SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.

7575 **DIAGNOSTICS**

7576 When *SQLBindParameter*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated
7577 SQLSTATE value may be obtained by calling *SQLGetDiagRec*() with *HandleType* of
7578 SQL_HANDLE_STMT and a *Handle* of *StatementHandle*. The following SQLSTATE values are
7579 commonly returned by *SQLBindParameter*(). The return code associated with each SQLSTATE

value is SQL_ERROR, except that for SQLSTATE values in class 01, the return code is SQL_SUCCESS_WITH_INFO.

01000 — General warning
Implementation-defined informational message.

07006 — Restricted data type attribute violation
The data type identified by the ValueType argument cannot be converted to the data type identified by the ParameterType argument. Note that this error may be returned by SQLExecDirect, SQLExecute, or SQLPutData at execution time, instead of by SQLBindParameter.

07009 — Invalid descriptor index
The value specified for *ParameterNumber* was less than to 0.

HY000 — General error
An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by *SQLGetDiagRec*() in the **MessageText* buffer describes the error and its cause.

HY001 — Memory allocation error
The implementation failed to allocate memory required to support execution or completion of the function.

HY003 — Invalid application buffer type
ValueType was not a valid data type or SQL_C_DEFAULT.

HY004 — Invalid SQL data type
ParameterType was neither a valid XDBC SQL data type identifier nor an implementation-defined SQL data type identifier that the data source supports.

HY009 — Invalid use of null pointer
ParameterValuePtr and *StrLen_or_IndPtr* were null pointers and *InputOutputType* was not SQL_PARAM_OUTPUT.

HY010 — Function sequence error
An asynchronously executing function was called for *StatementHandle* and was still executing when this function was called.

SQLBulkOperations(), *SQLExecDirect*(), *SQLExecute*(), or *SQLSetPos*() was called for *StatementHandle* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.

HY021 — Inconsistent descriptor information
The descriptor consistency check failed (see **Consistency Checks** on page 486).

The value specified for *DecimalDigits* was outside the range of values supported by the data source for a column of the SQL data type specified by *ParameterType*.

HY090 — Invalid string or buffer length
BufferLength was less than 0. (See the description of the SQL_DESC_DATA_PTR field in *SQLSetDescField*().)

HY104 — Invalid precision value
The value specified for *ColumnSize* or *DecimalDigits* was outside the range of values supported by the data source for a column of the SQL data type specified by *ParameterType*.

HY105 — Invalid parameter type
InputOutputType was invalid (see **InputOutputType Argument** on page 223.

7624 HYC00 — Optional feature not implemented
7625 The implementation does not support the conversion specified by the combination of
7626 *ValueType*and *ParameterType*.

7627 HYT01 — Connection timeout expired
7628 The connection timeout period expired before the data source responded to the request. The
7629 connection timeout period is set through *SQLSetConnectAttr*(),
7630 SQL_ATTR_CONNECTION_TIMEOUT.

7631 IM001 — Function not supported
7632 The function is not supported on the current connection to the data source.

7633 **COMMENTS**

7634 An application calls *SQLBindParameter*() to bind each parameter marker in an SQL statement.
7635 Bindings remain in effect until the application calls *SQLBindParameter*() again, calls
7636 *SQLFreeStmt*() with the SQL_RESET_PARAMS option, or calls *SQLSetDescField*() to set the
7637 SQL_DESC_COUNT header field of the APD to 0.

7638 **ParameterNumber Argument**

7639 If *ParameterNumber* in the call to *SQLBindParameter*() is greater than the value of
7640 SQL_DESC_COUNT, the value of the SQL_DESC_COUNT field implicitly increases to equal
7641 *ParameterNumber*.

7642 **InputOutputType Argument**

7643 *InputOutputType* specifies the type of the parameter. This argument sets the
7644 SQL_DESC_PARAMETER_TYPE field of the IPD. All parameters in SQL statements that do not
7645 call procedures, such as INSERT statements, are input parameters. Parameters in procedure
7646 calls can be input, input/output, or output parameters. (An application calls
7647 *SQLProcedureColumns*() to determine the type of a parameter in a procedure call; parameters in
7648 procedure calls whose type cannot be determined are assumed to be input parameters.)

7649 *InputOutputType* is one of the following values:

7650 • SQL_PARAM_INPUT. The parameter marks a parameter in an SQL statement that does not
7651 call a procedure, such as an INSERT statement, or it marks an input parameter in a
7652 procedure; these are collectively known as input parameters. For example, the parameters in
7653 **INSERT INTO Employee VALUES (?, ?, ?)** are input parameters.

7654 When the statement is executed, the implementation sends data for the parameter to the data
7655 source; the **ParameterValuePtr* buffer must contain a valid input value or the
7656 **StrLen_or_IndPtr* buffer must contain SQL_NULL_DATA, SQL_DATA_AT_EXEC, or the
7657 result of the SQL_LEN_DATA_AT_EXECmacro.

7658 If an application cannot determine the type of a parameter in a procedure call, it sets
7659 *InputOutputType* to SQL_PARAM_INPUT; if the data source returns a value for the
7660 parameter, the implementation discards it.

7661 • SQL_PARAM_INPUT_OUTPUT. The parameter marks an input/output parameter in a
7662 procedure. For example, the parameter in **{call GetEmpDept(?)}** is an input/output
7663 parameter that accepts an employee's name and returns the name of the employee's
7664 department.

7665 When the statement is executed, the implementation sends data for the parameter to the data
7666 source; the **ParameterValuePtr* buffer must contain a valid input value or the
7667 **StrLen_or_IndPtr* buffer must contain SQL_NULL_DATA, SQL_DATA_AT_EXEC, or the
7668 result of the SQL_LEN_DATA_AT_EXEC macro. After the statement is executed, the
7669 implementation returns data for the parameter to the application; if the data source does not

7670 return a value for an input/output parameter, the implementation sets the **StrLen_or_IndPtr*
7671 buffer to SQL_NULL_DATA.

- 7672 SQL_PARAM_OUTPUT. The parameter marks the return value of a procedure or an output
7673 parameter in a procedure; these are collectively known as output parameters. For example,
7674 the parameter in **{?=call GetNextEmpID}** is an output parameter that returns the next
7675 employee ID.

7676 After the statement is executed, the implementation returns data for the parameter to the
7677 application, unless *ParameterValuePtr* and *StrLen_or_IndPtr* are both null pointers, in which
7678 case the implementation discards the output value. If the data source does not return a value
7679 for an output parameter, the implementation sets the **StrLen_or_IndPtr* buffer to
7680 SQL_NULL_DATA.

7681 **ValueTypeArgument**

7682 *ValueType*specifies the C data type of the parameter. It must be one of the values in Section D.2
7683 on page 560. The implementation stores this value in the SQL_DESC_TYPE,
7684 SQL_DESC_CONCISE_TYPE, and SQL_DESC_DATETIME_INTERVAL_CODE fields of the
7685 APD.

7686 If *ValueType* is an interval data type, the implementation sets the SQL_DESC_TYPE field to
7687 SQL_INTERVAL, sets the SQL_DESC_CONCISE_TYPE field to the concise interval data type,
7688 and sets the SQL_DESC_DATETIME_INTERVAL_CODE field to a subcode for the specific
7689 date/time or interval data type (see Section D.4 on page 569). The default interval leading
7690 precision and default interval seconds precision (as set in the
7691 SQL_DESC_DATETIME_INTERVAL_PRECISION and SQL_DESC_PRECISION fields of the
7692 ARD, respectively) are used for the data.

7693 If *ValueType* is a date/time data type, the SQL_DESC_TYPE field is set to SQL_DATETIME, the
7694 SQL_DESC_CONCISE_TYPE field is set to the concise date/time data type, and the
7695 SQL_DESC_DATETIME_INTERVAL_CODE field is set to a subcode for the specific date/time
7696 data type (see Appendix D).

7697 If *ValueType* is an SQL_C_NUMERIC data type, the default precision and default scale (as set in
7698 the SQL_DESC_PRECISION and SQL_DESC_SCALE fields of the ARD) are used for the data. If
7699 any default precision or scale is not appropriate, the application should explicitly set the
7700 descriptor field by a call to *SQLSetDescField*() or *SQLSetDescRec*().

7701 If *ValueType* is SQL_C_DEFAULT, the parameter value is transferred from the default C data type
7702 for the SQL data type specified with *ParameterType*.

7703 Appendix D specifies the valid combinations of data types for type conversion, and defines the
7704 effects of SQL_C_DEFAULT.

7705 **ParameterType Argument**

7706 *ParameterType* must be one of the SQL data types listed in Section D.1 on page 556 or an
7707 implementation-defined value. This argument sets the SQL_DESC_TYPE,
7708 SQL_DESC_CONCISE_TYPE, and SQL_DESC_DATETIME_INTERVAL_CODE fields of the
7709 IPD.

7710 If *ParameterType* is one of the date/time identifiers, the SQL_DESC_TYPE field of the IPD is set to
7711 SQL_DATETIME, the SQL_DESC_CONCISE_TYPE field of the IPD is set to the concise
7712 date/time data type, and the SQL_DESC_DATETIME_INTERVAL_CODE field is set to the
7713 appropriate date/time subcode value.

7714 If *ParameterType* is one of the interval identifiers, the SQL_DESC_TYPE field of the IPD is set to
7715 SQL_INTERVAL, the SQL_DESC_CONCISE_TYPE field of the IPD is set to the concise interval

7716 data type. The SQL_DESC_DATETIME_INTERVAL_CODE field of the IPD is set to the
7717 appropriate interval subcode, the SQL_DESC_DATETIME_INTERVAL_PRECISION field of the
7718 IPD is set to the interval leading precision, and the SQL_DESC_PRECISION field is set to the
7719 interval seconds precision, if applicable. If the default value of
7720 SQL_DESC_DATETIME_INTERVAL_PRECISION or SQL_DESC_PRECISION is not
7721 appropriate, the application should explicitly set it by calling *SQLSetDescField*(). See the
7722 description for these fields in *SQLSetDescField*().

7723 Appendix D describes how data is converted.

7724 **ColumnSize Argument**

7725 *ColumnSize* specifies the size of the column or expression corresponding to the parameter
7726 marker, or the length of that data, or both. This argument determines the
7727 SQL_DESC_PRECISION or the SQL_DESC_LENGTH field of the IPD, or both, depending on the
7728 SQL data type in *ParameterType*. The following rules apply to this mapping:

7729 • If *ParameterType* is SQL_CHAR, SQL_VARCHAR, SQL_LONGVARCHAR, SQL_BINARY,
7730 SQL_VARBINARY, SQL_LONGVARBINARY, or one of the concise date/time or interval
7731 data types (that is, SQL_TYPE_DATE or SQL_INTERVAL_YEAR_TO_MONTH), the
7732 SQL_DESC_LENGTH field of the IPD is set to the value of *ColumnSize*.

7733 • If *ParameterType* is SQL_DECIMAL, SQL_NUMERIC, SQL_FLOAT, SQL_REAL, or
7734 SQL_DOUBLE, the SQL_DESC_PRECISION field of the IPD is set to the value of *ColumnSize*.

7735 • For other data types, *ColumnSize* is ignored.

7736 For more information on column size, see Section D.3.1 on page 562. Also see **Passing Parameter**
7737 **Values** on page 227 and SQL_DATA_AT_EXECin **StrLen_or_IndPtr Argument** on page 226.

7738 **DecimalDigits Argument**

7739 *DecimalDigits* sets the SQL_DESC_SCALE field of the IPD for all numeric data types.

7740 *DecimalDigits* sets the SQL_DESC_PRECISION field of the IPD for all data types that have a
7741 seconds field (the cases in which *ParameterType* is SQL_TYPE_TIME, SQL_TYPE_TIMESTAMP,
7742 SQL_TYPE_SECOND, SQL_TYPE_DAY_TO_SECOND, SQL_TYPE_HOUR_TO_SECOND, or
7743 SQL_TYPE_MINUTE_TO_SECOND).

7744 For other data types, *DecimalDigits* is ignored.

7745 **ParameterValuePtrArgument**

7746 *ParameterValuePtr* points to a buffer that, when *SQLExecute*() or *SQLExecDirect*() is called,
7747 contains the actual data for the parameter. The data must be in the form specified by *ValueType*.
7748 *ParameterValuePtr* sets the SQL_DESC_DATA_PTR field of the APD. An application can set
7749 *ParameterValuePtr* to a null pointer, as long as **StrLen_or_IndPtr* is SQL_NULL_DATA or
7750 SQL_DATA_AT_EXEC.

7751 If **StrLen_or_IndPtr* is the result of the SQL_LEN_DATA_AT_EXEC(length) macro or
7752 SQL_DATA_AT_EXEC, then *ParameterValuePtr* is an application-defined 32-bit value that is
7753 associated with the parameter. It is returned to the application through *SQLParamData*(). For
7754 example, *ParameterValuePtr* might be a token such as a parameter number, a pointer to data, or a
7755 pointer to a structure that the application used to bind input parameters. However, if the
7756 parameter is an input/output parameter, *ParameterValuePtr* must point to a buffer where the
7757 output value will be stored. If the value in the SQL_ATTR_PARAMSET_SIZE statement attribute
7758 is greater than 1, the application can use the value pointed to by the
7759 SQL_ATTR_PARAMS_PROCESSED_PTR statement attribute in conjunction with
7760 *ParameterValuePtr*. For example, *ParameterValuePtr* might point to an array of values and the

7761 application might use the value pointed to by SQL_ATTR_PARAMS_PROCESSED_PTR to
7762 retrieve the correct value from the array. For more information, see **Passing Parameter Values** on
7763 page 227.

7764 If *InputOutputType* is SQL_PARAM_INPUT_OUTPUT or SQL_PARAM_OUTPUT,
7765 *ParameterValuePtr* points to a buffer in which the implementation returns the output value. If the
7766 procedure returns one or more result sets, the **ParameterValuePtr* buffer is not guaranteed to be
7767 set until all results have been fetched.

7768 If the value in the SQL_ATTR_PARAMSET_SIZE statement attribute is greater than 1,
7769 *ParameterValuePtr* points to an array. A single SQL statement processes the entire array of input
7770 values for an input or input/output parameter and returns an array of output values for an
7771 input/output or output parameter.

7772 **BufferLength Argument**

7773 For character and binary C data, *BufferLength* specifies the length of the **ParameterValuePtr* buffer
7774 (if it is a single element) or the length of an element in the **ParameterValuePtr* array (if the value
7775 in the SQL_ATTR_PARAMSET_SIZE statement attribute is greater than 1). This argument sets
7776 the SQL_DESC_OCTET_LENGTH record field of the APD. If the application specifies multiple
7777 values, *BufferLength* is used to determine the location of values in the **ParameterValuePtr* array,
7778 both on input and on output. For input/output and output parameters, it is used to determine
7779 whether to truncate character and binary C data on output:

7780 • For character C data, if the number of octets available to return is greater than or equal to
7781 *BufferLength*, the data in **ParameterValuePtr* is truncated to *BufferLength* less the length of a
7782 null terminator and is null-terminated.

7783 • For binary C data, if the number of octets available to return is greater than *BufferLength*, the
7784 data in **ParameterValuePtr* is truncated to *BufferLength* octets.

7785 For all other types of C data, *BufferLength* is ignored. The length of the **ParameterValuePtr* buffer
7786 (if it is a single element) or the length of an element in the **ParameterValuePtr* array (if there are
7787 multiple values for each parameter) is assumed to be the length of the C data type.

7788 **StrLen_or_IndPtr Argument**

7789 *StrLen_or_IndPtr* points to a buffer that, when *SQLExecute*() or *SQLExecDirect*() is called,
7790 contains one of the following. This argument sets the SQL_DESC_OCTET_LENGTH_PTR and
7791 SQL_DESC_INDICATOR_PTR record fields of the application parameter pointers.

7792 • The length of the parameter value stored in **ParameterValuePtr*. This is ignored except for
7793 character or binary C data.

7794 • SQL_NTS. The parameter value is a null-terminated string.

7795 • SQL_NULL_DATA.The parameter value is NULL.

7796 • SQL_DEFAULT_PARAM.Directs a procedure to use the default value of a parameter, rather
7797 than a value retrieved from the application. This value is valid only in a procedure called
7798 using the XDBC escape clause (see Section 8.3 on page 84), and then only if *InputOutputType*
7799 is SQL_PARAM_INPUT or SQL_PARAM_INPUT_OUTPUT. The implementation ignores
7800 *ValueType*, *ParameterType*, *ColumnSize*, *DecimalDigits*, *BufferLength*, and *ParameterValuePtr* for
7801 input parameters, and uses them only to define the output parameter value for input/output
7802 parameters.

7803 • The result of the SQL_LEN_DATA_AT_EXEC(*length*) macro. The data for the parameter will
7804 be sent with *SQLPutData*(). If *ParameterType* is SQL_LONGVARBINARY,
7805 SQL_LONGVARCHAR, or a long, data-source-specific data type, and the

SQL_NEED_LONG_DATA_LEN option in *SQLGetInfo*() returns 'Y', then *length* is the number of octets of data to be sent for the parameter; otherwise, *length* must be a nonnegative value and is ignored. For more information, see **Passing Parameter Values** on page 227.

For example, to specify that 10,000 octets of data will be sent with *SQLPutData*() for an SQL_LONGVARCHAR parameter, an application sets **StrLen_or_IndPtr* to SQL_LEN_DATA_AT_EXEC(10000).

- SQL_DATA_AT_EXEC.The data for the parameter will be sent with *SQLPutData*().

If *StrLen_or_IndPtr* is a null pointer, the implementation assumes that all input parameter values are non-NULL and that character and binary data are null-terminated. If *InputOutputType* is SQL_PARAM_OUTPUT and *ParameterValuePtr* and *StrLen_or_IndPtr* are both null pointers, the implementation discards the output value.

Applications should provide valid length data, not a null pointer, through *StrLen_or_IndPtr* **when the data type of the parameter is SQL_C_BINARY, to prevent the implementation from truncating SQL_C_BINARY data.**

If *InputOutputType* is SQL_PARAM_INPUT_OUTPUT or SQL_PARAM_OUTPUT, *StrLen_or_IndPtr* points to a buffer in which the implementation returns SQL_NULL_DATA, the number of octets available to return in **ParameterValuePtr* (excluding the null terminator for character data), or SQL_NO_TOTAL if the number of octets available to return cannot be determined. If the procedure returns one or more result sets, the **StrLen_or_IndPtr* buffer is not guaranteed to be set until all results have been fetched.

If the value in the SQL_ATTR_PARAMSET_SIZE statement attribute is greater than 1, *StrLen_or_IndPtr* points to an array of SQLINTEGER values. These can be any of the values listed earlier in this section and are processed with a single SQL statement.

Passing Parameter Values

An application can pass the value for a parameter either in the **ParameterValuePtr* buffer or with one or more calls to *SQLPutData*(). Parameters whose data is passed with *SQLPutData*() are known as data-at-execution parameters. These are commonly used to send data for SQL_LONGVARBINARY and SQL_LONGVARCHAR parameters and can be mixed with other parameters.

To pass parameter values, an application:

1. Calls *SQLBindParameter*() for each parameter to bind buffers for the parameter's value (*ParameterValuePtr*) and length/indicator (*StrLen_or_IndPtr*). For data-at-execution parameters, *ParameterValuePtr* is an application-defined 32-bit value such as a parameter number or a pointer to data. The value is returned later and can be used to identify the parameter.

2. Places values for input and input/output parameters in the **ParameterValuePtr* and **StrLen_or_IndPtr* buffers:

   — For normal parameters, the application places the parameter value in the **ParameterValuePtr* buffer and the length of that value in the **StrLen_or_IndPtr* buffer.

   — For data-at-execution parameters, the application places the result of the SQL_LEN_DATA_AT_EXEC(length)macro in the **StrLen_or_IndPtr* buffer.

3. Calls *SQLExecute*() or *SQLExecDirect*() to execute the SQL statement. If there are no data-at-execution parameters, the process is complete. If there are any data-at-execution parameters, the function returns SQL_NEED_DATA.

4. Calls *SQLParamData*() to retrieve the application-defined value specified in *ParameterValuePtr* for the first data-at-execution parameter to be processed.

   Although data-at-execution parameters are similar to data-at-execution columns, the value returned by *SQLParamData*() is different for each.

   — Data-at-execution parameters are parameters in an SQL statement for which data will be sent with *SQLPutData*() when the statement is executed with *SQLExecDirect*() or *SQLExecute*(). They are bound with *SQLBindParameter*(). The value returned by *SQLParamData*() is a 32-bit value passed to *SQLBindParameter*() in *ParameterValuePtr*.

   — Data-at-execution columns are columns in a row-set for which data is sent with *SQLPutData*() when a row is updated or added with *SQLBulkOperations*() or updated with *SQLSetPos*(). They are bound with *SQLBindCol*(). The value returned by *SQLParamData*() is the address of the row in the **ParameterValuePtr* buffer that is being processed.

5. Calls *SQLPutData*() one or more times to send data for the parameter. More than one call is needed if the data value is larger than the **ParameterValuePtr* buffer specified in *SQLPutData*(); multiple calls to *SQLPutData*() for the same parameter are allowed only when sending character C data to a column with a character, binary, or data source-specific data type or when sending binary C data to a column with a character, binary, or data-source-specific data type.

6. Calls *SQLParamData*() again to signal that all data has been sent for the parameter.

7. If there are more data-at-execution parameters, *SQLParamData*() returns SQL_NEED_DATA and the application-defined value for the next data-at-execution parameter to be processed. The application repeats steps 5 and 6.

8. If there are no more data-at-execution parameters, the process is complete. If the statement was successfully executed, *SQLParamData*() returns SQL_SUCCESS or SQL_SUCCESS_WITH_INFO; if the execution failed, it returns SQL_ERROR. At this point, *SQLParamData*() can return any SQLSTATE that can be returned by the function used to execute the statement (*SQLExecDirect*() or *SQLExecute*()).

   Output values for any input/output or output parameters are available in the **ParameterValuePtr* and **StrLen_or_IndPtr* buffers after the application retrieves all result sets generated by the statement.

Calling *SQLExecute*() or *SQLExecDirect*() puts the statement in a SQL_NEED_DATA state. At this point, the application can only call *SQLCancel*(), *SQLGetDiagField*(), *SQLGetDiagRec*(), *SQLGetFunctions*(), *SQLParamData*(), or *SQLPutData*() with the statement or connection handle associated with the statement. If it calls any other function with the statement or the connection handle associated with the statement, the function returns SQLSTATE HY010 (Function sequence error). The statement leaves the SQL_NEED_DATA state when *SQLParamData*() or *SQLPutData*() returns an error, *SQLParamData*() returns SQL_SUCCESS or SQL_SUCCESS_WITH_INFO, or the statement is cancelled.

If the application calls *SQLCancel*() while the implementation still needs data for data-at-execution parameters, the implementation cancels statement execution; the application can then call *SQLExecute*() or *SQLExecDirect*() again.

7893 **Using Arrays of Parameters**

7894 When an application prepares a statement with parameter markers and passes an array of
7895 parameters, it is undefined whether the implementation uses any array-processing capabilities
7896 of the data source or generates a sequence of SQL statements, one for each set of parameters in
7897 the parameter array.

7898 The effect when arrays of parameters are used with an UPDATE WHERE CURRENT OF
7899 statement is implementation-defined.

7900 When an array of parameters is processed, it is implementation-defined whether one result sets
7901 and row count is available for each parameter set, or whether the result sets and row counts are
7902 combined. An application can determine the implementation's behavior by calling *SQLGetInfo*()
7903 with the SQL_PARAM_ARRAY_ROW_COUNTS option (regarding row counts) or the
7904 SQL_PARAM_ARRAY_SELECTS option (regarding result sets).

7905 In order to support arrays of parameters, the SQL_DESC_PARAMSET_SIZE statement attribute
7906 is set to specify the number of values for each parameter. If the field is greater than 1, the
7907 SQL_DESC_DATA_PTR, SQL_DESC_INDICATOR_PTR, and
7908 SQL_DESC_OCTET_LENGTH_PTR fields of the APD must point to arrays. The cardinality of
7909 each array is equal to the value of SQL_DESC_PARAMSET_SIZE.

7910 The SQL_DESC_ROWS_PROCESSED_PTR field of the APD points to a buffer in which to return
7911 the current row number. As each row of parameters is processed, this is set to the number of
7912 that row. No row number is returned if this is a null pointer. The implementation generates
7913 SQL_DESC_ROWS_PROCESSED_PTR.

7914 **Column-Wise Parameter Binding**

7915 Column-wise binding of parameters is used by setting the SQL_DESC_PARAM_BIND_TYPE
7916 statement attribute to SQL_PARAMETER_BIND_BY_COLUMN. When column-wise binding is
7917 used, all parameter values are stored in one array, and the associated data lengths are stored in
7918 another array.

7919 **Row-Wise Parameter Binding**

7920 Row-wise binding can be used for parameter buffers. When row-wise binding is used, all
7921 parameter values used in a SQL statement, and the associated data lengths, are stored in a
7922 structure. An array of structures can be allocated to specify multiple sets of parameters for bulk
7923 operations, such as bulk inserts.

7924 An application assigns buffers for row-wise bound parameters by allocating an array of
7925 structures and manipulating the application and IPDs. For more information, see **Row-wise**
7926 **Binding** on page 111.

7927 **Error Information**

7928 If an implementation does not implement parameter arrays as batches (the
7929 SQL_PARAM_ARRAY_ROW_COUNTS option of *SQLGetInfo*() is equal to
7930 SQL_PARC_NO_BATCH), error situations are handled as if one statement was executed. If the
7931 implementation does implement parameter arrays as batches, an application can use the
7932 SQL_DESC_ARRAY_STATUS_PTR header field of the IPD to determine which parameter of an
7933 SQL statement, or which parameter in an array of parameters, caused *SQLExecDirect*() or
7934 *SQLExecute*() to return an error. This field contains status information for each row of parameter
7935 values. If the field indicates that an error has occurred, fields in the diagnostic data structure will
7936 indicate the row and parameter number of the parameter that failed. The number of elements in
7937 the array will be defined by the SQL_DESC_ARRAY_SIZE header field in the IPD.

7938 When *SQLExecute*() or *SQLExecDirect*() returns SQL_ERROR, the elements in the array pointed
7939 to by the SQL_DESC_ARRAY_STATUS_PTR field of the IPD will contain SQL_PARAM_ERROR,
7940 SQL_PARAM_SUCCESS, SQL_PARAM_SUCCESS_WITH_INFO, SQL_PARAM_UNUSED, or
7941 SQL_PARAM_DIAG_UNAVAILABLE.

7942 For each SQL_PARAM_ERROR in this array, the diagnostic data structure contains one or more
7943 status records. The SQL_DESC_ROW_NUMBER field of the structure indicates the row number
7944 of the parameter values that caused the error. If it is possible to determine the particular
7945 parameter in a row of parameters that caused the error, then the parameter number is stored in
7946 the SQL_DIAG_COLUMN_NUMBER field.

7947 SQL_PARAM_UNUSED is entered when a parameter has not been used because an error
7948 occurred in an earlier parameter that forced *SQLExecute*() or *SQLExecDirect*() to abort. For
7949 example, if there are 50 parameters, and an error occurred while executing the 40th set of
7950 parameters that caused *SQLExecute*() or *SQLExecDirect*() to abort, then the implementation
7951 stores SQL_PARAM_UNUSED in the status array for parameters 41 through 50.

7952 SQL_PARAM_DIAG_UNAVAILABLE is stored when the implementation treats arrays of
7953 parameters as a unit and does not generate this level of error information.

7954 Some errors in the processing of a single set of parameters terminate processing of subsequent
7955 sets of parameters in the array. Other errors do not affect the processing of subsequent
7956 parameters. It is implementation-defined which errors stop processing. If processing is not
7957 stopped, all parameters in the array are processed, SQL_SUCCESS_WITH_INFO is returned as a
7958 result of the error, and the buffer defined by SQL_ATTR_PARAMS_PROCESSED_PTR is set to
7959 the total number of parameters processed, including error rows. (This is the value pointed to by
7960 SQL_ATTR_PARAMSET_SIZE.)

7961 When *SQLExecute*() or *SQLExecDirect*() returns before completing the processing of all
7962 parameter sets in a parameter array, such as when it returns SQL_ERROR or SQL_NEED_DATA,
7963 the status array contains elements for those parameters that have already been processed. The
7964 location pointed to by the SQL_DESC_ROWS_PROCESSED_PTR field in the IPD will contain
7965 the row number in the parameter array that caused the SQL_ERROR or SQL_NEED_DATA error
7966 code. When an array of parameters is sent to a SELECT statement, status array values are
7967 available after all result sets are fetched. On some implementations, they may be available after
7968 the statement has been executed.

7969 **Ignoring a Set of Parameters**

7970 The application parameter status array can be used to direct the implementation to ignore a set
7971 of bound parameters in a SQL statement. The application performs the following steps:

7972 • Call *SQLSetDescField*() to set the SQL_DESC_ARRAY_STATUS_PTR header field of the APD
7973 to point to an array of SQLUSMALLINTs to contain status information. This field can also be
7974 set by calling *SQLSetStmtAttr*() with an Attribute of SQL_ATTR_PARAM_STATUS_PTR,
7975 which allows an application to set the field without obtaining a descriptor handle.

7976 • For each row to be excluded from statement execution, set the corresponding element of the
7977 application row status array to SQL_PARAM_IGNORE. For other rows, set the element to
7978 SQL_PARAM_PROCEED (which is defined as 0 in the header file).

7979 • Call *SQLExecute*() to execute the prepared statement.

7980 The application parameter status array is a null pointer by default. If it is a null pointer when a
7981 prepared statement is executed, then all rows are updated, as if all elements were set to
7982 SQL_PARAM_PROCEED.

7983 Enabling inclusion of a row using the application parameter status array does not guarantee that
7984 the operation occurs on that row.

7985 An application can set the SQL_DESC_ARRAY_STATUS_PTR in the APD to point to the same
7986 array as that pointed to by the SQL_DESC_ARRAY_STATUS_PTR field in the IRD. This is useful
7987 when binding parameters to row data. Parameters can then be ignored according to the status of
7988 the row data. The following codes cause a parameter in a SQL statement to be ignored, in
7989 addition to SQL_PARAM_IGNORE: SQL_ROW_DELETED, SQL_ROW_UPDATED, and
7990 SQL_ROW_ERROR. The following codes cause a SQL statement to proceed, in addition to
7991 SQL_PARAM_PROCEED: SQL_ROW_SUCCESS, SQL_ROW_SUCCESS_WITH_INFO, and
7992 SQL_ROW_ADDED.

7993 **Rebinding with Offsets**

7994 When row-wise binding is used, rebinding of parameters can be performed by either making
7995 another call to *SQLBindParameter*(), or adding an offset to the binding pointers to rebind the
7996 parameter. This is especially useful when an application has a buffer area setup that is capable
7997 of containing many parameters, but a call to *SQLExecDirect*() or *SQLExecute*() uses only a few of
7998 the parameters. The remaining space in the buffer area can be used for the next set of parameters
7999 by modifying the existing binding by an offset.

8000 The SQL_DESC_BIND_OFFSET_PTR header field in the APD points to the bind offset. If the
8001 field is non-null, the implementation dereferences the pointer and if none of the values in the
8002 SQL_DESC_DATA_PTR, SQL_DESC_INDICATOR_PTR, and
8003 SQL_DESC_OCTET_LENGTH_PTR fields is a null pointer, adds the dereferenced value to those
8004 fields in the descriptor records at execution time. The new pointer values are used when the SQL
8005 statements are executed. The offset remains valid after rebinding. Because
8006 SQL_DESC_BIND_OFFSET_PTR is a pointer to the offset, rather than the offset itself, an
8007 application can change the offset directly, without having to call *SQLSetDescField*() or
8008 *SQLSetDescRec*() to change the descriptor field. The pointer is set to null by default. The
8009 SQL_DESC_BIND_OFFSET_PTR field of the ARD can be set by a call to *SQLSetDescField*() or by
8010 a call to *SQLSetStmtAttr*() with an *fAttribute* of SQL_ATTR_PARAM_BIND_OFFSET_PTR.

8011 The bind offset is always added directly to the values in the SQL_DESC_DATA_PTR,
8012 SQL_DESC_INDICATOR_PTR, and SQL_DESC_OCTET_LENGTH_PTR fields. If the offset is
8013 changed to a different value, the new value is still added directly to the value in each descriptor
8014 field. The new offset is not added to the field value plus any earlier offsets.

8015 **Descriptors**

8016 How a parameter is bound is determined by fields of the APD and IPDs. The arguments in
8017 *SQLBindParameter*() are used to set those descriptor fields. The fields can also be set by
8018 *SQLSetDescField*() although *SQLBindParameter*() is more efficient to use because the application
8019 does not have to obtain a descriptor handle to call *SQLBindParameter*().

8020 **Caution:** Calling *SQLBindParameter*() for one statement affects other statements if the ARD
8021 associated with the statement is explicitly allocated and is also associated with other statements.
8022 Any modifications made to a descriptor with *SQLBindParameter*() apply to all statements with
8023 which the descriptor is associated. To prevent this effect, the application must dissociate this
8024 descriptor from the other statements before calling *SQLBindParameter*().

8025 Conceptually, *SQLBindParameter*() performs the following steps in sequence:

- 8026 calls *SQLGetStmtAttr*() to obtain the application parameter descriptor handle
- 8027 calls *SQLGetDescField*() to get this descriptor's COUNT field, and if *ColumnNumber* exceeds 8028 the value of COUNT, calls *SQLSetDescField*() to increase the value of COUNT to 8029 *ColumnNumber*
- 8030 calls *SQLSetDescField*() multiple times to assign values to the following fields of the 8031 application parameter descriptor:
  - 8032 — sets TYPE to the value of *ValueType*
  - 8033 — sets DATA_PTR to the value of *ParameterValue*
  - 8034 — sets OCTET_LENGTH_PTR to the value of *StrLen_or_Ind*
  - 8035 — sets INDICATOR_PTR also to the value of *StrLen_or_Ind*.

  8036 The *StrLen_or_Ind* parameter specifies both the indicator information and the length for the 8037 parameter value.
- 8038 calls *SQLGetStmtAttr*() to obtain the implementation parameter descriptor handle
- 8039 calls *SQLGetDescField*() to get this descriptor's COUNT field, and if *ColumnNumber* exceeds 8040 the value of COUNT, calls *SQLSetDescField*() to increase the value of COUNT to 8041 *ColumnNumber*
- 8042 calls *SQLSetDescField*() multiple times to assign values to the following fields of the 8043 implementation parameter descriptor:
  - 8044 — sets TYPE to the value of *ParameterType*, except that if *ParameterType* is one of the concise 8045 identifiers of a date/time or interval subtype (see **Data Type Identification in** 8046 **Descriptors** on page 574), it sets TYPE to SQL_DATETIME or SQL_INTERVAL 8047 respectively, and sets DATETIME_INTERVAL_CODE to the corresponding date/time or 8048 interval subcode
  - 8049 — sets one or more of LENGTH, PRECISION and DATETIME_INTERVAL_PRECISION, as 8050 appropriate for *ParameterType*[30]
  - 8051 — sets SCALE to the value of *DecimalDigits*.

8052 If the call to *SQLBindParameter*() fails, the content of any descriptor fields that it would have set 8053 are undefined.

8054 SEE ALSO

8055 **For information about** **See**

8056 ________________

8057 30. For date/time data types, LENGTH is set to the total length in characters of a literal of that type. If the type has a seconds component, then PRECISION is set to *DecimalDigits*; otherwise, PRECISION is set to 0. If this result is positive, then the total 8058 length includes the length of the fractional part of the seconds component.

8059 For interval data types, LENGTH is set to *ColumnSize*. If the type has a seconds component, then PRECISION is set to *DecimalDigits*; otherwise, PRECISION is set to 0. DATETIME_INTERVAL_PRECISION is set to the leading field precision. This 8060 is the number of columns of *ColumnSize* that are available as the leading field, after subtracting the size of the fixed fields, and the size of the fractional part of the seconds component, if one is indicated by the subtype and by the value of *DecimalDigits*.

Returning information about a parameter in a statement	*SQLDescribeParam*()
Executing an SQL statement	*SQLExecDirect*()
Executing a prepared SQL statement	*SQLExecute*()
Returning the number of statement parameters	*SQLNumParams*()
Returning the next parameter to send data for	*SQLParamData*()
Sending parameter data at execution time	*SQLPutData*()

CHANGE HISTORY

Version 2

Revised generally. See **Alignment with Popular Implementations** on page 2.

8070 **NAME**

8071 SQLBrowseConnect — Iterative method of discovering and enumerating the attributes and
8072 attribute values required to connect to a data source.

8073 **SYNOPSIS**

```
8074 SQLRETURN SQLBrowseConnect(
8075   SQLHDBC ConnectionHandle,
8076   SQLCHAR * InConnectionString,
8077   SQLSMALLINT StringLength1,
8078   SQLCHAR * OutConnectionString,
8079   SQLSMALLINT BufferLength,
8080   SQLSMALLINT * StringLength2Ptr);
```

8081 **ARGUMENTS**

8082 *ConnectionHandle* [Input]
8083 Connection handle.

8084 *InConnectionString* [Input]
8085 Browse request connection string; see **InConnectionString Argument** on page 236.

8086 *StringLength1* [Input]
8087 Length of **InConnectionString*.

8088 *OutConnectionString* [Output]
8089 Pointer to a buffer in which to return the browse result connection string; see
8090 **OutConnectionString Argument** on page 236.

8091 *BufferLength* [Input]
8092 Length of the **OutConnectionString* buffer.

8093 *StringLength2Ptr* [Output]
8094 The total number of octets (excluding the null terminator) available to return in
8095 **OutConnectionString*. If the number of octets available to return is greater than or equal to
8096 *BufferLength*, the connection string in **OutConnectionString* is truncated to *BufferLength*
8097 minus the length of a null terminator.

8098 **RETURN VALUE**

8099 SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NEED_DATA, SQL_ERROR, or
8100 SQL_INVALID_HANDLE.

8101 **DIAGNOSTICS**

8102 When *SQLBrowseConnect*() returns SQL_ERROR, SQL_SUCCESS_WITH_INFO, or
8103 SQL_NEED_DATA, an associated SQLSTATE value can be obtained by calling *SQLGetDiagRec*()
8104 with a *HandleType* of SQL_HANDLE_STMT and a *Handle* of *ConnectionHandle*. The following
8105 SQLSTATE values are commonly returned by *SQLBrowseConnect*(). The return code associated
8106 with each SQLSTATE value is SQL_ERROR, except that for SQLSTATE values in class 01, the
8107 return code is SQL_SUCCESS_WITH_INFO except as noted below.

8108 01000 — General warning
8109 Implementation-defined informational message.

8110 01004 — String data, right truncation
8111 The buffer **OutConnectionString* was not large enough to return the entire browse result
8112 connection string, so the string was truncated.

8113 The buffer **StringLength2Ptr* contains the length of the untruncated browse result
8114 connection string.

8115 01S00 — Invalid connection string attribute
8116 An invalid attribute keyword was specified in *InConnectionString*. (The function returns
8117 SQL_NEED_DATA.)

8118 An attribute keyword was specified in *InConnectionString* that does not apply to the current
8119 connection level. (The function returns SQL_NEED_DATA.)

8120 01S02 — Attribute value changed
8121 The data source did not support the specified value of the *ValuePtr* argument in
8122 *SQLSetConnectAttr*() and substituted a similar value.

8123 08001 — Client unable to establish connection
8124 The implementation could not establish a connection to the data source.

8125 08002 — Connection name in use
8126 The specified connection had already been used to establish a connection with a data source
8127 and the connection was open.

8128 08004 — Data source rejected the connection
8129 The data source rejected the establishment of the connection for implementation-defined
8130 reasons.

8131 08S01 — Communication link failure
8132 The communication link to the data source failed before the function completed processing.

8133 28000 — Invalid authorization specification
8134 Either the user identifier or the authorization string or both as specified in
8135 *InConnectionString* violated restrictions defined by the data source.

8136 HY000 — General error
8137 An error occurred for which there was no specific SQLSTATE and for which no
8138 implementation-specific SQLSTATE was defined. The error message returned by
8139 *SQLGetDiagRec*() in the **MessageText* buffer describes the error and its cause.

8140 HY001 — Memory allocation error
8141 The implementation failed to allocate memory required to support execution or completion
8142 of the function.

8143 HY090 — Invalid string or buffer length
8144 *StringLength1* was less than 0 and was not equal to SQL_NTS.

8145 *BufferLength* was less than 0.

8146 HYT00 — Timeout expired
8147 The login timeout period expired before the connection to the data source completed. The
8148 timeout period is set through *SQLSetConnectAttr*(), SQL_ATTR_LOGIN_TIMEOUT.

8149 HYT01 — Connection timeout expired
8150 The connection timeout period expired before the data source responded to the request. The
8151 connection timeout period is set through *SQLSetConnectAttr*(),
8152 SQL_ATTR_CONNECTION_TIMEOUT.

8153 IM001 — Function not supported
8154 The function is not supported on the current connection to the data source.

8155 IM002 — Data source not found and no default driver specified
8156 The data source name specified in the browse request connection string (*InConnectionString*)
8157 was not found in the system information, nor was there a default data source specification.

8158 **COMMENTS**

8159 The application uses *SQLBrowseConnect*() to perform an iterative process resulting in connection

to a data source. Each call to *SQLBrowseConnect*() informs the application of the next level of detail that the application must specify. When the application has specified sufficient information, *SQLBrowseConnect*() returns SQL_SUCCESS or SQL_SUCCESS_WITH_INFO, provides in *OutConnectionString* a completed connection string, and completes a connection to the data source.

InConnectionString Argument

InConnectionString contains a browse request connection string with the following syntax:

```
connection-string ::= attribute[;] | attribute; connection-string

attribute ::= attribute-keyword=attribute-value

attribute-keyword ::= DSN | UID | PWD
                    | implementation-defined-attribute-keyword

attribute-value ::= character-string
implementation-defined-attribute-keyword ::= identifier
```

where *character-string* has zero or more characters; *identifier* has one or more characters; *attribute-keyword* is not case-sensitive; *attribute-value* may be case-sensitive; and the value of the **DSN** keyword does not consist solely of blanks. Keywords and attribute values should not contain the characters **[ ] { } ( ) , ; ? * = ! @** \

If in *InConnectionString* any keywords are repeated, or if the same or different keywords are used in ways that would be contradictory, the implementation uses the one that appears first.

OutConnectionString Argument

Each call to *SQLBrowseConnect*() returns in *OutConnectionString* a browse result connection string. This is a list of connection attributes. Each connection attribute consists of an attribute keyword and a corresponding attribute value.

The browse result connection string has the following syntax:

```
connection-string ::= attribute[;] | attribute; connection-string

attribute ::= [*]attribute-keyword=attribute-value
attribute-keyword ::= XDBC-attribute-keyword
          | implementation-defined-attribute-keyword

XDBC-attribute-keyword = {UID | PWD}[:localized-identifier]
implementation-defined-attribute-keyword ::= identifer[:localized-identifier]

attribute-value ::= {attribute-value-list} | ?
```

(The braces are returned literally in *OutConnectionString*.)

```
attribute-value-list ::= character-string [:localized-character-string]
     | character-string [:localized-character-string], attribute-value-list
```

where *character-string* and *localized-character* string have zero or more characters; *identifier* and *localized-identifier* have one or more characters; *attribute-keyword* is not case-sensitive; and *attribute-value* may be case sensitive. Keywords, localized identifiers, and attribute values should not contain the characters **[ ] { } ( ) , ; ? * = ! @** \

The browse result connection string syntax is used according to the following semantic rules:

- If an asterisk (*) precedes an *attribute-keyword*, the attribute is optional: The application is not required to provide a value for this attribute in the next call to *SQLBrowseConnect*().

- The attribute keywords **UID** and **PWD** have the same meaning as defined in *SQLDriverConnect*().

- An *implementation-defined-attribute-keyword* names the kind of attribute for which an attribute value may be supplied. For example, it might be **SERVER**, **DATABASE**, **HOST**, or **DBMS**.

- All attribute keywords include a localized or user-friendly version of the keyword. Applications might use this to interact with the user. However, the application must use the attribute keyword, not the localized version, when forming *InConnectionString* for the next call to *SQLBrowseConnect*().

- An *attribute-value-list* enumerates actual values valid for the corresponding *attribute-keyword*. For example, it might be a list of valid data sources. The application must select one element of the list when forming *InConnectionString* for the next call.

  If the *attribute-value* is a question mark, a single value corresponds to the *attribute-keyword*. For example, *SQLBrowseConnect*() might provide `UID=?; PWD=?` to report that the selected data source requires that a user identifier and password be specified. The application specifies a value by substituting it for the question mark when forming *InConnectionString* for the next call.

- Each call to *SQLBrowseConnect*() returns only the information the application requires to form the browse request string for the next call. The implementation associates sufficient information with *ConnectionHandle* to be able to determine the correct context for each call.

Using SQLBrowseConnect()

SQLBrowseConnect() requires an allocated connection handle. It is undefined whether the implementation establishes a connection with the data source during the browsing process. If *SQLBrowseConnect*() returns SQL_ERROR, it terminates any outstanding connections it has made and returns *ConnectionHandle* to the unconnected state.

When *SQLBrowseConnect*() is called for the first time on a connection, *InConnectionString* must contain the **DSN** keyword.

On each call to *SQLBrowseConnect*(), the application specifies the connection attribute values in *InConnectionString*. The implementation returns successive levels of attributes and attribute values in *OutConnectionString*; it returns SQL_NEED_DATA as long as there are connection attributes that have not yet been enumerated in *InConnectionString*. The application uses the contents of *OutConnectionString* to build *InConnectionString* for the next call to *SQLBrowseConnect*(). The application must include all mandatory attributes (those not preceded by an asterisk in *OutConnectionString*) in the next call to *SQLBrowseConnect*().

The application cannot specify different attribute values from those it specified in previous calls during the same browse process. If, before completing the dialogue, the application elects to change its selection of a data source or connection parameters, it must terminate the browse process (see below) and start over.

When the application calls *SQLBrowseConnect*() with a sufficiently-complete *InConnectionString* to establish a connection, *SQLBrowseConnect*() establishes the connection, returns SQL_SUCCESS, and returns in *OutConnectionString* a connection string that the application could provide to *SQLDriverConnect*() to establish a future connection to the same data source with the same connection parameters. However, this string is not useful to shorten the dialogue in future calls to *SQLBrowseConnect*(); to achieve the same connection again using *SQLBrowseConnect*(), the entire sequence of calls must be repeated.

8245 *SQLBrowseConnect*() returns SQL_NEED_DATA if there are recoverable, nonfatal errors during
8246 the browse process (for example, if the application supplies an invalid password or an invalid
8247 attribute keyword). When SQL_NEED_DATA is returned and the browse result connection
8248 string is unchanged, an error has occurred and the application can call *SQLGetDiagRec*() to
8249 return the SQLSTATE for browse-time errors. This lets the application correct the attribute and
8250 continue the browse.

8251 An application may terminate the browse process at any time by calling *SQLDisconnect*(). The
8252 implementation terminates any outstanding connections and returns *ConnectionHandle* to the
8253 unconnected state.

8254 **SEE ALSO**

8255	**For information about**	**See**
8256	Overview of function	Section 6.4.5 on page 62
8257	Allocating a connection handle	*SQLAllocHandle*()
8258	Connecting to a data source	*SQLConnect*()
8259	Disconnecting from a data source	*SQLDisconnect*()
8260 8261	Connecting to a data source using a connection string or dialog box	*SQLDriverConnect*()
8262	Freeing a connection handle	*SQLFreeHandle*()

8263 **CHANGE HISTORY**

8264 **Version 2**

8265 Function added in this version.

8266 **NAME**

8267 SQLBulkOperations — Perform bulk insertions and bulk bookmark operations, including
8268 update, delete, and fetch by bookmark.

8269 **SYNOPSIS**

```
8270 SQLRETURN SQLBulkOperations(
8271   SQLHSTMT StatementHandle,
8272   SQLUSMALLINT Operation);
```

8273 **ARGUMENTS**

8274 *StatementHandle* [Input]
8275 Statement handle.

8276 *Operation* [Input]
8277 Operation to perform:

8278 • SQL_ADD
8279 • SQL_UPDATE_BY_BOOKMARK
8280 • SQL_DELETE_BY_BOOKMARK
8281 • SQL_FETCH_BY_BOOKMARK

8282 **RETURN VALUE**

8283 SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NEED_DATA, SQL_STILL_EXECUTING,
8284 SQL_ERROR, or SQL_INVALID_HANDLE.

8285 **DIAGNOSTICS**

8286 When *SQLBulkOperations*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated
8287 SQLSTATE value can be obtained by calling *SQLGetDiagRec*() with a *HandleType* of
8288 SQL_HANDLE_STMT and a *Handle* of *StatementHandle*. The following SQLSTATE values are
8289 commonly returned by *SQLBulkOperations*().

8290 The return code associated with each SQLSTATE value is SQL_ERROR, except that for
8291 SQLSTATE values in class 01, the return code is SQL_SUCCESS_WITH_INFO, and except that, if
8292 the row-set size is greater than 1 and the operation was applied to at least one row successfully,
8293 the return code is SQL_SUCCESS_WITH_INFO.

8294 01000 — General warning
8295 Implementation-defined informational message.

8296 01004 — String data, right truncation
8297 String or binary data returned for a column or columns with a data type of SQL_C_CHAR
8298 or SQL_C_BINARY resulted in the truncation of non-blank character or non-NULL binary
8299 data.

8300 01S01 — Error in row
8301 The SQL_ATTR_ROW_ARRAY_SIZE statement attribute was greater than 1, *Operation* was
8302 SQL_ADD, and an error occurred in one or more rows while performing the operation, but
8303 at least one row was successfully added.

8304 01S07 — Fractional truncation
8305 *Operation* was SQL_FETCH_BY_BOOKMARK, the data type of the application buffer was
8306 not SQL_C_CHAR or SQL_C_BINARY, and the data returned to application buffers for one
8307 or more columns was truncated. For numeric data types, the fractional part of the number
8308 was truncated. For time, timestamp, and interval data types containing a time component,
8309 the fractional portion of the time was truncated.

8310 07006 — Restricted data type attribute violation
8311 *Operation* was SQL_FETCH_BY_BOOKMARK, and the data value of a column in the result
8312 set could not be converted to the data type specified by *TargetType* in the call to

8313 *SQLBindCol*().

8314 *Operation* was SQL_UPDATE_BY_BOOKMARK or SQL_ADD, and the data value in the
8315 application buffers could not be converted to the data type of a column in the result set.

8316 07009 — Invalid descriptor index
8317 *Operation* was SQL_ADD and a column was bound with a column number greater than the
8318 number of columns in the result set.

8319 21S02 — Degree of derived table does not match column list
8320 *Operation* was SQL_UPDATE_BY_BOOKMARK and no columns were updatable because all
8321 columns were either unbound, read-only, or the value in the bound length/indicator buffer
8322 was SQL_COLUMN_IGNORE.

8323 22001 — String data, right truncation
8324 The assignment of a character or binary value to a column in the result set resulted in the
8325 truncation of non-blank (for characters) or non-null (for binary) characters or octets.

8326 22003 — Numeric value out of range
8327 *Operation* was SQL_ADD or SQL_UPDATE_BY_BOOKMARK, and the assignment of a
8328 numeric value to a column in the result set caused the whole (as opposed to fractional) part
8329 of the number to be truncated.

8330 *Operation* was SQL_FETCH_BY_BOOKMARK, and returning the numeric value for one or
8331 more bound columns would have caused a loss of significant digits.

8332 22007 — Invalid date/time format
8333 *Operation* was SQL_ADD or SQL_UPDATE_BY_BOOKMARK, and an invalid date or
8334 timestamp value was assigned to a column in the result set.

8335 *Operation* was SQL_FETCH_BY_BOOKMARK, and an invalid date or timestamp value
8336 would have been returned for one or more bound columns.

8337 22008 — Date/time field overflow
8338 *Operation* was SQL_ADD or SQL_UPDATE_BY_BOOKMARK, and the performance of
8339 date/time arithmetic on data being sent to or retrieved from the result set resulted in a
8340 date/time field (that is, the year, month, day, hour, minute, or second field) of the result
8341 being outside the permissible range of values for the field, or being invalid based on the
8342 natural rules for date/times based on the Gregorian calendar.

8343 22015 — Interval field overflow
8344 *Operation* was SQL_ADD or SQL_UPDATE_BY_BOOKMARK, and the assignment of an
8345 exact numeric value to a column in the result set with an interval data type caused a loss of
8346 significant digits.

8347 *Operation* was SQL_ADD or SQL_UPDATE_BY_BOOKMARK, and the assignment of an
8348 interval value to a column in the result set with an interval data type caused a loss of
8349 significant digits in the leading field of the interval.

8350 *Operation* was SQL_ADD or SQL_UPDATE_BY_BOOKMARK, and there was no
8351 representation of the data in the interval data type of the result set.

8352 *Operation* argument was SQL_FETCH_BY_BOOKMARK, and returning an exact numeric
8353 value to an application buffer with an interval data type caused a loss of significant digits.

8354 *Operation* was SQL_FETCH_BY_BOOKMARK, and returning an interval value to an
8355 application buffer with an interval data type caused a loss of significant digits in the leading
8356 field of the interval.

8357 *Operation* was SQL_FETCH_BY_BOOKMARK, and there was no representation of the data
8358 in the interval C structure in the application buffer.

8359 22018 — Invalid character value for cast specification
8360 *Operation* was SQL_FETCH_BY_BOOKMARK, a character column in the result set was
8361 bound to an exact numeric or approximate numeric C buffer, and a character value in the
8362 result set could not be cast to a valid exact numeric or approximate numeric value,
8363 respectively.

8364 *Operation* was SQL_FETCH_BY_BOOKMARK, a character column in the result set was
8365 bound to a date, time, timestamp, or interval C buffer, and a character value in the result set
8366 could not be cast to a valid date, time, timestamp, or interval value, respectively.

8367 *Operation* was SQL_ADD or SQL_UPDATE_BY_BOOKMARK, a character column in an
8368 application buffer was bound to an exact numeric or approximate numeric data type in the
8369 result set, and a value in the application buffer could not be cast to a valid exact numeric or
8370 approximate numeric value, respectively.

8371 *Operation* was SQL_ADD or SQL_UPDATE_BY_BOOKMARK, a character column in an
8372 application buffer was bound to a date, time, timestamp, or interval data type in the result
8373 set, and a value in the application buffer could not be cast to a valid date, time, timestamp,
8374 or interval value, respectively.

8375 23000 — Integrity constraint violation
8376 *Operation* was SQL_ADD, SQL_DELETE_BY_BOOKMARK, or
8377 SQL_UPDATE_BY_BOOKMARK, and an integrity constraint was violated.

8378 *Operation* was SQL_ADD and a column that was not bound is defined as NOT NULL or has
8379 no default.

8380 *Operation* was SQL_ADD, the length specified in the bound *StrLen_or_IndPtr* buffer was
8381 SQL_COLUMN_IGNORE, and the column did not have a default value.

8382 24000 — Invalid cursor state
8383 *StatementHandle* was in an executed state but no result set was associated with
8384 *StatementHandle*.

8385 42000 — Syntax error or access violation
8386 The data source was unable to lock the row as needed to perform the operation requested in
8387 *Operation*.

8388 HY000 — General error
8389 An error occurred for which there was no specific SQLSTATE and for which no
8390 implementation-specific SQLSTATE was defined. The error message returned by
8391 *SQLGetDiagRec*() in the **MessageText* buffer describes the error and its cause.

8392 HY001 — Memory allocation error
8393 The implementation failed to allocate memory required to support execution or completion
8394 of the function.

8395 HY008 — Operation canceled
8396 Asynchronous processing was enabled for *StatementHandle*. The function was called and
8397 before it completed execution, *SQLCancel*() was called on *StatementHandle*. The function
8398 was then called again on *StatementHandle*.

8399 The function was called and, before it completed execution, *SQLCancel*() was called on
8400 *StatementHandle* from a different thread in a multithread application.

8401 HY009 — Invalid use of null pointer
8402 *Operation* was SQL_USE_ROW_OPERATION_PTR, and the
8403 SQL_ATTR_ROW_STATUS_PTR statement attribute was a null pointer.

HY010 — Function sequence error
StatementHandle was not in an executed state. The function was called without first calling *SQLExecDirect*(), *SQLExecute*(), or a catalog function.

An asynchronously executing function (not this one) was called for the StatementHandle and was still executing when this function was called.

SQLBulkOperations(), *SQLExecDirect*(), *SQLExecute*(), or *SQLSetPos*() was called for *StatementHandle* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.

HY090 — Invalid string or buffer length
Operation was SQL_ADD or SQL_UPDATE_BY_BOOKMARK, a data value was a null pointer, and the column length value was not 0, SQL_DATA_AT_EXEC, SQL_COLUMN_IGNORE, SQL_NULL_DATA, or less than or equal to SQL_LEN_DATA_AT_EXEC_OFFSET.

Operation was SQL_ADD or SQL_UPDATE_BY_BOOKMARK, a data value was not a null pointer, and the column length value was less than 0, but not equal to SQL_DATA_AT_EXEC,SQL_COLUMN_IGNORE, SQL_NTS, or SQL_NULL_DATA, or less than or equal to SQL_LEN_DATA_AT_EXEC_OFFSET. (This error is reported only if the application data type is SQL_C_BINARY or SQL_C_CHAR.)

The value in a length/indicator buffer was SQL_DATA_AT_EXEC;the SQL type was either SQL_LONGVARCHAR, SQL_WLONGVARCHAR, SQL_LONGVARBINARY, or a long, data-source-specific data type; and the SQL_NEED_LONG_DATA_LEN option in *SQLGetInfo*() was "Y".

HY092 — Invalid attribute identifier
Operation was invalid.

Operation was SQL_ADD, SQL_UPDATE_BY_BOOKMARK, or SQL_DELETE_BY_BOOKMARK, and the SQL_ATTR_CONCURRENCY statement attribute was set to SQL_CONCUR_READ_ONLY.

HYC00 — Optional feature not implemented
The implementation does not support the operation requested in *Operation*.

HYT00 — Timeout expired
The query timeout period expired before the data source returned the result set. The timeout period is set through *SQLSetStmtAttr*() with an *Attribute* of SQL_ATTR_QUERY_TIMEOUT.

HYT01 — Connection timeout expired
The connection timeout period expired before the data source responded to the request. The connection timeout period is set through *SQLSetConnectAttr*(), SQL_ATTR_CONNECTION_TIMEOUT.

IM001 — Function not supported
The function is not supported on the current connection to the data source.

COMMENTS

An application uses *SQLBulkOperations*() to perform the following operations on the table that corresponds to the current query:

- Add new rows.
- Update a set of rows where each row is identified by a bookmark.
- Delete a set of rows where each row is identified by a bookmark.
- Fetch a set of rows where each row is identified by a bookmark.

After a call to *SQLBulkOperations*(), the cursor position is undefined. The application has to call *SQLFetchScroll*() in order to set the cursor position. An application should only call *SQLFetchScroll*() with *FetchOrientation* of SQL_FETCH_FIRST, SQL_FETCH_LAST, SQL_FETCH_ABSOLUTE, or SQL_FETCH_BOOKMARK. The cursor position is undefined if the application calls *SQLFetch*(), or *SQLFetchScroll*() with *FetchOrientation* of SQL_FETCH_PRIOR, SQL_FETCH_NEXT, or SQL_FETCH_RELATIVE.

Column can be ignored in bulk operations performed by a call to *SQLBulkOperations*() by setting the column length/indicator buffer in the call to *SQLBindCol*() to SQL_COLUMN_IGNORE.

Performing Bulk Inserts

To insert data with *SQLBulkOperations*(), an application:

- Executes a query that returns a result set.
- Sets the SQL_ATTR_ROW_ARRAY_SIZE statement attribute to the number of rows that it wants to insert.
- Calls *SQLBindCol*() to bind the data that it wants to insert. The data is bound to an array with a size equal to the value of SQL_ATTR_ROW_ARRAY_SIZE.
- Optionally sets the SQL_ATTR_ROW_STATUS_PTR statement attribute to point to an array of elements with a size equal to the value of SQL_ATTR_ROW_ARRAY_SIZE. If the application does not do this, then it must set SQL_ATTR_ROW_STATUS_PTR to NULL if the value of SQL_ATTR_ROW_ARRAY_SIZE is greater than or equal to the current row-set size. Failing to do so can result in a crash.
- Calls *SQLBulkOperations*(*StatementHandle*, SQL_ADD) to perform the insertion.
- If the application has set the SQL_ATTR_ARRAY_STATUS_PTR statement attribute, then it can inspect this array to see the result of the operation.

The application need not set the SQL_ATTR_OPERATION_PTR statement attribute because it is not used. The application selects the rows it wants to add by copying only those rows into the bound data array.

If an application binds column 0 before calling *SQLBulkOperations*() with *Operation* of SQL_ADD, the implementation updates the bound column 0 buffers with the bookmark values for the newly inserted row. For this to occur, the application must have set SQL_ATTR_USE_BOOKMARKS statement attribute to SQL_UB_VARIABLE before executing the statement.

Long data can be added by *SQLBulkOperations*(). The application need not call *SQLFetch*() or *SQLFetchScroll*() before calling *SQLBulkOperations*().

It is implementation-defined what happens if the application calls *SQLBulkOperations*() with *Operation* of SQL_ADD on a cursor that contains duplicate columns.

Performing Bulk Updates Using Bookmarks

To perform bulk updates with *SQLBulkOperations*(), an application:

- Sets the SQL_ATTR_USE_BOOKMARKS statement attribute to SQL_UB_VARIABLE.
- Executes a query that returns a result set.
- Sets the SQL_ATTR_ROW_ARRAY_SIZE statement attribute to the number of rows that it wants to update.
- Calls *SQLBindCol*() to bind the data that it wants to update. The data is bound to an array with a size equal to the value of SQL_ATTR_ROW_ARRAY_SIZE. It also calls *SQLBindCol*()

8493 to bind column 0 (the bookmark column).

8494 • Copies the bookmarks for rows that it is interested in updating into the array bound to
8495 column 0.

8496 • Updates the data in the bound buffers.

8497 • Optionally sets the SQL_ATTR_ROW_STATUS_PTR statement attribute to point to an array
8498 of elements with a size equal to the value of SQL_ATTR_ROW_ARRAY_SIZE. If the
8499 application does not do this, then it must set SQL_ATTR_ROW_STATUS_PTR to NULL if the
8500 value of SQL_ATTR_ROW_ARRAY_SIZE is greater than or equal to the current row-set size.
8501 Failing to do so can result in a crash.

8502 • Calls *SQLBulkOperations*(*StatementHandle*, SQL_UPDATE_BY_BOOKMARK).

8503 • If the application has set the SQL_ATTR_ARRAY_STATUS_PTR statement attribute, then it
8504 can inspect this array to see the result of the operation.

8505 • Optionally calls *SQLBulkOperations*(*StatementHandle*, SQL_FETCH_BY_BOOKMARK) to
8506 fetch data into the bound application buffers to verify that the update has occurred.

8507 The application need not set the SQL_ATTR_OPERATION_PTR statement attribute because it is
8508 not used. The application selects the rows it wants to update by copying only the bookmarks for
8509 those rows into the bound bookmark array.

8510 Bulk updates performed by *SQLBulkOperations*() can include long data.

8511 If bookmarks persist across cursors, then the application need not call *SQLFetch*() or
8512 *SQLFetchScroll*() before updating by bookmarks. It can use bookmarks that it has stored from a
8513 previous cursor. If bookmarks do not persist across cursors, then the application has to call
8514 *SQLFetch*() or *SQLFetchScroll*() once in order to retrieve the bookmarks.

8515 It is implementation-defined what happens if *SQLBulkOperations*() with *Operation* argument of
8516 SQL_UPDATE_BY_BOOKMARK is called on a cursor that contains duplicate columns.

8517 **Performing Bulk Fetches Using Bookmarks**

8518 To perform bulk fetches with *SQLBulkOperations*(), an application:

8519 • Sets the SQL_ATTR_USE_BOOKMARKS statement attribute to SQL_UB_VARIABLE.

8520 • Executes a query that returns a result set.

8521 • Sets the SQL_ATTR_ROW_ARRAY_SIZE statement attribute to the number of rows that it
8522 wants to fetch.

8523 • Calls *SQLBindCol*() to bind the data that it wants to fetch. The data is bound to an array with
8524 a size equal to the value of SQL_ATTR_ROW_ARRAY_SIZE. It also calls *SQLBindCol*() to
8525 bind column 0 (the bookmark column).

8526 • Copies the bookmarks for rows that it is interested in fetching into the array bound to
8527 column 0.

8528 • Optionally sets the SQL_ATTR_ROW_STATUS_PTR statement attribute to point to an array
8529 of elements with a size equal to the value of SQL_ATTR_ROW_ARRAY_SIZE. If the
8530 application does not do this, then it must set SQL_ATTR_ROW_STATUS_PTR to NULL if the
8531 value of SQL_ATTR_ROW_ARRAY_SIZE is greater than or equal to the current row-set size.
8532 Failing to do so can result in a crash.

8533 • Calls *SQLBulkOperations*(*StatementHandle*, SQL_FETCH_BY_BOOKMARK).

8534 • If the application has set the SQL_ATTR_ARRAY_STATUS_PTR statement attribute, then it
8535 can inspect this array to see the result of the operation.

The application need not set the SQL_ATTR_OPERATION_PTR statement attribute because it is not used. The application selects the rows it wants to fetch by copying only the bookmarks for those rows into the bound bookmark array.

If bookmarks persist across cursors, then the application need not call *SQLFetch*() or *SQLFetchScroll*() before fetching by bookmarks. It can use bookmarks that it has stored from a previous cursor. If bookmarks do not persist across cursors, then the application has to call *SQLFetch*() or *SQLFetchScroll*() once in order to retrieve the bookmarks.

Performing Bulk Deletes Using Bookmarks

To perform bulk deletes with *SQLBulkOperations*(), an application:

- Sets the SQL_ATTR_USE_BOOKMARKS statement attribute to SQL_UB_VARIABLE.
- Executes a query that returns a result set.
- Sets the SQL_ATTR_ROW_ARRAY_SIZE statement attribute to the number of rows that it wants to delete.
- Calls *SQLBindCol*() to bind column 0 (the bookmark column).
- Copies the bookmarks for rows that it is interested in updating into the array bound to column 0.
- Optionally sets the SQL_ATTR_ROW_STATUS_PTR statement attribute to point to an array of elements with a size equal to the value of SQL_ATTR_ROW_ARRAY_SIZE. If the application does not do this, then it must set SQL_ATTR_ROW_STATUS_PTR to NULL if the value of SQL_ATTR_ROW_ARRAY_SIZE is greater than or equal to the current row-set size. Failing to do so can result in a crash.
- Calls *SQLBulkOperations*(*StatementHandle*, SQL_DELETE_BY_BOOKMARK).
- If the application has set the SQL_ATTR_ARRAY_STATUS_PTR statement attribute, then it can inspect this array to see the result of the operation.

The application need not set the SQL_ATTR_OPERATION_PTR statement attribute because it is not used. The application selects the rows it wants to delete by copying only the bookmarks for those rows into the bound bookmark array.

If bookmarks persist across cursors, then the application need not call *SQLFetch*() or *SQLFetchScroll*() before updating by bookmarks. It can use bookmarks that it has stored from a previous cursor. If bookmarks do not persist across cursors, then the application has to call *SQLFetch*() or *SQLFetchScroll*() once in order to retrieve the bookmarks.

Row Status Array

The implementation row status array contains status values for each row of data in the row-set after a call to *SQLBulkOperations*(). The implementation sets the status values in this array after a call to *SQLFetch*(), *SQLFetchScroll,*() *SQLSetPos*(), or *SQLBulkOperations*(). This array is initially populated by a call to *SQLBulkOperations*() if *SQLFetch*() or *SQLFetchScroll*() has not been called prior to *SQLBulkOperations*(). This array is pointed to by the SQL_ATTR_ROW_STATUS_PTR statement attribute. The number of elements in the row status arrays must equal the number of rows in the row-set (as defined by the SQL_ATTR_ROW_ARRAY_SIZE statement attribute). For information about this row status array, see the *SQLFetch*().

The application row status array, used to ignore a row in a bulk operation, is not used with *SQLBulkOperations*().

8578 **SEE ALSO**

For information about	See
Binding a buffer to a column in a result set	*SQLBindCol*()
Canceling statement processing	*SQLCancel*()
Fetching a block of data or scrolling through a result set	*SQLFetchScroll*()
Getting a single field of a descriptor	*SQLGetDescField*()
Getting multiple fields of a descriptor	*SQLGetDescRec*()
Setting a single field of a descriptor	*SQLSetDescField*()
Setting multiple fields of a descriptor	*SQLSetDescRec*()
Positioning the cursor, refreshing data in the row-set, or updating or deleting data in the row-set	*SQLSetPos*()
Setting a statement attribute	*SQLSetStmtAttr*()

8591 **CHANGE HISTORY**

8592 **Version 2**

8593 Function added in this version.

NAME

SQLCancel — Cancel the processing of a statement.

SYNOPSIS

```
SQLRETURN SQLCancel(
  SQLHSTMT StatementHandle);
```

ARGUMENTS

StatementHandle [Input]
Statement handle.

RETURN VALUE

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.

DIAGNOSTICS

When *SQLCancel*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling *SQLGetDiagRec*() with a *HandleType* of SQL_HANDLE_STMT and a *Handle* of *StatementHandle*. The following SQLSTATE values are commonly returned by *SQLCancel*(). The return code associated with each SQLSTATE value is SQL_ERROR, except that for SQLSTATE values in class 01, the return code is SQL_SUCCESS_WITH_INFO.

01000 — General warning
Implementation-defined informational message.

HY000 — General error
An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by *SQLGetDiagRec*() in the **MessageText* buffer describes the error and its cause.

HY001 — Memory allocation error
The implementation failed to allocate memory required to support execution or completion of the function.

HY018 — Server declined cancel request
Communication errors caused the server to decline the cancel request.

HYT01 — Connection timeout expired
The connection timeout period expired before the data source responded to the request. The connection timeout period is set through *SQLSetConnectAttr*(), SQL_ATTR_CONNECTION_TIMEOUT.

IM001 — Function not supported
The function is not supported on the current connection to the data source.

COMMENTS

An application can call *SQLCancel*() to cancel the following types of processing on a statement:

- A function running asynchronously on the statement.
- A function on a statement that needs data.
- A function running on the statement on another thread.

When *SQLCancel*() is called, diagnostic records are returned for a function running asynchronously in a statement, or for a function on a statement that needs data; diagnostic records are not returned, however, for a function running on a statement on another thread.

8636 **Canceling Asynchronous Processing**

8637 After an application calls a function asynchronously, it calls the function repeatedly to
8638 determine whether it has finished processing. If the function is still processing, it returns
8639 SQL_STILL_EXECUTING. If the function has finished processing, it returns a different code.

8640 After any call to the function that returns SQL_STILL_EXECUTING, an application can call
8641 *SQLCancel*() to cancel the function. If the cancel request is successful, *SQLCancel*() returns
8642 SQL_SUCCESS. This does not indicate that the function was actually canceled; it indicates that
8643 the cancel request was processed. The criteria under which a function is canceled are undefined.
8644 The application must continue to call the original function until the return code is not
8645 SQL_STILL_EXECUTING. If the function was successfully canceled, the return code is
8646 SQL_ERROR and SQLSTATE HY008 (Operation canceled). If the function completed its normal
8647 processing, the return code is SQL_SUCCESS or SQL_SUCCESS_WITH_INFO if the function
8648 succeeded or SQL_ERROR and a SQLSTATE other than HY008 (Operation canceled) if the
8649 function failed.

8650 **Canceling Functions that Need Data**

8651 After *SQLExecute*() or *SQLExecDirect*() returns SQL_NEED_DATA and before data has been sent
8652 for all data-at-execution parameters, an application can call *SQLCancel*() to cancel the statement
8653 execution. After the statement has been canceled, the application can call *SQLExecute*() or
8654 *SQLExecDirect*() again.

8655 After *SQLBulkOperations*() or *SQLSetPos*() returns SQL_NEED_DATA and before data has been
8656 sent for all data-at-execution columns, an application can call *SQLCancel*() to cancel the
8657 operation. After the operation has been canceled, the application can call *SQLBulkOperations*()
8658 or *SQLSetPos*() again; canceling does not affect the cursor state or the current cursor position.

8659 **Canceling Functions in Multithreaded Applications**

8660 If neither asynchronous execution nor the data-at-execution dialogue is active on
8661 *StatementHandle*, a multithread application can call *SQLCancel*() from one thread to try to cancel
8662 execution of an SQL statement by another thread that is using the same connection. The
8663 application passes *SQLCancel*() the statement handle used by the target function in the other
8664 thread. The return code of *SQLCancel*() indicates only whether the implementation processed
8665 the request successfully. Only SQL_SUCCESS or SQL_ERROR can be returned; no SQLSTATEs
8666 are returned. The return code of the original function indicates whether it completed normally or
8667 was canceled.

8668 This document does not specify whether or how an application could get control of the
8669 processor during SQL statement execution in order to call *SQLCancel*().

8670 If asynchronous execution or the data-at-execution dialogue is active, a call to *SQLCancel*()
8671 affects these features, as described above, in precedence to canceling an operation in a different
8672 thread. When calling *SQLCancel*() to try to cancel an operation in a different process or thread:

8673 • If an XDBC function is executing asynchronously on *StatementHandle*, the attempt to cancel
8674 execution may interfere with the other process' or thread's activity in polling the completion
8675 of, or cancelling, its own operation.

8676 • If the data-on-execute dialogue is in progress on *StatementHandle*, the attempt to cancel
8677 execution could affect the progress of that dialogue.

8678 This could result in the function in the other thread failing with an SQLSTATE of HY010
8679 (Function sequence error).

8680 **SEE ALSO**

8681 For information about	See
8682 Binding a buffer to a parameter	*SQLBindParameter*()
8683 Performing bulk insert or update operations	*SQLBulkOperations*()
8684 Executing an SQL statement	*SQLExecDirect*()
8685 Executing a prepared SQL statement	*SQLExecute*()
8686 Freeing a statement handle	*SQLFreeStmt*()
8687 Positioning the cursor in a row-set, refreshing data in the 8688 row-set, or updating or deleting data in the row-set	*SQLSetPos*()
8689 Returning the next parameter to send data for	*SQLParamData*()
8690 Sending parameter data at execution time	*SQLPutData*()

8691 **CHANGE HISTORY**

8692 **Version 2**

8693 Revised generally. See **Alignment with Popular Implementations** on page 2.

8694 **NAME**

8695 SQLCloseCursor — Close a cursor that has been opened on a statement, discarding pending
8696 results.

8697 **SYNOPSIS**

```
8698 SQLRETURN SQLCloseCursor(
8699   SQLHSTMT StatementHandle);
```

8700 **ARGUMENTS**

8701 *StatementHandle* [Input]
8702 Statement handle

8703 **RETURN VALUE**

8704 SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.

8705 **DIAGNOSTICS**

8706 When *SQLCloseCursor*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated
8707 SQLSTATE value can be obtained by calling *SQLGetDiagRec*() with a *HandleType* of
8708 SQL_HANDLE_STMT and a *Handle* of *StatementHandle*. The following SQLSTATE values are
8709 commonly returned by *SQLCloseCursor*(). The return code associated with each SQLSTATE
8710 value is SQL_ERROR, except that for SQLSTATE values in class 01, the return code is
8711 SQL_SUCCESS_WITH_INFO.

8712 01000 — General warning
8713 Implementation-defined informational message.

8714 24000 — Invalid cursor state
8715 No cursor was open on the *StatementHandle*.

8716 HY000 — General error
8717 An error occurred for which there was no specific SQLSTATE and for which no
8718 implementation-specific SQLSTATE was defined. The error message returned by
8719 *SQLGetDiagRec*() in the **MessageText* buffer describes the error and its cause.

8720 HY001 — Memory allocation error
8721 The implementation failed to allocate memory required to support execution or completion
8722 of the function.

8723 HY010 — Function sequence error
8724 An asynchronously executing function was called for *StatementHandle* and was still
8725 executing when this function was called.

8726 *SQLBulkOperations*(), *SQLExecDirect*(), *SQLExecute*(), or *SQLSetPos*() was called for
8727 *StatementHandle* and returned SQL_NEED_DATA. This function was called before data was
8728 sent for all data-at-execution parameters or columns.

8729 HYT01 — Connection timeout expired
8730 The connection timeout period expired before the data source responded to the request. The
8731 connection timeout period is set through *SQLSetConnectAttr*(),
8732 SQL_ATTR_CONNECTION_TIMEOUT.

8733 IM001 — Function not supported
8734 The function is not supported on the current connection to the data source.

8735 **COMMENTS**

8736 After an application calls *SQLCloseCursor*(), the application can reopen the cursor later by
8737 executing a SELECT statement again with the same or different parameter values.

8738 *SQLCloseCursor*() returns SQLSTATE 24000 (Invalid cursor state) if no cursor is open. Calling
8739 *SQLCloseCursor*() is equivalent to calling *SQLFreeStmt*() with the SQL_CLOSE option, with the

8740 exception that *SQLFreeStmt*() with SQL_CLOSE has no effect on the application if no cursor is
8741 open on the statement, while *SQLCloseCursor*() returns SQLSTATE 24000 (Invalid cursor state).

8742 **SEE ALSO**

8743 For information about	See
8744 Canceling statement processing	*SQLCancel*()
8745 Freeing a handle	*SQLFreeHandle*()
8746 Process multiple result sets	*SQLMoreResults*()

8747 **CHANGE HISTORY**

8748 **Version 2**

8749 Revised generally. See **Alignment with Popular Implementations** on page 2.

8750 **NAME**

8751 SQLColAttribute — Return descriptor information for a column in a result set.

8752 **SYNOPSIS**

```
8753 SQLRETURN SQLColAttribute (
8754   SQLHSTMT StatementHandle,
8755   SQLUSMALLINT ColumnNumber,
8756   SQLUSMALLINT FieldIdentifier,
8757   SQLPOINTER CharacterAttributePtr,
8758   SQLSMALLINT BufferLength,
8759   SQLSMALLINT * StringLengthPtr,
8760   SQLPOINTER NumericAttributePtr);
```

8761 **ARGUMENTS**

8762 *StatementHandle* [Input]
8763 Statement handle.

8764 *ColumnNumber* [Input]
8765 The number of the record in the IRD from which the field value is to be retrieved. This
8766 argument corresponds to the column number of result data, ordered sequentially from left
8767 to right, starting at 1. Columns may be described in any order.

8768 Column 0 can be specified in this argument, but all values except SQL_DESC_TYPE and
8769 SQL_DESC_OCTET_LENGTH return undefined values.

8770 *FieldIdentifier* [Input]
8771 The field in row *ColumnNumber* of the IRD that is to be returned (see ‘‘Comments’’).

8772 *CharacterAttributePtr* [Output]
8773 Pointer to a buffer in which to return the value in the FieldIdentifier field of the
8774 ColumnNumber row of the IRD, if the field is a character string. Otherwise, the field is
8775 unused.

8776 *BufferLength* [Input]
8777 The length of the **CharacterAttributePtr* buffer, if the field is a character string. Otherwise,
8778 this field is ignored.

8779 *StringLengthPtr* [Output]
8780 Pointer to a buffer in which to return the total number of octets (excluding the null
8781 terminator for character data) available to return in **CharacterAttributePtr*.

8782 • For character data, if the number of octets available to return is greater than or equal to
8783 *BufferLength*, the descriptor information in **CharacterAttributePtr* is truncated to
8784 *BufferLength* minus the length of a null terminator and is null-terminated.

8785 • For all other types of data, the value of *BufferLength* is ignored and the implementation
8786 assumes the size of **CharacterAttributePtr* is 32 bits.

8787 *NumericAttributePtr* [Output]
8788 Pointer to an integer buffer in which to return the value in the *FieldIdentifier* field of the
8789 *ColumnNumber* row of the IRD, if the field is numeric, such as
8790 SQL_DESC_COLUMN_LENGTH. Otherwise, the field is unused.

8791 **RETURN VALUE**

8792 SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or
8793 SQL_INVALID_HANDLE.

8794 **DIAGNOSTICS**

8795 When *SQLColAttribute*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated

8796 SQLSTATE value can be obtained by calling *SQLGetDiagRec*() with a *HandleType* of
8797 SQL_HANDLE_STMT and an *Handle* of *StatementHandle*. The following SQLSTATE values are
8798 commonly returned by *SQLColAttribute*(). The return code associated with each SQLSTATE
8799 value is SQL_ERROR, except that for SQLSTATE values in class 01, the return code is
8800 SQL_SUCCESS_WITH_INFO.

8801 01000 — General warning
8802 Implementation-defined informational message.

8803 01004 — String data, right truncation
8804 The buffer **CharacterAttributePtr* was not large enough to return the entire string value, so
8805 the string value was truncated. The length of the untruncated string value is returned in
8806 **StringLengthPtr*.

8807 07005 — Prepared statement not a cursor-specification
8808 The statement associated with the *StatementHandle* did not return a result set. There were
8809 no columns to describe.

8810 07009 — Invalid descriptor index
8811 *ColumnNumber* was 0 and the SQL_ATTR_USE_BOOKMARKS statement attribute was
8812 SQL_UB_OFF.

8813 *ColumnNumber* was less than 0.

8814 *ColumnNumber* was greater than the number of columns in the result set.

8815 HY000 — General error
8816 An error occurred for which there was no specific SQLSTATE and for which no
8817 implementation-specific SQLSTATE was defined. The error message returned by
8818 *SQLGetDiagField*() from the diagnostic data structure describes the error and its cause.

8819 HY001 — Memory allocation error
8820 The implementation failed to allocate memory required to support execution or completion
8821 of the function.

8822 HY008 — Operation canceled
8823 Asynchronous processing was enabled for *StatementHandle*. The function was called and
8824 before it completed execution, *SQLCancel*() was called on *StatementHandle*. The function
8825 was then called again on *StatementHandle*.

8826 The function was called and, before it completed execution, *SQLCancel*() was called on
8827 *StatementHandle* from a different thread in a multithread application.

8828 HY010 — Function sequence error
8829 The function was called prior to calling *SQLPrepare*() or *SQLExecDirect*() for
8830 *StatementHandle*.

8831 An asynchronously executing function (not this one) was called for *StatementHandle* and
8832 was still executing when this function was called.

8833 *SQLBulkOperations*(), *SQLExecDirect*(), *SQLExecute*(), or *SQLSetPos*() was called for
8834 *StatementHandle* and returned SQL_NEED_DATA. This function was called before data was
8835 sent for all data-at-execution parameters or columns.

8836 HY090 — Invalid string or buffer length
8837 *BufferLength* was less than 0.

8838 HY091 — Invalid descriptor field identifier
8839 *FieldIdentifier* was not one of the defined values or an implementation-defined value.

8840 HYC00 — Optional feature not implemented
8841 The data source does not support the value of *FieldIdentifier*.

8842 HYT01 — Connection timeout expired
8843 The connection timeout period expired before the data source responded to the request. The
8844 connection timeout period is set through *SQLSetConnectAttr*(),
8845 SQL_ATTR_CONNECTION_TIMEOUT.

8846 IM001 — Function not supported
8847 The function is not supported on the current connection to the data source.

8848 When the application calls *SQLColAttribute*() after *SQLPrepare*() and before *SQLExecute*(), it can
8849 return any SQLSTATE that can be returned by *SQLPrepare*() or *SQLExecute*(), depending on
8850 when the data source evaluates the SQL statement associated with *StatementHandle* (see
8851 **Performance Note** on page 279).

8852 **COMMENTS**

8853 A call to *SQLColAttribute*() returns one piece of descriptive information about a column of the
8854 result set returned on *StatementHandle*. The information is returned in one of two ways,
8855 indicated by (N) or (C) in the list of valid values of *FieldIdentifier:*

8856 (N) The information is either an integer value or a 32-bit implementation-defined value.
8857 *SQLColAttribute*() returns the information in **NumericAttributePtr* and does not use or
8858 modify *CharacterAttributePtr* or *StringLengthPtr*.

8859 (C) The information is a character string. *SQLColAttribute*() returns the information in
8860 **CharacterAttributePtr* and sets *StringLengthPtr*. It does not use or modify
8861 *NumericAttributePtr*.

8862 Valid values of *FieldIdentifier* include at least all the descriptor fields that are defined for the IRD.
8863 (This information can also be retrieved by calling *SQLGetDescField*() and supplying the same
8864 values as *FieldIdentifier*.) Additional descriptor fields are likely to be defined to take advantage
8865 of different data sources.

8866 The following are the valid values of *FieldIdentifier*. (Header) denotes header fields; for these,
8867 *SQLColAttribute*() ignores *ColumnNumber*. Header fields are defined in **Fields of the Descriptor**
8868 **Header** on page 472; record fields are defined in **Fields of Each Descriptor Record** on page 476.

8869 SQL_DESC_AUTO_UNIQUE_VALUE (N) SQL_DESC_LOCAL_TYPE_NAME (C)
8870 SQL_DESC_BASE_COLUMN_NAME (C) SQL_DESC_NAME (C)
8871 SQL_DESC_BASE_TABLE_NAME (C) SQL_DESC_NULLABLE (N)
8872 SQL_DESC_CASE_SENSITIVE (N) SQL_DESC_OCTET_LENGTH (N)
8873 SQL_DESC_CATALOG_NAME (C) SQL_DESC_PARAMETER_TYPE (N)
8874 SQL_DESC_CONCISE_TYPE (C) SQL_DESC_PRECISION (N)
8875 SQL_DESC_COUNT (N) (Header) SQL_DESC_SCALE (N)
8876 SQL_DESC_DATETIME_INTERVAL_CODE (N) SQL_DESC_SCHEMA_NAME (C)
8877 SQL_DESC_DATETIME_INTERVAL_PRECISION (N) SQL_DESC_SEARCHABLE (N)
8878 SQL_DESC_DISPLAY_SIZE (N) SQL_DESC_TABLE_NAME (C)
8879 SQL_DESC_FIXED_PREC_SCALE (N) SQL_DESC_TYPE (N)
8880 SQL_DESC_LABEL (C) SQL_DESC_TYPE_NAME (C)
8881 SQL_DESC_LENGTH (N) SQL_DESC_UNNAMED (N)
8882 SQL_DESC_LITERAL_PREFIX (C) SQL_DESC_UNSIGNED (N)
8883 SQL_DESC_LITERAL_SUFFIX (C) SQL_DESC_UPDATABLE (N)

8884 This function is an extensible alternative to *SQLDescribeCol*(). *SQLDescribeCol*() returns a fixed
8885 set of descriptor information. *SQLColAttribute*() allows access to the more extensive set of
8886 descriptor information available in the ISO SQL standard and accommodates future
8887 enhancements and vendor extensions.

8888 Calling *SQLColAttribute*() between the preparation and the execution of an SQL statement has
8889 performance implications; see **Performance Note** on page 279.

8890 **SEE ALSO**

8891	**For information about**	**See**
8892	Binding a buffer to a column in a result set	*SQLBindCol*()
8893	Canceling statement processing	*SQLCancel*()
8894	Returning information about a column in a result set	*SQLDescribeCol*()
8895	Fetching a block of data or scrolling through a result set	*SQLFetchScroll*()
8896	Fetching multiple rows of data	*SQLFetch*()
8897	Definition of all descriptor fields	*SQLSetDescField*()

8898 **CHANGE HISTORY**

8899 **Version 2**

8900 Revised generally. See **Alignment with Popular Implementations** on page 2.

8901 SQLColumnPrivileges — Return a list of columns and associated privileges for the specified table as a
8902 result set.

8903 **SYNOPSIS**

```
8904 SQLRETURN SQLColumnPrivileges(
8905   SQLHSTMT StatementHandle,
8906   SQLCHAR * CatalogName,
8907   SQLSMALLINT NameLength1,
8908   SQLCHAR * SchemaName,
8909   SQLSMALLINT NameLength2,
8910   SQLCHAR * TableName,
8911   SQLSMALLINT NameLength3,
8912   SQLCHAR * ColumnName,
8913   SQLSMALLINT NameLength4);
```

8914 **ARGUMENTS**

8915 *StatementHandle* [Input]
8916 Statement handle.

8917 *CatalogName* [Input]
8918 Catalog name. If a data source supports names for catalogs, an empty string denotes those
8919 catalogs that do not have names.

8920 If the SQL_ATTR_METADATA_ID statement attribute is SQL_TRUE, this argument is
8921 interpreted as specified in **Identifier (ID) Arguments** on page 72. If it is SQL_FALSE, this
8922 argument is interpreted as specified in **Ordinary Arguments (OA)** on page 71.

8923 *NameLength1* [Input]
8924 Length of **CatalogName*.

8925 *SchemaName* [Input]
8926 Schema name. If a data source supports schemas, an empty string denotes those tables that
8927 do not have schemas.

8928 If the SQL_ATTR_METADATA_ID statement attribute is SQL_TRUE, this argument is
8929 interpreted as specified in **Identifier (ID) Arguments** on page 72. If it is SQL_FALSE, this
8930 argument is interpreted as specified in **Ordinary Arguments (OA)** on page 71.

8931 *NameLength2* [Input]
8932 Length of **SchemaName*.

8933 *TableName* [Input]
8934 Table name. This argument cannot be a null pointer.

8935 If the SQL_ATTR_METADATA_ID statement attribute is SQL_TRUE, this argument is
8936 interpreted as specified in **Identifier (ID) Arguments** on page 72. If it is SQL_FALSE, this
8937 argument is interpreted as specified in **Ordinary Arguments (OA)** on page 71.

8938 *NameLength3* [Input]
8939 Length of **TableName*.

8940 *ColumnName* [Input]
8941 String search pattern for column names.

8942 If the SQL_ATTR_METADATA_ID statement attribute is SQL_TRUE, this argument is
8943 interpreted as specified in **Identifier (ID) Arguments** on page 72. If it is SQL_FALSE, this
8944 argument is interpreted as specified in **Pattern Value (PV) Arguments** on page 71 and the
8945 application may use a search pattern.

NameLength4 [Input]
Length of **ColumnName*.

RETURN VALUE

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE.

DIAGNOSTICS

When *SQLColumnPrivileges*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling *SQLGetDiagRec*() with a *HandleType* of SQL_HANDLE_STMT and a *Handle* of *StatementHandle*. The following SQLSTATE values are commonly returned by *SQLColumnPrivileges*(). The return code associated with each SQLSTATE value is SQL_ERROR, except that for SQLSTATE values in class 01, the return code is SQL_SUCCESS_WITH_INFO.

01000 — General warning
Implementation-defined informational message.

08S01 — Communication link failure
The communication link to the data source failed before the function completed processing.

24000 — Invalid cursor state
A cursor was open on *StatementHandle*.

HY000 — General error
An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by *SQLGetDiagRec*() in the **MessageText* buffer describes the error and its cause.

HY001 — Memory allocation error
The implementation failed to allocate memory required to support execution or completion of the function.

HY008 — Operation canceled
Asynchronous processing was enabled for *StatementHandle*. The function was called and before it completed execution, *SQLCancel*() was called on *StatementHandle*. The function was then called again on *StatementHandle*.

The function was called and, before it completed execution, *SQLCancel* was called on *StatementHandle* from a different thread in a multithread application.

HY009 — Invalid use of null pointer
TableName was a null pointer.

The SQL_ATTR_METADATA_ID statement attribute was set to SQL_TRUE, *CatalogName* was a null pointer, and the SQL_CATALOG_NAME option of *SQLGetInfo*() returns that catalog names are supported.

The SQL_ATTR_METADATA_ID statement attribute was set to SQL_TRUE, and *SchemaName*, *TableName*, or *ColumnName* was a null pointer.

The SQL_ATTR_METADATA_ID statement attribute was set to SQL_FALSE, and *TableName* was a null pointer.

HY010 — Function sequence error
An asynchronously executing function (not this one) was called for *StatementHandle* and was still executing when this function was called.

SQLBulkOperations(), *SQLExecDirect*(), *SQLExecute*(), or *SQLSetPos*() was called for *StatementHandle* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.

8992 HY090 — Invalid string or buffer length
8993 The value of one of the name length arguments was less than 0, but not equal to SQL_NTS.

8994 The value of one of the name length arguments exceeded the maximum length value for the
8995 corresponding name (see ‘‘Comments’’).

8996 HYC00 — Optional feature not implemented
8997 A catalog name was specified and the implementation does not support catalogs.

8998 A schema name was specified and the implementation does not support schemas.

8999 A string search pattern was specified for the column name and the data source does not
9000 support search patterns for that argument.

9001 The data source does not support the combination of the current settings of the
9002 SQL_ATTR_CONCURRENCY and SQL_ATTR_CURSOR_TYPE statement attributes.

9003 The SQL_ATTR_USE_BOOKMARKS statement attribute was set to SQL_UB_VARIABLE,
9004 and the SQL_ATTR_CURSOR_TYPE statement attribute was set to a cursor type for which
9005 the data source does not support bookmarks.

9006 HYT00 — Timeout expired
9007 The query timeout period expired before the data source returned the result set. The
9008 timeout period is set through *SQLSetStmtAttr*(), SQL_ATTR_QUERY_TIMEOUT.

9009 HYT01 — Connection timeout expired
9010 The connection timeout period expired before the data source responded to the request. The
9011 connection timeout period is set through *SQLSetConnectAttr*(),
9012 SQL_ATTR_CONNECTION_TIMEOUT.

9013 IM001 — Function not supported
9014 The function is not supported on the current connection to the data source.

9015 **COMMENTS**

9016 *SQLColumnPrivileges*() returns the results as a standard result set, ordered by TABLE_CAT,
9017 TABLE_SCHEM, TABLE_NAME, COLUMN_NAME, and PRIVILEGE.

9018 **Note:** *SQLColumnPrivileges*() might not return privileges for all columns. For example, a data
9019 source might not return information about privileges for pseudo-columns. Applications can use
9020 any valid column, regardless of whether it is returned by *SQLColumnPrivileges*().

9021 The lengths of VARCHAR columns shown in the table are maximums; the actual lengths depend
9022 on the data source. To determine the actual lengths of the CATALOG_NAME,
9023 SCHEMA_NAME, TABLE_NAME, and COLUMN_NAME columns, an application can call
9024 *SQLGetInfo*() with the SQL_MAX_CATALOG_NAME_LEN,
9025 SQL_MAX_SCHEMA_NAME_LEN, SQL_MAX_TABLE_NAME_LEN, and
9026 SQL_MAX_COLUMN_NAME_LEN options.

9027 The following table lists the columns in the result set. Additional columns beyond column 8
9028 (IS_GRANTABLE) can be defined by the implementation. An application should gain access to
9029 implementation-defined columns by counting down from the end of the result set rather than by
9030 specifying an explicit ordinal position; see Section 7.3 on page 68.

9031–9032

Column Name	Col. No.	Data Type	Comments
TABLE_CAT	1	Varchar	Catalog identifier; NULL if not applicable to the data source. If a data source supports catalogs, it returns an empty string for those tables that do not have catalogs.

9033–9035

TABLE_SCHEM	2	Varchar	Schema identifier; NULL if not applicable to the data source. If a data source supports schemas, it returns an empty string for those tables that do not have schemas.
TABLE_NAME	3	Varchar not NULL	Table identifier.
COLUMN_NAME	4	Varchar not NULL	Column identifier; an empty string for unnamed columns.
GRANTOR	5	Varchar	Identifier of the user who granted the privilege; NULL if not applicable to the data source. For all rows in which the value in the GRANTEE column is the owner of the object, the GRANTOR column is ‘‘_SYSTEM’’.
GRANTEE	6	Varchar not NULL	Identifier of the user to whom the privilege was granted.
PRIVILEGE	7	Varchar not NULL	Identifies the column privilege. May be one of the following or others supported by the data source when implementation-defined: SELECT: The grantee is permitted to retrieve data for the column. INSERT: The grantee is permitted to provide data for the column in new rows that are inserted into the associated table. UPDATE: The grantee is permitted to update data in the column. REFERENCES: The grantee is permitted to refer to the column within a constraint (for example, a unique, referential, or table check constraint).
IS_GRANTABLE	8	Varchar	Indicates whether the grantee is permitted to grant the privilege to other users; ‘‘YES’’, ‘‘NO’’, or NULL if unknown or not applicable to the data source.A privilege is either grantable or not grantable, but not both. The result set returned by *SQLColumnPrivileges*() will never contain two rows for which all columns except the IS_GRANTABLE column contain the same value.

SEE ALSO

For information about	See
Overview of catalog functions	Chapter 7
Binding a buffer to a column in a result set	*SQLBindCol*()
Canceling statement processing	*SQLCancel*()

Returning the columns in a table or tables	*SQLColumns*()
Fetching a block of data or scrolling through a result set	*SQLFetchScroll*()
Fetching multiple rows of data	*SQLFetch*()
Returning privileges for a table or tables	*SQLTablePrivileges*()
Returning a list of tables in a data source	*SQLTables*()

CHANGE HISTORY

Version 2

Function added in this version.

NAME

SQLColumns — Return the list of column names in specified tables as a result set.

SYNOPSIS

```
SQLRETURN SQLColumns(
  SQLHSTMT StatementHandle,
  SQLCHAR * CatalogName,
  SQLSMALLINT NameLength1,
  SQLCHAR * SchemaName,
  SQLSMALLINT NameLength2,
  SQLCHAR * TableName,
  SQLSMALLINT NameLength3,
  SQLCHAR * ColumnName,
  SQLSMALLINT NameLength4);
```

ARGUMENTS

StatementHandle [Input]
Statement handle.

CatalogName [Input]
Catalog name. If a data source supports catalogs, an empty string denotes those tables that do not have catalogs.

If the SQL_ATTR_METADATA_ID statement attribute is SQL_TRUE, this argument is interpreted as specified in **Identifier (ID) Arguments** on page 72. If it is SQL_FALSE, this argument is interpreted as specified in **Ordinary Arguments (OA)** on page 71.

NameLength1 [Input]
Length of **CatalogName*.

SchemaName [Input]
String search pattern for schema names. If a data source supports schemas, an empty string denotes those tables that do not have schemas.

If the SQL_ATTR_METADATA_ID statement attribute is SQL_TRUE, this argument is interpreted as specified in **Identifier (ID) Arguments** on page 72. If it is SQL_FALSE, this argument is interpreted as specified in **Pattern Value (PV) Arguments** on page 71 and the application may use a search pattern.

NameLength2 [Input]
Length of **SchemaName*.

TableName [Input]
String search pattern for table names.

If the SQL_ATTR_METADATA_ID statement attribute is SQL_TRUE, this argument is interpreted as specified in **Identifier (ID) Arguments** on page 72. If it is SQL_FALSE, this argument is interpreted as specified in **Pattern Value (PV) Arguments** on page 71 and the application may use a search pattern.

NameLength3 [Input]
Length of **TableName*.

ColumnName [Input]
String search pattern for column names.

If the SQL_ATTR_METADATA_ID statement attribute is SQL_TRUE, this argument is interpreted as specified in **Identifier (ID) Arguments** on page 72. If it is SQL_FALSE, this argument is interpreted as specified in **Pattern Value (PV) Arguments** on page 71 and the

9130 application may use a search pattern.

9131 *NameLength4* [Input]
9132 Length of **ColumnName*.

9133 **RETURN VALUE**
9134 SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or
9135 SQL_INVALID_HANDLE.

9136 **DIAGNOSTICS**
9137 When *SQLColumns*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated
9138 SQLSTATE value can be obtained by calling *SQLGetDiagRec*() with a *HandleType* of
9139 SQL_HANDLE_STMT and a *Handle* of *StatementHandle*. The following SQLSTATE values are
9140 commonly returned by *SQLColumns*(). The return code associated with each SQLSTATEvalue is
9141 SQL_ERROR, except that for SQLSTATE values in class 01, the return code is
9142 SQL_SUCCESS_WITH_INFO.

9143 01000 — General warning
9144 Implementation-defined informational message.

9145 08S01 — Communication link failure
9146 The communication link to the data source failed before the function completed processing.

9147 24000 — Invalid cursor state
9148 A cursor was open on *StatementHandle*.

9149 HY000 — General error
9150 An error occurred for which there was no specific SQLSTATE and for which no
9151 implementation-specific SQLSTATE was defined. The error message returned by
9152 *SQLGetDiagRec*() in the **MessageText*buffer describes the error and its cause.

9153 HY001 — Memory allocation error
9154 The implementation failed to allocate memory required to support execution or completion
9155 of the function.

9156 HY008 — Operation canceled
9157 Asynchronous processing was enabled for *StatementHandle*. The function was called and
9158 before it completed execution, *SQLCancel*() was called on *StatementHandle*. The function
9159 was then called again on *StatementHandle*.

9160 The function was called and, before it completed execution, *SQLCancel*() was called on
9161 *StatementHandle* from a different thread in a multithread application.

9162 HY009 — Invalid use of null pointer
9163 The SQL_ATTR_METADATA_IDstatement attribute was set to SQL_TRUE, *CatalogName*
9164 was a null pointer, and the SQL_CATALOG_NAME option of *SQLGetInfo*() returns that
9165 catalog names are supported.

9166 The SQL_ATTR_METADATA_ID statement attribute was set to SQL_TRUE, and
9167 *SchemaName*, *TableName*, or *ColumnName* was a null pointer.

9168 HY010 — Function sequence error
9169 An asynchronously executing function (not this one) was called for *StatementHandle* and
9170 was still executing when this function was called.

9171 *SQLBulkOperations*(), *SQLExecDirect*(), *SQLExecute*(), or *SQLSetPos*() was called for
9172 *StatementHandle* and returned SQL_NEED_DATA.This function was called before data was
9173 sent for all data-at-execution parameters or columns.

9174 HY090 — Invalid string or buffer length
9175 The value of one of the name length arguments was less than 0, but not equal to SQL_NTS.

9176 The value of one of the name length arguments exceeded the maximum length value for the
9177 corresponding catalog or name. The maximum length of each catalog or name may be
9178 obtained by calling *SQLGetInfo*() with the SQL_MAX_CATALOG_NAME_LEN or
9179 SQL_MAX_SCHEMA_NAME_LEN options (see "Comments").

9180 HYC00 — Optional feature not implemented
9181 A catalog name was specified and the implementation does not support catalogs.

9182 A schema name was specified and the implementation does not support schemas.

9183 A string search pattern was specified for the schema name, table name, or column name and
9184 the data source does not support search patterns for one or more of those arguments.

9185 The data source does not support the combination of the current settings of the
9186 SQL_ATTR_CONCURRENCY and SQL_ATTR_CURSOR_TYPE statement attributes.

9187 The SQL_ATTR_USE_BOOKMARKS statement attribute was set to SQL_UB_VARIABLE,
9188 and the SQL_ATTR_CURSOR_TYPE statement attribute was set to a cursor type for which
9189 the data source does not support bookmarks.

9190 HYT00 — Timeout expired
9191 The query timeout period expired before the data source returned the result set. The
9192 timeout period is set through *SQLSetStmtAttr*(), SQL_ATTR_QUERY_TIMEOUT.

9193 HYT01 — Connection timeout expired
9194 The connection timeout period expired before the data source responded to the request. The
9195 connection timeout period is set through *SQLSetConnectAttr*(),
9196 SQL_ATTR_CONNECTION_TIMEOUT.

9197 IM001 — Function not supported
9198 The function is not supported on the current connection to the data source.

9199 **COMMENTS**

9200 This function is typically used before statement execution to retrieve information about columns
9201 for a table or tables from the data source's catalog. *SQLColumns*() can be used to retrieve data
9202 for all types of items returned by *SQLTables*(), including base tables, views, synonyms, and
9203 system tables. By contrast, *SQLColAttribute*() and *SQLDescribeCol*() describe the columns in a
9204 result set and *SQLNumResultCols*() returns the number of columns in a result set.

9205 *SQLColumns*() returns the results as a standard result set, ordered by TABLE_CAT,
9206 TABLE_SCHEM, TABLE_NAME, and ORDINAL_POSITION. The order of the columns in the
9207 column list returned by *SQLColumns*() is not necessarily the same as the order of the columns
9208 returned when the application performs a SELECT statement on all columns in that table.

9209 **Note:** *SQLColumns*() might not return all columns. For example, a data source might not return
9210 information about pseudo-columns. Applications can use any valid column, regardless of
9211 whether it is returned by *SQLColumns*().

9212 Some columns that can be returned by *SQLStatistics*() are not returned by *SQLColumns*(). For
9213 example, *SQLColumns*() does not return the columns in an index created over an expression or
9214 filter, such as SALARY + BENEFITS or DEPT = 0012.

9215 The lengths of VARCHAR columns shown in the table are maximums; the actual lengths depend
9216 on the data source. To determine the actual lengths of the TABLE_CAT, TABLE_SCHEM,
9217 TABLE_NAME, and COLUMN_NAME columns, an application can call *SQLGetInfo*() with the
9218 SQL_MAX_CATALOG_NAME_LEN, SQL_MAX_SCHEMA_NAME_LEN,
9219 SQL_MAX_TABLE_NAME_LEN, and SQL_MAX_COLUMN_NAME_LEN options.

9220 The following table lists the columns in the result set. Additional columns beyond column 18
9221 (IS_NULLABLE) can be defined by the implementation. An application should gain access to
9222 implementation-defined columns by counting down from the end of the result set rather than by
9223 specifying an explicit ordinal position; see Section 7.3 on page 68.

Column Name	Col. No.	Data Type	Comments
TABLE_CAT	1	Varchar	Catalog identifier; NULL if not applicable to the data source. If a data source supports catalogs, it returns an empty string for those tables that do not have catalogs.
TABLE_SCHEM	2	Varchar	Schema identifier; NULL if not applicable to the data source. If a data source supports schemas, it returns an empty string for those tables that do not have schemas.
TABLE_NAME	3	Varchar not NULL	Table identifier.
COLUMN_NAME	4	Varchar not NULL	Column identifier; an empty string for unnamed columns.
DATA_TYPE	5	Smallint not NULL	SQL data type. This can be an XDBC SQL data type or an implementation-defined SQL data type. For date/time and interval data types, this column returns the concise data type (for example, SQL_TYPE_DATE or SQL_INTERVAL_YEAR_TO_MONTH) rather than the non-concise data type (SQL_DATETIME or SQL_INTERVAL). For a list of valid XDBC SQL data types, see Section D.1 on page 556.
TYPE_NAME	6	Varchar not NULL	Data source-dependent data type name; for example, ''CHAR'', ''VARCHAR'', ''MONEY'', ''LONG VARBINARY'', or ''CHAR ( ) FOR BIT DATA''.

COLUMN_SIZE	7	Integer	If DATA_TYPE is SQL_CHAR or SQL_VARCHAR, then this column contains the maximum length in characters of the column. For date/time data types, this is the total number of characters required to display the value when converted to characters. For numeric data types, this is either the total number of digits or the total number of bits allowed in the column, according to the NUM_PREC_RADIX column. For interval data types, this is the number of characters in the character representation of the interval literal (as defined by the interval leading precision, see **Interval Data Type Length** on page 571). For more information, see Section D.3 on page 562.
BUFFER_LENGTH	8	Integer	The length in octets of data transferred on an *SQLGetData*(), *SQLFetch*(), or *SQLFetchScroll*() operation if SQL_C_DEFAULT is specified. For numeric data, this size may be different from the size of the data stored on the data source. This value is the same as the COLUMN_SIZE column for character or binary data. For more information about length, see Section D.3 on page 562.
DECIMAL_DIGITS	9	Smallint	The total number of significant digits to the right of the decimal point. For SQL_TYPE_TIME and SQL_TYPE_TIMESTAMP, this column contains the number of digits in the fractional seconds component. For the other data types this is the scale of the column on the data source. For interval data types that contain a time component, this column contains the number of digits to the right of the decimal point (that is, fractional seconds). For interval data types that do not contain a time component, this column is 0. For more information on decimal digits, see Section D.3 on page 562. NULL is returned for data types where scale is not applicable.

NUM_PREC_RADIX	10	Smallint	For numeric data types, either 10 or 2. If it is 10, the values in COLUMN_SIZE and DECIMAL_DIGITS give the number of decimal digits allowed for the column. For example, a DECIMAL(12,5) column returns a NUM_PREC_RADIX of 10, a COLUMN_SIZE of 12, and a DECIMAL_DIGITS of 5; A FLOAT column could return a NUM_PREC_RADIX of 10, a COLUMN_SIZE of 15 and a DECIMAL_DIGITS of NULL. If it is 2, the values in COLUMN_SIZE and DECIMAL_DIGITS give the number of bits allowed in the column. For example, a FLOAT column could return a RADIX of 2, a COLUMN_SIZE of 53, and a DECIMAL_DIGITS of NULL. NULL is returned for data types where NUM_PREC_RADIX is not applicable.
NULLABLE	11	Smallint not NULL	SQL_NO_NULLS if the column does not accept NULL values. SQL_NULLABLE if the column accepts NULL values. SQL_NULLABLE_UNKNOWN if it is not known if the column accepts NULL values. The value returned for this column is different from the value returned for the IS_NULLABLE column. The NULLABLE column indicates with certainty that a column can accept NULLs, but cannot indicate with certainty that a column does not accept NULLs. The IS NULLABLE column indicates with certainty that a column cannot accept NULLs, but cannot indicate with certainty that a column accepts NULLs.

REMARKS	12	Varchar	A description of the column.
COLUMN_DEF	13	Varchar	The default value of the column. See Section 7.3.1 on page 68.
SQL_DATA_TYPE	14	Smallint not NULL	SQL data type, as it appears in the SQL_DESC_TYPE record field in the IRD. This can be an XDBC SQL data type or an implementation-defined SQL data type. This column is the same as the DATA_TYPEcolumn, except that for date/time and interval data types, this column returns the non-concise data type (SQL_DATETIME or SQL_INTERVAL) rather than the concise data type (for example, SQL_TYPE_DATE or SQL_INTERVAL_YEAR_TO_MONTH); in this case, the specific data type can be determined from the SQL_DATETIME_SUB column. For a list of valid XDBC SQL data types, see Section D.1 on page 556.
SQL_DATETIME_SUB	15	Smallint	The subtype code for date/time and interval data types. For other data types, this column returns a NULL. For more information about date/time and interval subcodes, see "SQL_DESC_DATETIME_INTERVAL_CODE" in *SQLSetDescField*().
CHAR_OCTET_LENGTH	16	Integer	The maximum length in octets of a character or binary data type column. For all other data types, this column returns a NULL.
ORDINAL_POSITION	17	Integer not NULL	The ordinal position of the column in the table. The first column in the table is number 1.

9376 IS_NULLABLE 18 Varchar ‘‘NO’’ if the column does not include
9377 NULLs. ‘‘YES’’ if the column could
9378 include NULLS.

9379 This column returns a zero-length string
9380 if nullability is unknown. ISO rules are
9381 followed to determine nullability. An
9382 ISO SQL compliant data source cannot
9383 return an empty string.

9384 The value returned for this column is
9385 different from the value returned for the
9386 NULLABLE column. (See the
9387 description of the NULLABLE column.)

9388 **SEE ALSO**

9389 For information about	See
9390 Overview of catalog functions	Chapter 7
9391 Binding a buffer to a column in a result set	*SQLBindCol*()
9392 Canceling statement processing	*SQLCancel*()
9393 Returning privileges for a column or columns	*SQLColumnPrivileges*()
9394 Fetching a block of data or scrolling through a result set	*SQLFetchScroll*()
9395 Fetching multiple rows of data	*SQLFetch*()
9396 Returning columns that uniquely identify a row, or 9397 columns automatically updated by a transaction	*SQLSpecialColumns*()
9398 Returning table statistics and indexes	*SQLStatistics*()
9399 Returning a list of tables in a data source	*SQLTables*()
9400 Returning privileges for a table or tables	*SQLTablePrivileges*()

9401 **CHANGE HISTORY**

9402 **Version 2**

9403 Revised generally. See **Alignment with Popular Implementations** on page 2.

NAME

SQLConnect — Establish connections to a data source.

SYNOPSIS

```
SQLRETURN SQLConnect(
  SQLHDBC ConnectionHandle,
  SQLCHAR * ServerName,
  SQLSMALLINT NameLength1,
  SQLCHAR * UserName,
  SQLSMALLINT NameLength2,
  SQLCHAR * Authentication,
  SQLSMALLINT NameLength3);
```

ARGUMENTS

ConnectionHandle [Input]
Connection handle.

ServerName [Input]
The name of the data source to which to connect. All leading and trailing blanks are significant. This is a literal, not an identifier, and the value is not enclosed in quotes (either single or double). If *ServerName* is a zero-length string, a null pointer or DEFAULT, then it indicates the default data source. The length of *ServerName* must not exceed 128 characters.

NameLength1 [Input]
Length of **ServerName*.

UserName [Input]
User identifier.

NameLength2 [Input]
Length of **UserName*.

Authentication [Input]
Authentication string (typically the password).

NameLength3 [Input]
Length of **Authentication*.

RETURN VALUE

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.

DIAGNOSTICS

When *SQLConnect*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling *SQLGetDiagRec*() with a *HandleType* of SQL_HANDLE_DBC and a *Handle* of *ConnectionHandle*. The following SQLSTATE values are commonly returned by *SQLConnect*(). The return code associated with each SQLSTATE value is SQL_ERROR, except that for SQLSTATE values in class 01, the return code is SQL_SUCCESS_WITH_INFO.

01000 — General warning
Implementation-defined informational message.

01S02 — Attribute value changed
The data source did not support the specified value of *ValuePtr* in *SQLSetConnectAttr*() and substituted a similar value.

08001 — Client unable to establish connection
The implementation could not establish a connection to the data source.

9449 08002 — Connection name in use
9450 *ConnectionHandle* had already been used to establish a connection with a data source and
9451 the connection was still open or the user was browsing for a connection.

9452 08004 — Data source rejected the connection
9453 The data source rejected the establishment of the connection for implementation-defined
9454 reasons.

9455 08S01 — Communication link failure
9456 The communication link to the data source failed before the function completed processing.

9457 28000 — Invalid authorization specification
9458 *UserName* or *Authentication* violated restrictions defined by the data source.

9459 HY000 — General error
9460 An error occurred for which there was no specific SQLSTATE and for which no
9461 implementation-specific SQLSTATE was defined. The error message returned by
9462 *SQLGetDiagRec*() in the **MessageText* buffer describes the error and its cause.

9463 HY001 — Memory allocation error
9464 The implementation failed to allocate memory required to support execution or completion
9465 of the function.

9466 HY090 — Invalid string or buffer length
9467 *NameLength1* was less than 0, but not equal to SQL_NTS.

9468 *NameLength1* exceeded the maximum length for a data source name.

9469 *NameLength2* was less than 0, but not equal to SQL_NTS.

9470 *NameLength3* was less than 0, but not equal to SQL_NTS.

9471 HYT00 — Timeout expired
9472 The query timeout period expired before the connection to the data source completed. The
9473 timeout period is set through *SQLSetConnectAttr*(), SQL_ATTR_LOGIN_TIMEOUT.

9474 HYT01 — Connection timeout expired
9475 The connection timeout period expired before the data source responded to the request. The
9476 connection timeout period is set through *SQLSetConnectAttr*(),
9477 SQL_ATTR_CONNECTION_TIMEOUT.

9478 IM001 — Function not supported
9479 The function is not supported on the current connection to the data source.

9480 IM002 — Data source not found and no default driver specified
9481 The data source name specified in *ServerName* was not found in the system information; and
9482 either no default data source was specified or information on the default data source could
9483 not be found in the system information.

9484 **COMMENTS**

9485 Calling *SQLConnect*() establishes a connection to *ServerName*. The mapping from *ServerName* (or
9486 from the default data source) to a physical database is implementation-defined.

9487 The specified data source uses the values of *UserName* and *Authentication* and may apply other
9488 criteria when the application calls *SQLConnect*() to determine whether to accept or reject the
9489 connection. If the data source accepts the connection, then *UserName* becomes the name of its
9490 current user. It is implementation-defined how the data source selects a default catalog and
9491 schema.

9492 *ConnectionHandle* references storage of all information about the connection to the data source,
9493 including status, transaction state, and error information.

9494 The application can establish more than one connection.

9495 **SEE ALSO**

For information about	See
Allocating a handle	*SQLAllocHandle*()
Discovering and enumerating values required to connect to a data source	*SQLBrowseConnect*()
Disconnecting from a data source	*SQLDisconnect*()
Connecting to a data source using a connection string or dialog box	*SQLDriverConnect*()
Returning the setting of a connection attribute	*SQLGetConnectAttr*()
Setting a connection attribute	*SQLSetConnectAttr*()

9505 **CHANGE HISTORY**

9506 **Version 2**

9507 Revised generally. See **Alignment with Popular Implementations** on page 2.

9508 **NAME**

9509 SQLCopyDesc — Copy descriptor information from one descriptor handle to another.

9510 **SYNOPSIS**

```
SQLRETURN SQLCopyDesc(
  SQLHDESC SourceDescHandle,
  SQLHDESC TargetDescHandle);
```

9514 **ARGUMENTS**

9515 *SourceDescHandle* [Input]

9516 Source descriptor handle.

9517 *TargetDescHandle* [Input]

9518 Target descriptor handle. This can be a handle to an application descriptor or to an IPD. The
9519 function returns SQLSTATE HY016 if this is a handle to an IRD.

9520 **RETURN VALUE**

9521 SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.

9522 **DIAGNOSTICS**

9523 When *SQLCopyDesc*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated
9524 SQLSTATE value can be obtained by calling *SQLGetDiagRec*() with a *HandleType* of
9525 SQL_HANDLE_DESC and a *Handle* of *TargetDescHandle*. If an invalid *SourceDescHandle* was
9526 passed in the call, SQL_INVALID_HANDLE is returned, but no SQLSTATE is returned. The
9527 following SQLSTATE values are commonly returned by *SQLCopyDesc*(). The return code
9528 associated with each SQLSTATE value is SQL_ERROR, except that for SQLSTATE values in class
9529 01, the return code is SQL_SUCCESS_WITH_INFO.

9530 When an error is returned, the call to *SQLCopyDesc*() is immediately aborted, and the contents of
9531 the fields in the *TargetDescHandle* descriptor are undefined.

9532 01000 — General warning

9533 Implementation-defined informational message.

9534 08S01 — Communication link failure

9535 The communication link to the data source failed before the function completed processing.

9536 HY000 — General error

9537 An error occurred for which there was no specific SQLSTATE and for which no
9538 implementation-specific SQLSTATE was defined. The error message returned by
9539 *SQLGetDiagRec*() in the **MessageText* buffer describes the error and its cause.

9540 HY001 — Memory allocation error

9541 The implementation failed to allocate memory required to support execution or completion
9542 of the function.

9543 HY007 — Associated statement is not prepared

9544 *SourceDescHandle* was associated with an IRD, and the associated statement handle was not
9545 in the prepared or executed state.

9546 HY010 — Function sequence error

9547 The descriptor handle in *SourceDescHandle* or *TargetDescHandle* was associated with a
9548 statement handle for which an asynchronously executing function (not this one) was called
9549 and was still executing when this function was called.

9550 The descriptor handle in *SourceDescHandle* or *TargetDescHandle* was associated with a
9551 statement handle for which *SQLBulkOperations*(), *SQLExecDirect*(), *SQLExecute*(), or
9552 *SQLSetPos*() was called and returned SQL_NEED_DATA. This function was called before
9553 data was sent for all data-at-execution parameters or columns.

9554 A previous call to the same function returned SQL_STILL_EXECUTING and the present call
9555 specified different values of *SourceDescHandle* or *TargetDescHandle*.

9556 HY016 — Cannot modify an implementation row descriptor
9557 *TargetDescHandle* was a handle for an IRD.

9558 HY021 — Inconsistent descriptor information
9559 The descriptor consistency check failed (see **Consistency Checks** on page 486).

9560 HY092 — Invalid attribute identifier
9561 *SourceDescHandle* and *TargetDescHandle* pertain to different XDBC implementations, and
9562 there is at least one XDBC-defined descriptor field that the source data source supports but
9563 the target data source does not.

9564 HYT01 — Connection timeout expired
9565 The connection timeout period expired before the data source responded to the request. The
9566 connection timeout period is set through *SQLSetConnectAttr*(),
9567 SQL_ATTR_CONNECTION_TIMEOUT.

9568 IM001 — Function not supported
9569 The function is not supported on the current connection to the data source.

9570 **COMMENTS**

9571 A call to *SQLCopyDesc*() copies the fields of *SourceDescHandle* to *TargetDescHandle*.
9572 *SourceDescHandle* can be any type of descriptor handle; *TargetDescHandle* can be any type of
9573 descriptor handle except one that pertains to an IRD.

9574 The following fields are copied, overwriting existing information in the target descriptor:

9575 • In no case is SQL_DESC_ALLOC_TYPE copied; it specifies whether the descriptor handle
9576 was automatically or explicitly allocated.

9577 • If *SourceDescHandle* and *TargetDescHandle* pertain to the same XDBC implementation, then all
9578 other fields are copied, even if the two descriptors are on different connections or in different
9579 environments.

9580 • If *SourceDescHandle* and *TargetDescHandle* pertain to different XDBC implementations, only
9581 XDBC-defined fields are copied; implementation-defined fields are not copied.

9582 When copying a descriptor in which the SQL_DESC_DATA_PTR field is not a null pointer, the
9583 consistency check defined in **Consistency Checks** on page 486 occurs. If the consistency check
9584 fails, SQLSTATE HY021 (Inconsistent descriptor information) is returned and the call to
9585 *SQLCopyDesc*() is immediately aborted.

9586 On this and any other error, the contents of the target descriptor are undefined.

9587 Descriptor handles can be copied across connections or environments.

9588 **Alternative to SQLCopyDesc()**

9589 An application may be able to associate descriptor information with a different statement handle
9590 without calling *SQLCopyDesc*(): An explicitly-allocated descriptor can be associated with
9591 another statement handle on the same connection by setting the SQL_ATTR_APP_ROW_DESC
9592 or SQL_ATTR_APP_PARAM_DESC statement attribute to the handle of the explicitly-allocated
9593 descriptor. However, if the target statement handle is on a different connection, the application
9594 must instead use *SQLCopyDesc*() to copy descriptor field values to a descriptor on that
9595 connection.

9596 **Copying Rows between Tables**

9597 An ARD on one statement handle can serve as the APD on another statement handle. This lets
9598 an application copy rows between tables without copying data at the application level. To do
9599 this, an application calls *SQLCopyDesc*() to copy the fields of an ARD that describes a fetched
9600 row of a table, to the APD for a parameter in an INSERT statement on another statement handle.
9601 When copying across statements on the same connection, the value returned by *SQLGetInfo*()
9602 with the SQL_ACTIVE_STATEMENTS option must be greater than 1 for this operation to
9603 succeed. This is not required when copying across connections.

9604 **Copying from Implementation Descriptors**

9605 If *SourceDescHandle* is an IRD, the statement must be prepared, or *SQLCopyDesc*() fails, setting
9606 SQLSTATEto HY007 (Associated statement is not prepared).

9607 If *SourceDescHandle* is an IPD, it can be copied whether or not the statement is prepared.
9608 However, any automatic descriptor population takes effect before the descriptor is copied. That
9609 is, if the statement is prepared, if the implementation supports automatic population (see the
9610 SQL_ATTR_AUTO_IPD connection attribute), if the application has enabled this feature (by use
9611 of the SQL_ATTR_ENABLE_AUTO_IPD statement attribute), and if the prepared statement has
9612 dynamic parameters, then the implementation populates the IPD with descriptor information,
9613 and this information is copied to *TargetDescHandle*.

9614 **SEE ALSO**

9615	For information about	See
9616	Overview of descriptors	Chapter 13
9617	Getting multiple descriptor fields	*SQLGetDescRec*()
9618	Setting a single descriptor field; list of all descriptor fields	*SQLSetDescField*()
9619	Setting multiple descriptor fields	*SQLSetDescRec*()

9620 **CHANGE HISTORY**

9621 **Version 2**

9622 Revised generally. See **Alignment with Popular Implementations** on page 2.

NAME

SQLDataSources — Return information about a data source.

SYNOPSIS

```
SQLRETURN SQLDataSources(
  SQLHENV EnvironmentHandle,
  SQLUSMALLINT Direction,
  SQLCHAR * ServerName,
  SQLSMALLINT BufferLength1,
  SQLSMALLINT * NameLength1Ptr,
  SQLCHAR * Description,
  SQLSMALLINT BufferLength2,
  SQLSMALLINT * NameLength2Ptr);
```

ARGUMENTS

EnvironmentHandle [Input]
Environment handle.

Direction [Input]
Specifies a method of fetching entries from the list of data sources. SQL_FETCH_FIRST fetches from the beginning of the list. SQL_FETCH_NEXT fetches the next data source in the list.

ServerName [Output]
Pointer to a buffer in which to return the data source name.

BufferLength1 [Input]
Length of the **ServerName* buffer, in octets; this does not need to be longer than SQL_MAX_DSN_LENGTH plus the null terminator.

NameLength1Ptr [Output]
Pointer to a buffer in which to return the total number of octets (excluding the null terminator) available to return in **ServerName*. If the number of octets available to return is greater than or equal to *BufferLength1*, the data source name in **ServerName* is truncated to *BufferLength1* minus the length of a null terminator.

Description [Output]
Pointer to a buffer in which to return the description of the implementation associated with the data source.

BufferLength2 [Input]
Length of the **Description* buffer.

NameLength2Ptr [Output]
Pointer to a buffer in which to return the total number of octets (excluding the null terminator) available to return in **Description*. If the number of octets available to return is greater than or equal to *BufferLength2*, the description in **Description* is truncated to *BufferLength2* minus the length of a null terminator.

RETURN VALUE

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NO_DATA, SQL_ERROR, or SQL_INVALID_HANDLE.

DIAGNOSTICS

When *SQLDataSources*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling *SQLGetDiagRec*() with a *HandleType* of SQL_HANDLE_ENV and a *Handle* of *EnvironmentHandle*. The following SQLSTATE values are commonly returned by *SQLDataSources*(). The return code associated with each SQLSTATE

9670 value is SQL_ERROR, except that for SQLSTATE values in class 01, the return code is
9671 SQL_SUCCESS_WITH_INFO.

9672 01000 — General warning
9673 Implementation-defined informational message.

9674 01004 — String data, right truncation
9675 The buffer **ServerName* was not large enough to return the entire data source name, so the
9676 name was truncated. The length of the entire data source name is returned in
9677 **NameLength1Ptr*.

9678 The buffer **Description* was not large enough to return the entire data source description, so
9679 the description was truncated. The length of the untruncated data source description is
9680 returned in **NameLength2Ptr*.

9681 HY000 — General error
9682 An error occurred for which there was no specific SQLSTATE and for which no
9683 implementation-specific SQLSTATE was defined. The error message returned by
9684 *SQLGetDiagRec*() in the **MessageText* buffer describes the error and its cause.

9685 HY001 — Memory allocation error
9686 The implementation failed to allocate memory required to support execution or completion
9687 of the function.

9688 HY090 — Invalid string or buffer length
9689 *BufferLength1* was less than 0.

9690 *BufferLength2* was less than 0.

9691 HY103 — Invalid retrieval code
9692 *Direction* was not equal to SQL_FETCH_FIRST or SQL_FETCH_NEXT.

9693 **COMMENTS**

9694 An application can call *SQLDataSources*() multiple times to retrieve all data source names. When
9695 there are no more data source names, the function returns SQL_NO_DATA.

9696 If *SQLDataSources*() is called with SQL_FETCH_NEXT initially, or when the immediate previous
9697 call returned SQL_NO_DATA, it returns the first data source name.

9698 It is implementation-defined how data source names are mapped to actual data sources.

9699 **SEE ALSO**

9700	**For information about**	**See**
9701–9702	Discovering and listing values required to connect to a data source	*SQLBrowseConnect*()
9703	Connecting to a data source	*SQLConnect*()
9704–9705	Connecting to a data source using a connection string or dialog box	*SQLDriverConnect*()

9706 **CHANGE HISTORY**

9707 **Version 2**

9708 Revised generally. See **Alignment with Popular Implementations** on page 2.

9709 **NAME**

9710 SQLDescribeCol — Return the result descriptor for one column in the result set.

9711 **SYNOPSIS**

```
9712 SQLRETURN SQLDescribeCol(
9713   SQLHSTMT StatementHandle,
9714   SQLSMALLINT ColumnNumber,
9715   SQLCHAR * ColumnName,
9716   SQLSMALLINT BufferLength,
9717   SQLSMALLINT * NameLengthPtr,
9718   SQLSMALLINT * DataTypePtr,
9719   SQLINTEGER * ColumnSizePtr,
9720   SQLSMALLINT * DecimalDigitsPtr,
9721   SQLSMALLINT * NullablePtr);
```

9722 **ARGUMENTS**

9723 *StatementHandle* [Input]

9724 Statement handle.

9725 *ColumnNumber* [Input]

9726 Column number of result data, ordered sequentially left to right, starting at 1.
9727 *ColumnNumber* can also be set to 0 to describe the bookmark column.

9728 *ColumnName* [Output]

9729 Pointer to a buffer in which to return the column name. This value is read from the
9730 SQL_DESC_NAME field of the IRD. If the column is unnamed or the column name cannot
9731 be determined, an empty string is returned.

9732 *BufferLength* [Input]

9733 Length of the **ColumnName* buffer, in octets.

9734 *NameLengthPtr* [Output]

9735 Pointer to a buffer in which to return the total number of octets (excluding the null
9736 terminator) available to return in **ColumnName*. If the number of octets available to return
9737 is greater than or equal to *BufferLength*, the column name in **ColumnName* is truncated to
9738 *BufferLength* minus the length of a null terminator.

9739 *DataTypePtr* [Output]

9740 Pointer to a buffer in which to return the SQL data type of the column. This value is read
9741 from the SQL_DESC_TYPE field of the IRD, or for date/time and interval types, the concise
9742 type in the SQL_DESC_DATETIME_INTERVAL_CODE field. This is one of the values in
9743 Section D.1 on page 556 or an implementation-specific SQL data type. If the data type
9744 cannot be determined, SQL_UNKNOWN_TYPE is returned.

9745 If SQL_INTERVAL or SQL_DATETIME is returned in **DataTypePtr*, the
9746 SQL_DESC_DATETIME_INTERVAL_CODE record field in the IRD is set to the appropriate
9747 code: SQL_TYPE_DATE for dates, SQL_TYPE_TIME for times, and
9748 SQL_TYPE_TIMESTAMP for timestamps. See the
9749 SQL_DESC_DATETIME_INTERVAL_CODE field in *SQLSetDescField*().

9750 When *ColumnNumber* is 0 (for a bookmark column), SQL_BINARY is returned in
9751 **DataTypePtr*.

9752 For more information, see Section D.1 on page 556.

9753 *ColumnSizePtr* [Output]

9754 Pointer to a buffer in which to return the size of the column on the data source. If the
9755 column size cannot be determined, 0 is returned. Column size is defined in Section D.3.1 on

page 562.

DecimalDigitsPtr [Output]
Pointer to a buffer in which to return the number of decimal digits of the column on the data source. If the number of decimal digits cannot be determined or is not applicable, the implementation returns 0. Decimal digits is defined in Section D.3.2 on page 564.

NullablePtr [Output]
Pointer to a buffer in which to return a value that indicates whether the column allows NULL values. This value is read from the SQL_DESC_NULLABLE field of the IRD. One of the following values:

SQL_NO_NULLS	The column does not allow NULL values.
SQL_NULLABLE	The column allows NULL values.
SQL_NULLABLE_UNKNOWN	The implementation cannot determine if the column allows NULL values.

RETURN VALUE

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE.

DIAGNOSTICS

When *SQLDescribeCol*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling *SQLGetDiagRec*() with a *HandleType* of SQL_HANDLE_STMT and a *Handle* of *StatementHandle*. The following SQLSTATE values are commonly returned by *SQLDescribeCol*(). The return code associated with each SQLSTATE value is SQL_ERROR, except that for SQLSTATE values in class 01, the return code is SQL_SUCCESS_WITH_INFO.

01000 — General warning
Implementation-defined informational message.

01004 — String data, right truncation
The buffer **ColumnName* was not large enough to return the entire column name, so the column name was truncated. The length of the untruncated column name is returned in **NameLengthPtr*.

07005 — Prepared statement not a cursor-specification
The statement associated with *StatementHandle* did not return a result set. There were no columns to describe.

07009 — Invalid descriptor index
ColumnNumber was 0, and the SQL_ATTR_USE_BOOKMARKS statement option was SQL_UB_OFF.

ColumnNumber was less than 0.

ColumnNumber was greater than the number of columns in the result set.

08S01 — Communication link failure
The communication link to the data source failed before the function completed processing.

HY000 — General error
An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by *SQLGetDiagRec*() in the **MessageText* buffer describes the error and its cause.

HY001 — Memory allocation error
The implementation failed to allocate memory required to support execution or completion

9801 of the function.

9802 HY007 — Associated statement is not prepared
9803 The function was called prior to calling *SQLPrepare*(), *SQLExecDirect*(), or a catalog function
9804 for *StatementHandle*.

9805 HY008 — Operation canceled
9806 Asynchronous processing was enabled for *StatementHandle*. The function was called and
9807 before it completed execution, *SQLCancel*() was called on *StatementHandle*. The function
9808 was then called again on *StatementHandle*.

9809 The function was called and, before it completed execution, *SQLCancel*() was called on
9810 *StatementHandle* from a different thread in a multithread application.

9811 HY010 — Function sequence error
9812 An asynchronously executing function (not this one) was called for *StatementHandle* and
9813 was still executing when this function was called.

9814 *SQLBulkOperations*(), *SQLExecDirect*(), *SQLExecute*(), or *SQLSetPos*() was called for
9815 *StatementHandle* and returned SQL_NEED_DATA. This function was called before data was
9816 sent for all data-at-execution parameters or columns.

9817 HY090 — Invalid string or buffer length
9818 *BufferLength* was less than 0.

9819 HYT01 — Connection timeout expired
9820 The connection timeout period expired before the data source responded to the request. The
9821 connection timeout period is set through *SQLSetConnectAttr*(),
9822 SQL_ATTR_CONNECTION_TIMEOUT.

9823 IM001 — Function not supported
9824 The function is not supported on the current connection to the data source.

9825 When the application calls *SQLDescribeCol*() after *SQLPrepare*() and before *SQLExecute*(), it can
9826 return any SQLSTATE that can be returned by *SQLPrepare*() or *SQLExecute*(), depending on
9827 when the data source evaluates the SQL statement associated with *StatementHandle* (see
9828 **Performance Note**).

9829 **COMMENTS**

9830 An application typically calls *SQLDescribeCol*() after a call to *SQLPrepare*() and before or after the
9831 associated call to *SQLExecute*(). An application can also call *SQLDescribeCol*() after a call to
9832 *SQLExecDirect*().

9833 *SQLDescribeCol*() retrieves the column name, type, and length generated by a SELECT statement.
9834 If the column is an expression, the retrieved column name is implementation-defined.

9835 **Performance Note**

9836 The information reported by *SQLDescribeCol*() and *SQLNumResultCols*() is available on all
9837 implementations after an SQL statement is executed.

9838 It is also valid to call the function between the preparation and the execution of a statement.
9839 However, on implementations where *SQLPrepare*() involves little or no work, a call to this
9840 function may involve an analysis of the SQL statement that has not yet occurred.[31] This may
9841 result in more processing than anticipated and harm performance.

9842 ________________

9843 31. For example, with some data sources, the only way to return this description may be to execute a SELECT statement, inhibiting the generation of result-set data by replacing its WHERE clause with a clause such as `WHERE 1 = 2`.

9844 **SEE ALSO**

9845	For information about	See
9846	Binding a buffer to a column in a result set	*SQLBindCol*()
9847	Canceling statement processing	*SQLCancel*()
9848	Returning information about a column in a result set	*SQLColAttribute*()
9849	Fetching multiple rows of data	*SQLFetch*()
9850	Returning the number of result set columns	*SQLNumResultCols*()
9851	Preparing a statement for execution	*SQLPrepare*()

9852 **CHANGE HISTORY**

9853 **Version 2**

9854 Revised generally. See **Alignment with Popular Implementations** on page 2.

NAME

SQLDescribeParam — Return the description of a parameter marker associated with a prepared SQL statement.

SYNOPSIS

```
SQLRETURN SQLDescribeParam(
  SQLHSTMT StatementHandle,
  SQLUSMALLINT ParameterNumber,
  SQLSMALLINT * DataTypePtr,
  SQLUINTEGER * ParameterSizePtr,
  SQLSMALLINT * DecimalDigitsPtr,
  SQLSMALLINT * NullablePtr);
```

ARGUMENTS

StatementHandle [Input]
Statement handle.

ParameterNumber [Input]
Parameter marker number ordered sequentially left to right, starting at 1.

DataTypePtr [Output]
Pointer to a buffer in which to return the SQL data type of the parameter. This value is read from the SQL_DESC_TYPE record field of the IPD. This is one of the values in Section D.1 on page 556 or an implementation-specific SQL data type.

If SQL_INTERVAL or SQL_DATETIME is returned in **DataTypePtr*, the SQL_DESC_DATETIME_INTERVAL_CODE record field in the IRD is set to the appropriate code: SQL_TYPE_DATE for dates, SQL_TYPE_TIME for times, and SQL_TYPE_TIMESTAMP for timestamps. See the SQL_DESC_DATETIME_INTERVAL_CODE field in *SQLSetDescField*(), or Section D.1 on page 556.

ColumnSizePtr [Output]
Pointer to a buffer in which to return the size of the column or expression of the corresponding parameter marker as defined by the data source. If the column size cannot be determined, 0 is returned. Column size is defined in Section D.3.1 on page 562.

DecimalDigitsPtr [Output]
Pointer to a buffer in which to return the number of decimal digits of the column or expression of the corresponding parameter as defined by the data source. If the number of decimal digits cannot be determined or is not applicable, the implementation returns 0. Decimal digits is defined in Section D.3.2 on page 564.

NullablePtr [Output]
Pointer to a buffer in which to return a value that indicates whether the parameter allows NULL values. This value is read from the SQL_DESC_NULLABLE field of the IPD. One of the following:

SQL_NO_NULLS	The parameter does not allow NULL values (this is the default value).
SQL_NULLABLE	The parameter allows NULL values.
SQL_NULLABLE_UNKNOWN	The implementation cannot determine if the parameter allows NULL values.

RETURN VALUE

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE.

DIAGNOSTICS

When *SQLDescribeParam*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling *SQLGetDiagRec*() with a *HandleType* of SQL_HANDLE_STMT and a *Handle* of *StatementHandle*. The following SQLSTATE values are commonly returned by *SQLDescribeParam*(). The return code associated with each SQLSTATE value is SQL_ERROR, except that for SQLSTATE values in class 01, the return code is SQL_SUCCESS_WITH_INFO.

01000 — General warning
Implementation-defined informational message.

07009 — Invalid descriptor index
ParameterNumber was less than 0.

ParameterNumber was greater than the number of parameters in the associated SQL statement.

The parameter marker was part of a non-DML statement.

The parameter marker was part of a SELECT list.

08S01 — Communication link failure
The communication link to the data source failed before the function completed processing.

21S01 — Insert value list does not match column list
The number of parameters in the INSERT statement did not match the number of columns in the table named in the statement.

HY000 — General error
An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by *SQLGetDiagRec*() in the **MessageText* buffer describes the error and its cause.

HY001 — Memory allocation error
The implementation failed to allocate memory required to support execution or completion of the function.

HY008 — Operation canceled
Asynchronous processing was enabled for *StatementHandle*. The function was called and before it completed execution, *SQLCancel*() was called on *StatementHandle*. The function was then called again on *StatementHandle*.

The function was called and, before it completed execution, *SQLCancel*() was called on *StatementHandle* from a different thread in a multithread application.

HY010 — Function sequence error
The function was called prior to calling *SQLPrepare*(), *SQLExecDirect*(), *SQLExecute*(), or a catalog function for *StatementHandle*.

An asynchronously executing function (not this one) was called for *StatementHandle* and was still executing when this function was called.

SQLBulkOperations(), *SQLExecDirect*(), *SQLExecute*(), or *SQLSetPos*() was called for *StatementHandle* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.

HYT01 — Connection timeout expired
The connection timeout period expired before the data source responded to the request. The connection timeout period is set through *SQLSetConnectAttr*(), SQL_ATTR_CONNECTION_TIMEOUT.

9947 IM001 — Function not supported
9948 The function is not supported on the current connection to the data source.

9949 **COMMENTS**

9950 Parameter markers are numbered from left to right, starting with 1, in the order they appear in
9951 the SQL statement.

9952 *SQLDescribeParam*() does not return the type (input, input/output, or output) of a parameter in
9953 an SQL statement. Except in calls to procedures, all parameters in SQL statements are input
9954 parameters. To determine the type of each parameter in a call to a procedure, an application calls
9955 *SQLProcedureColumns*().

9956 **SEE ALSO**

For information about	See
Canceling statement processing	*SQLCancel*()
Executing a prepared SQL statement	*SQLExecute*()
Preparing a statement for execution	*SQLPrepare*()
Binding a buffer to a parameter	*SQLBindParameter*()

9962 **CHANGE HISTORY**

9963 **Version 2**

9964 Function added in this version.

9965 **NAME**

9966 SQLDisconnect — Close the connection associated with a specific connection handle.

9967 **SYNOPSIS**

```
9968 SQLRETURN SQLDisconnect(
9969   SQLHDBC ConnectionHandle);
```

9970 **ARGUMENTS**

9971 *ConnectionHandle* [Input]
9972 Connection handle.

9973 **RETURN VALUE**

9974 SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.

9975 **DIAGNOSTICS**

9976 When *SQLDisconnect*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated
9977 SQLSTATE value can be obtained by calling *SQLGetDiagRec*() with a *HandleType* of
9978 SQL_HANDLE_DBC and a *Handle* of *ConnectionHandle*. The following SQLSTATE values are
9979 commonly returned by *SQLDisconnect*(). The return code associated with each SQLSTATEvalue
9980 is SQL_ERROR, except that for SQLSTATE values in class 01, the return code is
9981 SQL_SUCCESS_WITH_INFO.

9982 01000 — General warning
9983 Implementation-defined informational message.

9984 01002 — Disconnect error
9985 An error occurred during the disconnect. However, the disconnect succeeded.

9986 08003 — Connection does not exist
9987 The connection specified in *ConnectionHandle* was not open.

9988 25000 — Invalid transaction state
9989 There was a transaction in process on the connection specified by *ConnectionHandle*. The
9990 transaction remains active.

9991 HY000 — General error
9992 An error occurred for which there was no specific SQLSTATE and for which no
9993 implementation-specific SQLSTATE was defined. The error message returned by
9994 *SQLGetDiagRec*() in the **MessageText*buffer describes the error and its cause.

9995 HY001 — Memory allocation error
9996 The implementation failed to allocate memory required to support execution or completion
9997 of the function.

9998 HY010 — Function sequence error
9999 An asynchronously executing function was called for a statement handle associated with
10000 *ConnectionHandle* and was still executing when *SQLDisconnect*() was called.

10001 *SQLBulkOperations*(), *SQLExecDirect*(), *SQLExecute*(), or *SQLSetPos*() was called for a
10002 statement handle associated with *ConnectionHandle* and returned SQL_NEED_DATA. This
10003 function was called before data was sent for all data-at-execution parameters or columns.

10004 HYT01 — Connection timeout expired
10005 The connection timeout period expired before the data source responded to the request. The
10006 connection timeout period is set through *SQLSetConnectAttr*(),
10007 SQL_ATTR_CONNECTION_TIMEOUT. The connection remains active.

10008 IM001 — Function not supported
10009 The function is not supported on the current connection to the data source.

COMMENTS

If an application calls *SQLDisconnect*() after *SQLBrowseConnect*() returns SQL_NEED_DATA and before it returns a different return code, then *SQLDisconnect*() cancels the connection browsing process and returns the connection to an unconnected state.

If an application calls *SQLDisconnect*() while there is an incomplete transaction associated with the connection handle, the function returns SQLSTATE 25000 (Invalid transaction state), indicating that the transaction is unchanged and the connection is open. An incomplete transaction is one that has not been completed with *SQLEndTran*().

If an application calls *SQLDisconnect*() before it has freed all statements associated with the connection, it frees those statements, and all descriptors that have been explicitly allocated on the connection, after it successfully disconnects from the data source. However, if one or more of the statements associated with the connection are still executing asynchronously, *SQLDisconnect*() returns SQL_ERROR with a SQLSTATE value of HY010 (Function sequence error).

SEE ALSO

For information about	See
Allocating a handle	*SQLAllocHandle*()
Connecting to a data source	*SQLConnect*()
Connecting to a data source using a connection string or dialog box	*SQLDriverConnect*()
Freeing a connection handle	*SQLFreeHandle*()
Executing a commit or rollback operation	*SQLEndTran*()

CHANGE HISTORY

Version 2

Revised generally. See **Alignment with Popular Implementations** on page 2.

10035 **NAME**

10036 SQLDriverConnect — Connect to a data source using implementation-defined interaction with
10037 the user.

10038 **SYNOPSIS**

```
10039 SQLRETURN SQLDriverConnect(
10040   SQLHDBC ConnectionHandle,
10041   SQLHWND WindowHandle,
10042   SQLCHAR * InConnectionString,
10043   SQLSMALLINT StringLength1,
10044   SQLCHAR * OutConnectionString,
10045   SQLSMALLINT BufferLength,
10046   SQLSMALLINT * StringLength2Ptr,
10047   SQLUSMALLINT DriverCompletion);
```

10048 **ARGUMENTS**

10049 *ConnectionHandle* [Input]
10050 Connection handle.

10051 *WindowHandle* [Input]
10052 Window handle. This is an implementation-defined data structure that indicates the context
10053 for any user interaction. In graphical user-interface environments, the application passes
10054 the handle of the parent window. If the window handle is not applicable (for example, if
10055 the value of *DriverCompletion* directs *SQLDriverConnect*() to not interact with the user), the
10056 application provides a null pointer.

10057 *InConnectionString* [Input]
10058 A full connection string (see the syntax in ''Comments''), a partial connection string, or an
10059 empty string.

10060 *StringLength1* [Input]
10061 Length of **InConnectionString*, in octets.

10062 *OutConnectionString* [Output]
10063 Pointer to a buffer for the completed connection string. Upon successful connection to the
10064 target data source, this buffer contains the completed connection string. Applications
10065 should allocate at least 1024 octets for this buffer.

10066 *BufferLength* [Input]
10067 Length of the **OutConnectionString* buffer.

10068 *StringLength2Ptr* [Output]
10069 Pointer to a buffer in which to return the total number of octets (excluding the null
10070 terminator) available to return in **OutConnectionString*. If the number of octets available to
10071 return is greater than or equal to *BufferLength*, the completed connection string in
10072 **OutConnectionString* is truncated to *BufferLength* minus the length of a null terminator.

10073 *DriverCompletion* [Input]
10074 A flag that indicates whether the implementation should interact with the user:
10075 SQL_DRIVER_PROMPT, SQL_DRIVER_COMPLETE,
10076 SQL_DRIVER_COMPLETE_REQUIRED, or SQL_DRIVER_NOPROMPT. (See
10077 ''Comments'' for additional information.)

10078 **RETURN VALUE**

10079 SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NO_DATA, SQL_ERROR, or
10080 SQL_INVALID_HANDLE.

10081 **DIAGNOSTICS**

10082 When *SQLDriverConnect*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated
10083 SQLSTATE value can be obtained by calling *SQLGetDiagRec*() with a *HandleType* of
10084 SQL_HANDLE_DBC and an *Handle* of *ConnectionHandle*. The following SQLSTATE values are
10085 commonly returned by *SQLDriverConnect*(). The return code associated with each SQLSTATE
10086 value is SQL_ERROR, except that for SQLSTATE values in class 01, the return code is
10087 SQL_SUCCESS_WITH_INFO.

10088 01000 — General warning
10089 Implementation-defined informational message.

10090 01004 — String data, right truncation
10091 The buffer **OutConnectionString* was not large enough to return the entire connection string,
10092 so the connection string was truncated. The length of the untruncated connection string is
10093 returned in **StringLength2Ptr*.

10094 01S00 — Invalid connection string attribute
10095 An invalid attribute keyword was specified by *InConnectionString* but the implementation
10096 was able to connect to the data source anyway.

10097 01S02 — Attribute value changed
10098 The data source did not support the specified value pointed to by *ValuePtr* in
10099 *SQLSetConnectAttr*() and substituted a similar value.

10100 08001 — Client unable to establish connection
10101 The implementation could not establish a connection to the data source.

10102 08002 — Connection name in use
10103 *ConnectionHandle* had already been used to establish a connection with a data source and
10104 the connection was still open.

10105 08004 — Data source rejected the connection
10106 The data source rejected the establishment of the connection for implementation-defined
10107 reasons.

10108 08S01 — Communication link failure
10109 The communication link to the data source failed before the function completed processing.

10110 28000 — Invalid authorization specification
10111 Either the user identifier or the authorization string or both as specified by
10112 *InConnectionString* violated restrictions defined by the data source.

10113 HY000 — General error
10114 An error occurred for which there was no specific SQLSTATE and for which no
10115 implementation-specific SQLSTATE was defined. The error message returned by
10116 *SQLGetDiagRec*() in the **MessageText* buffer describes the error and its cause.

10117 HY001 — Memory allocation error
10118 The implementation failed to allocate memory required to support execution or completion
10119 of the function.

10120 HY090 — Invalid string or buffer length
10121 *StringLength1* was less than 0 and was not equal to SQL_NTS.

10122 *BufferLength* was less than 0.

10123 HY092 — Invalid attribute identifier
10124 *DriverCompletion* was SQL_DRIVER_PROMPT, and *WindowHandle* was a null pointer.

10125 HY110 — Invalid value of DriverCompletion
10126 *DriverCompletion* was not equal to SQL_DRIVER_PROMPT, SQL_DRIVER_COMPLETE,

SQL_DRIVER_COMPLETE_REQUIRED or SQL_DRIVER_NOPROMPT.

HYC00 — Optional feature not implemented
The data source does not support the operation that the application requested.

HYT00 — Timeout expired
The login timeout period expired before the connection to the data source completed. The timeout period is set through *SQLSetConnectAttr*(), SQL_ATTR_LOGIN_TIMEOUT.

HYT01 — Connection timeout expired
The connection timeout period expired before the data source responded to the request. The connection timeout period is set through *SQLSetConnectAttr*(), SQL_ATTR_CONNECTION_TIMEOUT.

IM001 — Function not supported
The function is not supported on the current connection to the data source.

IM002 — Data source not found and no default driver specified
The data source name specified in the connection string (*InConnectionString*) was not found in the system information; and either no default data source was specified or information on the default data source could not be found in the system information.

COMMENTS

SQLDriverConnect() is an alternative to *SQLConnect*() that connects to a data source based on information obtained interactively from the user. *SQLDriverConnect*() is suitable in the following cases:

- To establish a connection using a connection string that contains information more extensive than that allowed by the arguments of *SQLConnect*(), (for example, the data source name, one or more user IDs, one or more passwords, and other information required by the data source).
- To establish a connection using partial or no connection information, relying on the implementation to obtain required information from the user interactively.
- To establish a connection to a data source that is not defined in the system information.
- To establish a connection to a data source using a prearranged connection string.

Once a connection is established, *SQLDriverConnect*() returns the completed connection string. The application can use this string for subsequent connection requests.

Connection Strings

A connection string has the following syntax:

```
connection-string ::= empty-string[;] | attribute[;] | attribute; connection-string
empty-string ::=
attribute ::= attribute-keyword=attribute-value

attribute-keyword ::= DSN | UID | PWD
                        | implementation-defined-attribute-keyword
attribute-value ::= character-string
implementation-defined-attribute-keyword ::= identifier
```

where *character-string* has zero or more characters; *identifier* has one or more characters; *attribute-keyword* is not case-sensitive; *attribute-value* may be case-sensitive; and the value of the **DSN** keyword does not consist solely of blanks. Keywords and attribute values should not contain the characters **[ ] { } ( ) , ; ? * = ! @** \

10170 The connection string may include any number of implementation-defined keywords. If in
10171 *InConnectionString* any keywords are repeated, or if the same or different keywords are used in
10172 ways that would be contradictory, the implementation uses the one that appears first.

10173 The following table describes the attribute values of the **DSN**, **UID**, and **PWD** keywords:

Keyword	Attribute value description
DSN	Name of a data source as returned by *SQLDataSources*() or the data sources dialog box of *SQLDriverConnect*().
UID	A user ID.
PWD	The password corresponding to the user ID, or an empty string if there is no password for the user ID (`PWD=;`).

10180 **Interpretation of InConnectionString**

10181 The implementation retrieves information about a specific data source from the system
10182 information. If *InConnectionString* contains the **DSN** keyword, the implementation retrieves
10183 information about the data source it specifies. If not, if the specified data source is not found in
10184 the system information, or if the application specifies `DSN=DEFAULT`, the implementation
10185 retrieves the information for the default data source.

10186 The information retrieved from the system information augments other information the
10187 application placed in *InConnectionString*. If the application provides connection information in
10188 *InConnectionString* that contradicts the corresponding information for that data source in the
10189 system information, then the information in *InConnectionString* prevails.

10190 Based on the value of *DriverCompletion*, the implementation interacts with the user to obtain
10191 connection information, such as the user ID and password, and connects to the data source:

10192 SQL_DRIVER_NOPROMPT
10193 If the connection string contains sufficient information to establish a connection,
10194 *SQLDriverConnect*() does so. Otherwise, it returns SQL_ERROR.

10195 SQL_DRIVER_PROMPT
10196 The implementation interacts with the user to obtain any connection information not
10197 provided in *InConnectionString*. In graphical user-interface environments, *WindowHandle*
10198 indicates the context in which this interaction occurs; for instance, it may denote the parent
10199 window in which a dialog box appears. If the application provided information in
10200 *InConnectionString* or if the implementation obtained information from the system
10201 information, the prevailing information is used as initial values for the interaction with the
10202 user.

10203 When the user completes the interaction, having specified sufficient information to establish
10204 a connection, the implementation connects to the data source. It also constructs a
10205 connection string from the value of the **DSN** keyword in **InConnectionString* and the other
10206 information resulting from the interaction with the user. It places this connection string in
10207 the **OutConnectionString* buffer.

10208 If the user does not specify sufficient information to establish a connection, the
10209 implementation identifies the missing information and again requests it from the user.

10210 If the user aborts the interaction without specifying sufficient information,
10211 *SQLDriverConnect*() returns SQL_NO_DATA.

10212 SQL_DRIVER_COMPLETE or SQL_DRIVER_COMPLETE_REQUIRED
10213 If *InConnectionString* contains sufficient information to establish a connection, the

implementation does so. If any information is missing or incorrect, the implementation interacts with the user, as described above for SQL_DRIVER_PROMPT. However, if *DriverCompletion* is SQL_DRIVER_COMPLETE_REQUIRED, the implementation restricts the interaction so that the user only specifies the minimum information required to connect to the data source.

On successful connection to the data source, the implementation sets **OutConnectionString* to a connection string that achieved the connection. (This is the value of *InConnectionString*, as modified by the system information and any interaction with the user.) The implementation also sets **StringLength2Ptr* to the length of **OutConnectionString*.

Connection Attributes

The SQL_ATTR_LOGIN_TIMEOUT connection attribute, set using *SQLSetConnectAttr*(), defines the number of seconds to wait for a login request to complete successfully before returning to the application. If the user is prompted to complete the connection string, a waiting period for each login request begins when the connection process starts.

By default, the implementation opens the connection in SQL_MODE_READ_WRITE access mode. To set the access mode to SQL_MODE_READ_ONLY, the application must call *SQLSetConnectAttr*() with the SQL_ATTR_ACCESS_MODE attribute prior to calling *SQLDriverConnect*().

SEE ALSO

For information about	See
Allocating a handle	*SQLAllocHandle*()
Discovering and enumerating values required to connect to a data source	*SQLBrowseConnect*()
Connecting to a data source	*SQLConnect*()
Disconnecting from a data source	*SQLDisconnect*()
Freeing a handle	*SQLFreeHandle*()
Setting a connection attribute	*SQLSetConnectAttr*()

CHANGE HISTORY

Version 2

Function added in this version.

NAME

SQLDrivers — List driver descriptions and driver attribute keywords.

SYNOPSIS

OP
```
SQLRETURN SQLDrivers(
    SQLHENV EnvironmentHandle,
    SQLUSMALLINT Direction,
    SQLCHAR * DriverDescription,
    SQLSMALLINT BufferLength1,
    SQLSMALLINT * DescriptionLengthPtr,
    SQLCHAR * DriverAttributes,
    SQLSMALLINT BufferLength2,
    SQLSMALLINT * AttributesLengthPtr);
```

ARGUMENTS

EnvironmentHandle [Input]
Environment handle.

Direction [Input]
Determines whether the Driver Manager fetches the next driver description in the list (SQL_FETCH_NEXT) or whether the search starts from the beginning of the list (SQL_FETCH_FIRST).

DriverDescription [Output]
Pointer to a buffer in which to return the driver description.

BufferLength1 [Input]
Length of the **DriverDescription* buffer, in octets.

DescriptionLengthPtr [Output]
Pointer to a buffer in which to return the total number of octets (excluding the null terminator) available to return in **DriverDescription*. If the number of octets available to return is greater than or equal to *BufferLength1*, the driver description in **DriverDescription* is truncated to *BufferLength1* minus the length of a null terminator.

DriverAttributes [Output]
Pointer to a buffer in which to return the list of driver attribute value pairs (see ‘‘Comments’’).

BufferLength2 [Input]
Length of the **DriverAttributes* buffer, in octets.

AttributesLengthPtr [Output]
Pointer to a buffer in which to return the total number of octets (excluding the null terminator) available to return in **DriverAttributes*. If the number of octets available to return is greater than or equal to *BufferLength2*, the list of attribute value pairs in **DriverAttributes* is truncated to *BufferLength2* minus the length of a null terminator.

RETURN VALUE

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NO_DATA, SQL_ERROR, or SQL_INVALID_HANDLE.

DIAGNOSTICS

When *SQLDrivers*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling *SQLGetDiagRec*() with a *HandleType* of SQL_HANDLE_ENV and a *Handle* of *EnvironmentHandle*. The following SQLSTATE values are commonly returned by *SQLDrivers*(). The return code associated with each SQLSTATE value is SQL_ERROR, except that for SQLSTATE values in class 01, the return code is

10291 SQL_SUCCESS_WITH_INFO.

10292 01000 — General warning
10293 Implementation-defined informational message.

10294 01004 — String data, right truncation
10295 The buffer **DriverDescription* was not large enough to return the entire driver description, so
10296 the description was truncated. The length of the entire driver description is returned in
10297 **DescriptionLengthPtr*.

10298 The buffer **DriverAttributes* was not large enough to return the entire list of attribute value
10299 pairs, so the list was truncated. The length of the untruncated list of attribute value pairs is
10300 returned in **AttributesLengthPtr*.

10301 HY000 — General error
10302 An error occurred for which there was no specific SQLSTATE and for which no
10303 implementation-specific SQLSTATE was defined. The error message returned by
10304 *SQLGetDiagRec*() in the **MessageText* buffer describes the error and its cause.

10305 HY001 — Memory allocation error
10306 The implementation failed to allocate memory required to support execution or completion
10307 of the function.

10308 HY090 — Invalid string or buffer length
10309 *BufferLength1* was less than 0.

10310 *BufferLength2* was less than 0 or equal to 1.

10311 HY103 — Invalid retrieval code
10312 *Direction* was not equal to SQL_FETCH_FIRST or SQL_FETCH_NEXT.

10313 **COMMENTS**

10314 *SQLDrivers*() is an optional function in the XSQL implementation. However, if it is provided, it
10315 returns information about all accessible data sources, regardless of their XDBC compliance level.

10316 *SQLDrivers*() returns the driver description in the **DriverDescription* buffer. It returns additional
10317 information about the driver in the **DriverAttributes* buffer as a list of keyword-value pairs.
10318 Each pair is terminated with a null octet, and the entire list is terminated with a null octet (that
10319 is, two null octets mark the end of the list). For example, a file-based driver using C syntax might
10320 return the following list of attributes ('\0' represents a null character):

10321
```
FileUsage=1\0FileExtns=*.dbf\0\0
```

10322 If **DriverAttributes* is not large enough to hold the entire list, the list is truncated, *SQLDrivers*()
10323 returns SQLSTATE 01004 (Data truncated), and the length of the list (excluding the final null
10324 terminator) is returned in **AttributesLengthPtr*.

10325 Driver attribute keywords are added from the system information when the driver is installed.

10326 An application can call *SQLDrivers*() multiple times to retrieve all driver descriptions. The
10327 Driver Manager retrieves this information from the system information. When there are no more
10328 driver descriptions, *SQLDrivers*() returns SQL_NO_DATA. If *SQLDrivers*() is called with
10329 SQL_FETCH_NEXT immediately after it returns SQL_NO_DATA, it returns the first driver
10330 description.

10331 If SQL_FETCH_NEXT is passed to *SQLDrivers*() the very first time it is called, *SQLDrivers*()
10332 returns the first data source name.

10333 **SEE ALSO**

For information about	See
Discovering and listing values required to connect to a data source	*SQLBrowseConnect*()
Connecting to a data source	*SQLConnect*()
Returning data source names	*SQLDataSources*()
Connecting to a data source using a connection string or dialog box	*SQLDriverConnect*()

CHANGE HISTORY

Version 2

Function added in this version.

10344 **NAME**

10345 SQLEndTran — Request commit or rollback of all active operations on all statements associated
10346 with a connection, or for all connections associated with an environment.

10347 **SYNOPSIS**

```
10348 SQLRETURN SQLEndTran(
10349   SQLSMALLINT HandleType,
10350   SQLHANDLE Handle,
10351   SQLSMALLINT CompletionType);
```

10352 **ARGUMENTS**

10353 *HandleType* [Input]
10354 Handle type identifier. Contains either SQL_HANDLE_ENV if *Handle* is an environment
10355 handle, or SQL_HANDLE_DBC if *Handle* is a connection handle.

10356 *Handle* [Input]
10357 The handle, of the type indicated by *HandleType*, indicating the scope of the transaction. See
10358 the ''Comments'' section below for more information.

10359 *CompletionType* [Input]
10360 Either SQL_COMMIT or SQL_ROLLBACK.

10361 **RETURN VALUE**

10362 SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.

10363 **DIAGNOSTICS**

10364 When *SQLEndTran*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated
10365 SQLSTATE value can be obtained by calling *SQLGetDiagRec*() with the appropriate *HandleType*
10366 and *Handle*. The following SQLSTATE values are commonly returned by *SQLEndTran*(). The
10367 return code associated with each SQLSTATE value is SQL_ERROR, except that for SQLSTATE
10368 values in class 01, the return code is SQL_SUCCESS_WITH_INFO.

10369 01000 — General warning
10370 Implementation-defined informational message.

10371 08003 — Connection not open
10372 *ConnectionHandle* was not in a connected state.

10373 08007 — Connection failure during transaction
10374 The connection associated with *ConnectionHandle* failed during the execution of the function
10375 and it cannot be determined whether the requested COMMIT or ROLLBACK occurred
10376 before the failure.

10377 25S01 — Transaction state unknown
10378 One or more of the connections in *Handle* failed to complete the transaction with the
10379 outcome specified, and the outcome is unknown.

10380 25S02 — Transaction is still active
10381 The implementation was unable to guarantee that all work in the global transaction could
10382 be completed atomically, and the transaction is still active.

10383 25S03 — Transaction is rolled back
10384 The implementation was unable to guarantee that all work in the global transaction could
10385 be completed atomically, and all work in the transaction active in *Handle* was rolled back.

10386 HY000 — General error
10387 An error occurred for which there was no specific SQLSTATE and for which no
10388 implementation-specific SQLSTATE was defined. The error message returned by
10389 *SQLGetDiagRec*() in the **MessageText* buffer describes the error and its cause.

10390 HY001 — Memory allocation error
10391 The implementation failed to allocate memory required to support execution or completion
10392 of the function.

10393 HY010 — Function sequence error
10394 An asynchronously executing function was called for a statement handle associated with
10395 *ConnectionHandle* and was still executing when *SQLEndTran*() was called.

10396 *SQLBulkOperations*(), *SQLExecDirect*(), *SQLExecute*(), or *SQLSetPos*() was called for a
10397 statement handle associated with *ConnectionHandle* and returned SQL_NEED_DATA. This
10398 function was called before data was sent for all data-at-execution parameters or columns.

10399 HY012 — Invalid transaction operation code
10400 *CompletionType* was neither SQL_COMMIT nor SQL_ROLLBACK.

10401 HY092 — Invalid attribute identifier
10402 *HandleType* was neither SQL_HANDLE_ENV nor SQL_HANDLE_DBC.

10403 HYC00 — Optional feature not implemented
10404 The data source does not support the ROLLBACK operation.

10405 HYT01 — Connection timeout expired
10406 The connection timeout period expired before the data source responded to the request. The
10407 connection timeout period is set through *SQLSetConnectAttr*(),
10408 SQL_ATTR_CONNECTION_TIMEOUT.

10409 IM001 — Function not supported
10410 The function is not supported on the current connection to the data source.

10411 **COMMENTS**

10412 Calling *SQLEndTran*() attempts to complete (commit or roll back, according to *CompletionType*),
10413 all specified transactions:

10414 • If *HandleType* is SQL_HANDLE_DBC, then *Handle* must be a connection handle and
10415 *SQLEndTran*() completes the transaction on that connection.

10416 • If *HandleType* is SQL_HANDLE_ENV, then *Handle* must be an environment handle.
10417 *SQLEndTran*() completes transactions on all connections that are in a connected state on that
10418 environment. *SQLEndTran*() generates at least one diagnostic record for each of these
10419 connections and associates it with the connection handle.

10420 Connections on which no transaction has begun are not affected by *SQLEndTran*(), do not
10421 affect the success or failure of *SQLEndTran*(), and diagnostic information is not associated
10422 with the connection handle.

10423 *SQLEndTran*() returns SQL_SUCCESS only if completion of the transaction succeeds on each
10424 affected connection. If completion of the transaction fails on any connection, *SQLEndTran*()
10425 returns SQL_ERROR and the application can determine the location and cause of the failure
10426 by calling *SQLGetDiagRec*() for each affected connection.

10427 **This function does not simulate a global transaction across all connections and therefore does**
10428 **not use two-phase commit protocols.**

10429 If *CompletionType* is SQL_COMMIT, *SQLEndTran*() issues a commit request for all active
10430 operations on any statement associated with an affected connection. If CompletionType is
10431 SQL_ROLLBACK, *SQLEndTran*() issues a rollback request for all active operations on any
10432 statement associated with an affected connection. If no transactions are active, *SQLEndTran*()
10433 returns SQL_SUCCESS with no effect on any data sources.

10434 If the data source is in manual-commit mode (by calling *SQLSetConnectAttr*() with the
10435 SQL_ATTR_AUTOCOMMIT attribute set to SQL_AUTOCOMMIT_OFF), a new transaction is

10436 implicitly started when an SQL statement that can be contained within a transaction is executed
10437 against the current data source.

10438 **Effects on Cursors**

10439 To determine how transaction operations affect cursors, an application calls *SQLGetInfo*() with
10440 the SQL_CURSOR_ROLLBACK_BEHAVIOR and SQL_CURSOR_COMMIT_BEHAVIOR
10441 options.

10442 If the SQL_CURSOR_ROLLBACK_BEHAVIOR or SQL_CURSOR_COMMIT_BEHAVIOR value
10443 equals SQL_CB_DELETE, *SQLEndTran*() closes and deletes all open cursors on all statements
10444 associated with the connection and discards all pending results. *SQLEndTran*() leaves any
10445 statement present in an allocated (unprepared) state; the application can reuse them for
10446 subsequent SQL requests or can call *SQLFreeStmt*() or *SQLFreeHandle*() with a *HandleType* of
10447 SQL_HANDLE_STMT to deallocate them.

10448 If the SQL_CURSOR_ROLLBACK_BEHAVIOR or SQL_CURSOR_COMMIT_BEHAVIOR value
10449 equals SQL_CB_CLOSE, *SQLEndTran*() closes all open cursors on all statements associated with
10450 the connection. *SQLEndTran*() leaves any statement present in a prepared state; the application
10451 can call *SQLExecute*() for a statement associated with the connection without first calling
10452 *SQLPrepare*().

10453 If the SQL_CURSOR_ROLLBACK_BEHAVIOR or SQL_CURSOR_COMMIT_BEHAVIOR value
10454 equals SQL_CB_PRESERVE, *SQLEndTran*() does not affect open cursors associated with the
10455 connection. Cursors remain at the row they pointed to prior to the call to *SQLEndTran*().

10456 **Effects When No Transaction Active**

10457 Calling *SQLEndTran*() when no transaction is active returns SQL_SUCCESS (indicating that
10458 there is no work to be committed or rolled back) and has no effect on the data source.

10459 Implementations that do not support transactions (the SQL_TXN_CAPABLE option of
10460 *SQLGetInfo*() is SQL_TC_NONE) are effectively always in auto-commit mode. Calling
10461 *SQLEndTran*() always returns SQL_SUCCESS. These implementations do not roll back
10462 transactions; if *CompletionType* is SQL_ROLLBACK, the function fails and sets SQLSTATE to
10463 HYC00 (Optional feature not implemented).

10464 **SEE ALSO**

10465	**For information about**	**See**
10466	Returning information about an implementation	*SQLGetInfo*()
10467	Freeing a handle	*SQLFreeHandle*()
10468	Freeing a statement handle	*SQLFreeStmt*()

10469 **CHANGE HISTORY**

10470 **Version 2**

10471 Revised generally. See **Alignment with Popular Implementations** on page 2.

10472 **NAME**

10473 SQLExecDirect — Execute a preparable statement, using the current values of the parameter
10474 marker variables if any parameters exist in the statement.

10475 **SYNOPSIS**

```
10476 SQLRETURN SQLExecDirect(
10477   SQLHSTMT StatementHandle,
10478   SQLCHAR * StatementText,
10479   SQLINTEGER TextLength);
```

10480 **ARGUMENTS**

10481 *StatementHandle* [Input]
10482 Statement handle.

10483 *StatementText* [Input]
10484 SQL statement to be executed.

10485 *TextLength* [Input]
10486 Length of **StatementText*.

10487 **RETURN VALUE**

10488 SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NEED_DATA, SQL_STILL_EXECUTING,
10489 SQL_ERROR, SQL_NO_DATA, or SQL_INVALID_HANDLE.

10490 **DIAGNOSTICS**

10491 When *SQLExecDirect*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated
10492 SQLSTATE value can be obtained by calling *SQLGetDiagRec*() with a *HandleType* of
10493 SQL_HANDLE_STMT and a *Handle* of *StatementHandle*. The following SQLSTATE values are
10494 commonly returned by *SQLExecDirect*(). The return code associated with each SQLSTATE value
10495 is SQL_ERROR, except that for SQLSTATE values in class 01, the return code is
10496 SQL_SUCCESS_WITH_INFO.

10497 01000 — General warning
10498 Implementation-defined informational message.

10499 01001 — Cursor operation conflict
10500 **StatementText* contained a positioned UPDATE or DELETE statement and no rows or more
10501 than one row were updated or deleted. (For more information about updates to more than
10502 one row, see the description of the SQL_ATTR_SIMULATE_CURSOR statement attribute in
10503 *SQLSetStmtAttr*().)

10504 01S02 — Attribute value changed
10505 A specified statement attribute was invalid and a similar value was temporarily substituted.
10506 See Section 9.2.1 on page 93.

10507 01S07 — Fractional truncation
10508 The data returned for an input/output or output parameter was truncated so as to truncate
10509 the fractional part of a numeric data type; or the fractional portion of the seconds
10510 component of a time, timestamp, or interval data type.

10511 07001 — Wrong number of parameters
10512 The number of parameters specified in *SQLBindParameter*() was less than the number of
10513 parameters in the SQL statement contained in **StatementText*.

10514 07002 — COUNT field incorrect
10515 *SQLBindParameter*() was called with *ParameterValuePtr* set to a null pointer, *StrLen_or_IndPtr*
10516 not set to SQL_NULL_DATA or SQL_DATA_AT_EXEC, and *InputOutputType* not set to
10517 SQL_PARAM_OUTPUT.

10518 07006 — Restricted data type attribute violation
10519 The data value identified by *ValueType* in *SQLBindParameter*() for the bound parameter
10520 could not be converted to the data type identified by *ParameterType* in *SQLBindParameter*().

10521 The data value returned for a parameter bound as SQL_PARAM_INPUT_OUTPUT or
10522 SQL_PARAM_OUTPUT could not be converted to the data type identified by *ValueType* in
10523 *SQLBindParameter*().

10524 (If the data values for one or more rows could not be converted, but one or more rows were
10525 successfully returned, this function returns SQL_SUCCESS_WITH_INFO.)

10526 07S01 — Invalid use of default parameter
10527 A parameter value, set with *SQLBindParameter*(), was SQL_DEFAULT_PARAM, and the
10528 corresponding parameter was not a parameter for a procedure called using the XDBC
10529 escape sequence (see Section 8.3 on page 84).

10530 08S01 — Communication link failure
10531 The communication link to the data source failed before the function completed processing.

10532 22001 — String data, right truncation
10533 The assignment of a character or binary value to a column resulted in the truncation of
10534 non-blank character data or non-null binary data.

10535 22002 — Indicator variable required but not supplied
10536 NULL data was bound to an output parameter whose *StrLen_or_IndPtr* set by
10537 *SQLBindParameter*() was a null pointer.

10538 22025 — Invalid escape sequence
10539 **StatementText* contained ‘‘LIKE *pattern value* ESCAPE *escape character*’’ in the WHERE
10540 clause, and the character following the escape character in the pattern value was not one of
10541 ‘‘%’’ or ‘‘_’’.

10542 34000 — Invalid cursor name
10543 **StatementText* contained a positioned UPDATE or DELETE statement and the cursor
10544 referenced by the statement being executed was not open.

10545 HY000 — General error
10546 An error occurred for which there was no specific SQLSTATE and for which no
10547 implementation-specific SQLSTATE was defined. The error message returned by
10548 *SQLGetDiagRec*() in the **MessageText* buffer describes the error and its cause.

10549 HY001 — Memory allocation error
10550 The implementation failed to allocate memory required to support execution or completion
10551 of the function.

10552 HY008 — Operation canceled
10553 Asynchronous processing was enabled for *StatementHandle*. The function was called and
10554 before it completed execution, *SQLCancel*() was called on *StatementHandle*. The function
10555 was then called again on *StatementHandle*.

10556 The function was called and, before it completed execution, *SQLCancel*() was called on
10557 *StatementHandle* from a different thread in a multithread application.

10558 HY009 — Invalid use of null pointer
10559 *StatementText* was a null pointer.

10560 HY010 — Function sequence error
10561 An asynchronously executing function (not this one) was called for *StatementHandle* and
10562 was still executing when this function was called.

10563 *SQLBulkOperations*(), *SQLExecDirect*(), *SQLExecute*(), or *SQLSetPos*() was called for
10564 *StatementHandle* and returned SQL_NEED_DATA. This function was called before data was
10565 sent for all data-at-execution parameters or columns.

10566 HY090 — Invalid string or buffer length
10567 *TextLength* was less than or equal to 0, but not equal to SQL_NTS.

10568 A parameter value, set with *SQLBindParameter*(), was a null pointer and the parameter
10569 length value was not 0, SQL_NULL_DATA, SQL_DATA_AT_EXEC,
10570 SQL_DEFAULT_PARAM, or less than or equal to SQL_LEN_DATA_AT_EXEC_OFFSET.

10571 A parameter value, set with *SQLBindParameter*(), was not a null pointer and the parameter
10572 length value was less than 0, but was not SQL_NTS, SQL_NULL_DATA,
10573 SQL_DATA_AT_EXEC, SQL_DEFAULT_PARAM, or less than or equal to
10574 SQL_LEN_DATA_AT_EXEC_OFFSET. (This error is reported only if the application data
10575 type is SQL_C_BINARY or SQL_C_CHAR.)

10576 A parameter length value bound by *SQLBindParameter*() was set to SQL_DATA_AT_EXEC;
10577 the SQL type was either SQL_LONGVARCHAR, SQL_LONGVARBINARY, or a long, data-
10578 source-specific data type; and the SQL_NEED_LONG_DATA_LEN option in *SQLGetInfo*()
10579 was ''Y''.

10580 HY105 — Invalid parameter type
10581 A value specified for *InputOutputType* in *SQLBindParameter*() did not accurately describe the
10582 corresponding parameter as it was used in the SQL statement. For example,
10583 SQL_PARAM_OUTPUT was specified for a parameter used other than in conjunction with
10584 a procedure, or SQL_PARAM_INPUT was specified for a parameter that was a return value
10585 from a procedure.

10586 HY109 — Invalid cursor position
10587 **StatementText* contained a positioned UPDATE or DELETE statement and the cursor was
10588 positioned (by *SQLSetPos*() or *SQLFetchScroll*()) on a row that had been deleted or could not
10589 be fetched.

10590 HYC00 — Optional feature not implemented
10591 The data source does not support the combination of the current settings of the
10592 SQL_ATTR_CONCURRENCY and SQL_ATTR_CURSOR_TYPE statement attributes.

10593 The SQL_ATTR_USE_BOOKMARKS statement attribute was set to SQL_UB_VARIABLE,
10594 and the SQL_ATTR_CURSOR_TYPE statement attribute was set to a cursor type for which
10595 the data source does not support bookmarks.

10596 HYT00 — Timeout expired
10597 The query timeout period expired before the data source returned the result set. The
10598 timeout period is set through *SQLSetStmtAttr*(), SQL_ATTR_QUERY_TIMEOUT.

10599 HYT01 — Connection timeout expired
10600 The connection timeout period expired before the data source responded to the request. The
10601 connection timeout period is set through *SQLSetConnectAttr*(),
10602 SQL_ATTR_CONNECTION_TIMEOUT.

10603 IM001 — Function not supported
10604 The function is not supported on the current connection to the data source.

10605 In addition, the following diagnostics, defined in the X/Open **SQL** specification, can occur based
10606 on the SQL statement text:

	Success with warning [SQL_SUCCESS_WITH_INFO]
01003	— NULL value eliminated in set function.
01004	— String data, right truncation.
01006	— Privilege not revoked.
01007	— Privilege not granted.
	Cardinality violation
21S01	— Insert value does not match column list.
21S02	— Degree of derived table does not match column list.
	Data exception
22003	— Numeric value out of range.
22007	— Invalid date/time format.
22008	— Date/time field overflow.
22012	— Division by zero.
22015	— Interval field overflow.
22018	— Invalid character value for cast specification.
22019	— Invalid escape character.
23000	**Integrity constraint violation**
24000	**Invalid cursor state**
42000	**Syntax error or access violation**
42S01	— Base table or view already exists.
42S02	— Base table or view not found.
42S11	— Index already exists.
42S12	— Index not found.
42S21	— Column already exists.
42S22	— Column not found.
44000	**WITH CHECK OPTION violation**

COMMENTS

SQLExecDirect() is the fastest way to submit an SQL statement for one-time execution.

The application calls *SQLExecDirect*() to send an SQL statement to the data source. The implementation first makes any necessary modifications to the statement so that the result uses the form of SQL that the data source supports; in particular, the implementation translates all occurrences of the XDBC escape sequences defined in Section 8.3 on page 84 into the data-source-specific SQL language.

The application can include one or more parameter markers in the SQL statement. To include a parameter marker, the application embeds a question mark into the SQL statement at the appropriate position.

If the SQL statement is a SELECT statement, and if the application called *SQLSetCursorName*() to associate a cursor with a statement, then the implementation uses the specified cursor. Otherwise, it generates a cursor name.

If the data source is in manual-commit mode (requiring explicit transaction initiation), and a transaction has not already been initiated, it initiates a transaction before executing the SQL statement.

If an application uses *SQLExecDirect*() to submit a COMMIT or ROLLBACK statement, it will not be interoperable between data sources. To commit or roll back a transaction, an application calls *SQLEndTran*().

If *SQLExecDirect*() encounters a data-at-execution parameter, it returns SQL_NEED_DATA. The application sends the data using *SQLParamData*() and *SQLPutData*(). See *SQLBindParameter*(), *SQLParamData*(), and *SQLPutData*() for more information.

10655 A call to *SQLExecDirect*() that executes a searched UPDATE or DELETE statement that does not
10656 affect any rows at the data source returns SQL_NO_DATA.

10657 If the value of the SQL_ATTR_PARAMSET_SIZE statement attribute is greater than 1, and the
10658 SQL statement contains at least one parameter marker, *SQLExecDirect*() executes the SQL
10659 statement once for each set of parameter values from the arrays pointed to by the
10660 ParameterValuePointer argument in the call to *SQLBindParameter*().

10661 **SEE ALSO**

	For information about	See
10662	**For information about**	**See**
10663	Binding a buffer to a column in a result set	*SQLBindCol*()
10664	Canceling statement processing	*SQLCancel*()
10665	Executing a prepared SQL statement	*SQLExecute*()
10666	Fetching a block of data or scrolling through a result set	*SQLFetchScroll*()
10667	Fetching multiple rows of data	*SQLFetch*()
10668	Returning a cursor name	*SQLGetCursorName*()
10669	Fetching part or all of a column of data	*SQLGetData*()
10670	Returning the next parameter to send data for	*SQLParamData*()
10671	Preparing a statement for execution	*SQLPrepare*()
10672	Sending parameter data at execution time	*SQLPutData*()
10673	Setting a cursor name	*SQLSetCursorName*()
10674	Setting a statement attribute	*SQLSetStmtAttr*()
10675	Executing a commit or rollback operation	*SQLEndTran*()

10676 **CHANGE HISTORY**

10677 **Version 2**

10678 Revised generally. See **Alignment with Popular Implementations** on page 2.

10679 **NAME**

10680 SQLExecute — Execute a prepared statement, using the current values of the parameter marker
10681 variables if any parameter markers exist in the statement.

10682 **SYNOPSIS**

```
10683 SQLRETURN SQLExecute(
10684   SQLHSTMT StatementHandle);
```

10685 **ARGUMENTS**

10686 *StatementHandle* [Input]
10687 Statement handle.

10688 **RETURN VALUE**

10689 SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NEED_DATA, SQL_STILL_EXECUTING,
10690 SQL_ERROR, SQL_NO_DATA, or SQL_INVALID_HANDLE.

10691 **DIAGNOSTICS**

10692 When *SQLExecute*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated
10693 SQLSTATE value can be obtained by calling *SQLGetDiagRec*() with a *HandleType* of
10694 SQL_HANDLE_STMT and a *Handle* of *StatementHandle*. The following SQLSTATE values are
10695 commonly returned by *SQLExecute*(). The return code associated with each SQLSTATE value is
10696 SQL_ERROR, except that for SQLSTATE values in class 01, the return code is
10697 SQL_SUCCESS_WITH_INFO.

10698 ''The statement'' in the following list means the prepared statement associated with
10699 *StatementHandle* based on a previous call to *SQLPrepare*().

10700 01000 — General warning
10701 Implementation-defined informational message.

10702 01001 — Cursor operation conflict
10703 **StatementText* contained a positioned UPDATE or DELETE statement and no rows or more
10704 than one row were updated or deleted. (For more information about updates to more than
10705 one row, see the description of the SQL_ATTR_SIMULATE_CURSOR statement attribute in
10706 *SQLSetStmtAttr*().)

10707 01S02 — Attribute value changed
10708 A specified statement attribute was invalid and a similar value was temporarily substituted.
10709 See Section 9.2.1 on page 93.

10710 01S07 — Fractional truncation
10711 The data returned for an input/output or output parameter was truncated so as to truncate
10712 the fractional part of a numeric data type; or the fractional portion of the seconds
10713 component of a time, timestamp, or interval data type.

10714 07001 — Wrong number of parameters
10715 The number of parameters specified in *SQLBindParameter*() was less than the number of
10716 parameters in the statement.

10717 07002 — COUNT field incorrect
10718 *SQLBindParameter*() was called with *ParameterValuePtr* set to a null pointer, *StrLen_or_IndPtr*
10719 not set to SQL_NULL_DATA or SQL_DATA_AT_EXEC, and *InputOutputType* not set to
10720 SQL_PARAM_OUTPUT.

10721 07006 — Restricted data type attribute violation
10722 The data value identified by *ValueType* in *SQLBindParameter*() for the bound parameter
10723 could not be converted to the data type identified by *ParameterType* in *SQLBindParameter*().

10724 The data value returned for a parameter bound as SQL_PARAM_INPUT_OUTPUT or
10725 SQL_PARAM_OUTPUT could not be converted to the data type identified by *ValueType* in
10726 *SQLBindParameter*().

10727 (If the data values for one or more rows could not be converted, but one or more rows were
10728 successfully returned, this function returns SQL_SUCCESS_WITH_INFO.)

10729 07S01 — Invalid use of default parameter
10730 A parameter value, set with *SQLBindParameter*(), was SQL_DEFAULT_PARAM, and the
10731 corresponding parameter was not a parameter for a procedure called using the XDBC
10732 escape sequence (see Section 8.3 on page 84).

10733 08S01 — Communication link failure
10734 The communication link to the data source failed before the function completed processing.

10735 22001 — String data, right truncation
10736 The assignment of a character or binary value to a column resulted in the truncation of
10737 non-blank (character) or non-null (binary) characters or octets.

10738 22002 — Indicator variable required but not supplied
10739 NULL data was bound to an output parameter whose *StrLen_or_IndPtr* set by
10740 *SQLBindParameter*() was a null pointer.

10741 22025 — Invalid escape sequence
10742 The statement contained ‘‘LIKE *pattern value* ESCAPE *escape character*’’ in the WHERE
10743 clause, and the character following the escape character in the pattern value was not one of
10744 ‘‘%’’ or ‘‘_’’.

10745 42000 — Syntax error or access violation
10746 The user did not have permission to execute the statement.

10747 HY000 — General error
10748 An error occurred for which there was no specific SQLSTATE and for which no
10749 implementation-specific SQLSTATE was defined. The error message returned by
10750 *SQLGetDiagRec*() in the **MessageText* buffer describes the error and its cause.

10751 HY001 — Memory allocation error
10752 The implementation failed to allocate memory required to support execution or completion
10753 of the function.

10754 HY008 — Operation canceled
10755 Asynchronous processing was enabled for *StatementHandle*. The function was called and
10756 before it completed execution, *SQLCancel*() was called on *StatementHandle*. The function
10757 was then called again on *StatementHandle*.

10758 The function was called and, before it completed execution, *SQLCancel*() was called on
10759 *StatementHandle* from a different thread in a multithread application.

10760 HY010 — Function sequence error
10761 An asynchronously executing function (not this one) was called for *StatementHandle* and
10762 was still executing when this function was called.

10763 *SQLBulkOperations*(), *SQLExecDirect*(), *SQLExecute*(), or *SQLSetPos*() was called for
10764 *StatementHandle* and returned SQL_NEED_DATA. This function was called before data was
10765 sent for all data-at-execution parameters or columns.

10766 *StatementHandle* was not prepared. Either *StatementHandle* was not in an executed state, or a
10767 cursor was open on *StatementHandle* and *SQLFetch*() or *SQLFetchScroll*() had been called.

10768 *StatementHandle* was not prepared. It was in an executed state and either no result set was
10769 associated with *StatementHandle* or *SQLFetch*() or *SQLFetchScroll*() had not been called.

HY090 — Invalid string or buffer length

A parameter value, set with *SQLBindParameter*(), was a null pointer and the parameter length value was not 0, SQL_NULL_DATA, SQL_DATA_AT_EXEC, SQL_DEFAULT_PARAM, or less than or equal to SQL_LEN_DATA_AT_EXEC_OFFSET.

A parameter value, set with *SQLBindParameter*(), was not a null pointer and the parameter length value was less than 0, but was not SQL_NTS, SQL_NULL_DATA, SQL_DEFAULT_PARAM, or SQL_DATA_AT_EXEC, or less than or equal to SQL_LEN_DATA_AT_EXEC_OFFSET. (This error is reported only if the application data type is SQL_C_BINARY or SQL_C_CHAR.)

A parameter length value bound by *SQLBindParameter*() was set to SQL_DATA_AT_EXEC; the SQL type was either SQL_LONGVARCHAR, SQL_LONGVARBINARY, or a long, data-source-specific data type; and the SQL_NEED_LONG_DATA_LEN option in *SQLGetInfo*() was "Y".

HY105 — Invalid parameter type

A value specified for *InputOutputType* in *SQLBindParameter*() did not accurately describe the corresponding parameter as it was used in the SQL statement. For example, SQL_PARAM_OUTPUT was specified for a parameter used other than in conjunction with a procedure, or SQL_PARAM_INPUT was specified for a parameter that was a return value from a procedure.

HY109 — Invalid cursor position

The statement was a positioned UPDATE or DELETE statement and the cursor was positioned (by *SQLSetPos*() or *SQLFetchScroll*()) on a row that had been deleted or could not be fetched.

HYC00 — Optional feature not implemented

The data source does not support the combination of the current settings of the SQL_ATTR_CONCURRENCY and SQL_ATTR_CURSOR_TYPE statement attributes.

The SQL_ATTR_USE_BOOKMARKS statement attribute was set to SQL_UB_VARIABLE, and the SQL_ATTR_CURSOR_TYPE statement attribute was set to a cursor type for which the data source does not support bookmarks.

HYT00 — Timeout expired

The query timeout period expired before the data source returned the result set. The timeout period is set through *SQLSetStmtAttr*(), SQL_ATTR_QUERY_TIMEOUT.

HYT01 — Connection timeout expired

The connection timeout period expired before the data source responded to the request. The connection timeout period is set through *SQLSetConnectAttr*(), SQL_ATTR_CONNECTION_TIMEOUT.

IM001 — Function not supported

The function is not supported on the current connection to the data source.

In addition, the following diagnostics, defined in the X/Open **SQL** specification, can occur based on the SQL statement text:

Success with warning [SQL_SUCCESS_WITH_INFO]

01003 — NULL value eliminated in set function.

01004 — String data, right truncation.

10813 01006 — Privilege not revoked.
10814 01007 — Privilege not granted.
10815 **Cardinality violation**
10816 21S02 — Degree of derived table does not match column list.
10817 **Data exception**
10818 22003 — Numeric value out of range.
10819 22007 — Invalid date/time format.
10820 22008 — Date/time field overflow.
10821 22012 — Division by zero.
10822 22015 — Interval field overflow.
10823 22018 — Invalid character value for cast specification.
10824 22019 — Invalid escape character.
10825 23000 **Integrity constraint violation**
10826 24000 **Invalid cursor state**
10827 44000 **WITH CHECK OPTION violation**

10828 *SQLExecute*() can return any SQLSTATE that can be returned by *SQLPrepare*() based on when the
10829 data source evaluates the SQL statement associated with the statement.

10830 **COMMENTS**

10831 *SQLExecute*() executes a statement prepared by *SQLPrepare*(). After the application processes or
10832 discards the results from a call to *SQLExecute*(), the application can call *SQLExecute*() again with
10833 new parameter values.

10834 To execute a SELECT statement more than once, the application must call *SQLCloseCursor*()
10835 before reexecuting the SELECT statement.

10836 If the data source is in manual-commit mode (requiring explicit transaction initiation), and a
10837 transaction has not already been initiated, it initiates a transaction before executing the SQL
10838 statement.

10839 If an application uses *SQLPrepare*() to prepare and *SQLExecute*() to submit a COMMIT or
10840 ROLLBACK statement, it will not be interoperable between data sources. To commit or roll back
10841 a transaction, call *SQLEndTran*().

10842 If *SQLExecute*() encounters a data-at-execution parameter, it returns SQL_NEED_DATA. The
10843 application sends the data using *SQLParamData*() and *SQLPutData*(). See *SQLBindParameter*(),
10844 *SQLParamData*(), and *SQLPutData*() for more information.

10845 A call to *SQLExecute*() that executes a searched UPDATE or DELETE statement that does not
10846 affect any rows at the data source returns SQL_NO_DATA.

10847 If the value of the SQL_ATTR_PARAMSET_SIZE statement attribute is greater than 1, and the
10848 SQL statement contains at least one parameter marker, *SQLExecute*() executes the SQL statement
10849 once for each set of parameter values in the arrays pointed to by **ParameterValuePtr* in the call to
10850 *SQLBindParameter*().

10851 **SEE ALSO**

	For information about	See
10852		
10853	Binding a buffer to a column in a result set	*SQLBindCol*()
10854	Canceling statement processing	*SQLCancel*()
10855	Closing the cursor	*SQLCloseCursor*()

Executing an SQL statement	*SQLExecDirect*()
Fetching a block of data or scrolling through a result set	*SQLFetchScroll*()
Fetching multiple rows of data	*SQLFetch*()
Freeing a statement handle	*SQLFreeStmt*()
Returning a cursor name	*SQLGetCursorName*()
Fetching part or all of a column of data	*SQLGetData*()
Returning the next parameter to send data for	*SQLParamData*()
Preparing a statement for execution	*SQLPrepare*()
Sending parameter data at execution time	*SQLPutData*()
Setting a cursor name	*SQLSetCursorName*()
Setting a statement attribute	*SQLSetStmtAttr*()
Executing a commit or rollback operation	*SQLEndTran*()

CHANGE HISTORY

Version 2

Revised generally. See **Alignment with Popular Implementations** on page 2.

10871 **NAME**

10872 SQLFetch — Fetch the next row-set of data from the result set and return data for all bound
10873 columns.

10874 **SYNOPSIS**

```
10875 SQLRETURN SQLFetch(
10876   SQLHSTMT StatementHandle);
```

10877 **ARGUMENTS**

10878 *StatementHandle* [Input]
10879 Statement handle.

10880 **RETURN VALUE**

10881 SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NO_DATA, SQL_STILL_EXECUTING,
10882 SQL_ERROR, or SQL_INVALID_HANDLE.

10883 **DIAGNOSTICS**

10884 When *SQLFetch*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated
10885 SQLSTATE value can be obtained by calling *SQLGetDiagRec*() with a *HandleType* of
10886 SQL_HANDLE_STMT and a *Handle* of *StatementHandle*. The following SQLSTATE values are
10887 commonly returned by *SQLFetch*().

10888 The return code associated with each SQLSTATE value is SQL_ERROR, except that for
10889 SQLSTATE values in class 01, the return code is SQL_SUCCESS_WITH_INFO, and except that, if
10890 the row-set size is greater than 1 and the operation was applied to at least one row successfully,
10891 the return code is SQL_SUCCESS_WITH_INFO.

10892 If an error occurs on a single column, *SQLGetDiagField*() can be called with a *DiagIdentifier* of
10893 SQL_DIAG_COLUMN_NUMBER to determine the column the error occurred on; and
10894 *SQLGetDiagField*() can be called with a *DiagIdentifier* of SQL_DIAG_ROW_NUMBER to
10895 determine the row containing that column.

10896 01000 — General warning
10897 Implementation-defined informational message.

10898 01004 — String data, right truncation
10899 String or binary data returned for a column resulted in the truncation of non-blank character
10900 or non-NULL binary data. If it was a string value, it was right truncated.

10901 01S01 — Error in row
10902 An error occurred while fetching one or more rows.

10903 01S07 — Fractional truncation
10904 The data returned for a column was truncated. For numeric data types, the fractional part of
10905 the number was truncated. For time, timestamp, and interval data types containing a time
10906 component, the fractional portion of the time was truncated.

10907 07006 — Restricted data type attribute violation
10908 The data value of a column in the result set could not be converted to the data type
10909 specified by *TargetType*in *SQLBindCol*().

10910 Column 0 was bound with a data type of SQL_C_VARBOOKMARK and the
10911 SQL_ATTR_USE_BOOKMARKS statement option was not set to SQL_UB_VARIABLE.

10912 08S01 — Communication link failure
10913 The communication link to the data source failed before the function completed processing.

10914 22001 — String data, right truncation
10915 A bookmark returned for a column was truncated.

10916 22002 — Indicator variable required but not supplied
10917 A null value was fetched into a column whose pointer (the *StrLen_or_IndValue* argument to
10918 *SQLBindCol*() or SQL_DESC_INDICATOR_PTR set by *SQLSetDescField*() or
10919 *SQLSetDescRec*()) was a null pointer.

10920 22003 — Numeric value out of range
10921 Returning the numeric value (as numeric or string) for one or more bound columns would
10922 have caused the whole (as opposed to fractional) part of the number to be truncated.

10923 For more information, see Section D.6 on page 576.

10924 22007 — Invalid date/time format
10925 A character column in the result set was bound to a date, time, or timestamp C structure,
10926 and a value in the column was, respectively, an invalid date, time, or timestamp.

10927 22012 — Division by zero
10928 A value from an arithmetic expression was returned which resulted in division by zero.

10929 22015 — Interval field overflow
10930 An exact numeric column in the result set was bound to an interval C structure and
10931 returning the data caused a loss of significant digits.

10932 An interval column in the result set was bound to an interval C structure and returning the
10933 data caused a loss of significant digits.

10934 Data in the result set was bound to an interval C structure and there was no representation
10935 of the data in the interval C structure.

10936 22018 — Invalid character value for cast specification
10937 A character column in the result set was bound to a character C buffer and the column
10938 contained a character for which there was no representation in the character set of the
10939 buffer.

10940 A character column in the result set was bound to an approximate numeric C buffer and a
10941 value in the column could not be cast to a valid approximate numeric value.

10942 A character column in the result set was bound to an exact numeric C buffer and a value in
10943 the column could not be cast to a valid exact numeric value.

10944 A character column in the result set was bound to a date/time or interval C buffer and a
10945 value in the column could not be cast to a valid date/time or interval value.

10946 24000 — Invalid cursor state
10947 *StatementHandle* was in an executed state but no result set was associated with
10948 *StatementHandle.*

10949 40001 — Serialization failure
10950 The transaction in which the fetch was executed was terminated to prevent deadlock.

10951 HY000 — General error
10952 An error occurred for which there was no specific SQLSTATE and for which no
10953 implementation-specific SQLSTATE was defined. The error message returned by
10954 *SQLGetDiagRec*() in the **MessageText* buffer describes the error and its cause.

10955 HY001 — Memory allocation error
10956 The implementation failed to allocate memory required to support execution or completion
10957 of the function.

10958 HY008 — Operation canceled
10959 Asynchronous processing was enabled for *StatementHandle*. The function was called and
10960 before it completed execution, *SQLCancel*() was called on *StatementHandle*. The function

10961 was then called again on *StatementHandle*.

10962 The function was called and, before it completed execution, *SQLCancel*() was called on
10963 *StatementHandle* from a different thread in a multithread application.

10964 HY010 — Function sequence error
10965 *StatementHandle* was not in an executed state. The function was called without first calling
10966 *SQLExecDirect*(), *SQLExecute*(), or a catalog function.

10967 An asynchronously executing function (not this one) was called for *StatementHandle* and
10968 was still executing when this function was called.

10969 *SQLBulkOperations*(), *SQLExecDirect*(), *SQLExecute*(), or *SQLSetPos*() was called for
10970 *StatementHandle* and returned SQL_NEED_DATA. This function was called before data was
10971 sent for all data-at-execution parameters or columns.

10972 HY107 — Row value out of range
10973 The value specified with the SQL_ATTR_CURSOR_TYPE statement attribute was
10974 SQL_CURSOR_KEYSET_DRIVEN, but the value specified with the
10975 SQL_ATTR_KEYSET_SIZE statement attribute was greater than 0 and less than the value
10976 specified with the SQL_ATTR_ROW_ARRAY_SIZE statement attribute.

10977 HYC00 — Optional feature not implemented
10978 The data source does not support the conversion specified by the combination of *TargetType*
10979 in *SQLBindCol*() and the SQL data type of the corresponding column.

10980 HYT01 — Connection timeout expired
10981 The connection timeout period expired before the data source responded to the request. The
10982 connection timeout period is set through *SQLSetConnectAttr*(),
10983 SQL_ATTR_CONNECTION_TIMEOUT.

10984 IM001 — Function not supported
10985 The function is not supported on the current connection to the data source.

10986 **COMMENTS**

10987 **Overview**

10988 *SQLFetch*() returns the next row-set in the result set. It can be called only while a result set exists
10989 — that is, after a call that creates a result set and before the cursor over that result set is closed. If
10990 any columns are bound, it returns the data in those columns. If the application has specified a
10991 pointer to a row status array or a buffer in which to return the number of rows fetched,
10992 *SQLFetch*() returns this information as well.

10993 *SQLFetch*() is equivalent to calling *SQLFetchScroll*() with *FetchOrientation* set to
10994 SQL_FETCH_NEXT. Calls to *SQLFetch*() can be mixed with calls to *SQLFetchScroll*().

10995 **Positioning the Cursor**

10996 When the result set is created, the cursor is positioned before the start of the result set. The first
10997 call to *SQLFetch*() positions the cursor to row 1 and fetches a row-set starting there.

10998 Subsequent calls to *SQLFetch*() move the cursor to the start of the next row-set by advancing the
10999 number of rows in the row-set, as specified by the SQL_ATTR_ROW_ARRAY_SIZE statement
11000 attribute.[32] If this advance of the cursor is beyond the last row of the result set, the cursor moves

11001 ___

11002 32. Applications are free to change the row-set size between fetches. The cursor movement is based on the row-set size as of the previous fetch; the number of rows fetched is based on the current row-set size.

11003 there, and *SQLFetch*() returns SQL_NO_DATA.

11004 If there are not sufficient rows left in the result set to fetch a complete row-set of the row-set size
11005 specified by SQL_ATTR_ROW_ARRAY_SIZE, then *SQLFetch*() returns a partial row-set. The
11006 remaining rows are empty and have a status of SQL_ROW_NOROW.

11007 After *SQLFetch*() returns, the cursor is positioned on the first row of the row-set.

11008 For example, suppose a result set has 100 rows and the row-set size is 5. The following table
11009 shows the row-set and return code returned by *SQLFetch*() for different starting positions.

11010 Current 11011 Row-set	Return code	New row-set	Rows Fetched
11012 Before start	SQL_SUCCESS	1 to 5	5
11013 1 to 5	SQL_SUCCESS	6 to 10	5
11014 91 to 95	SQL_SUCCESS	96 to 100	5
11015 93 to 97 11016	SQL_SUCCESS	98 to 100. Rows 4 and 5 of the row status array are set to SQL_ROW_NOROW.	3
11017 96 to 100 11018	SQL_NO_DATA	None. Rows 1 to 5 of the row status array are set to SQL_ROW_NOROW.	0
11019 99 to 100 11020	SQL_NO_DATA	None. Rows 1 to 5 of the row status array are set to SQL_ROW_NOROW.	0
11021 After end 11022	SQL_NO_DATA	None. Rows 1 to 5 of the row status array are set to SQL_ROW_NOROW.	0

11023 **Returning Data in Bound Columns**

11024 As *SQLFetch*() returns each row, it places the data for each bound column in the buffer bound to
11025 that column. If no columns are bound, *SQLFetch*() does not return any data but does move the
11026 cursor forward. The data can still be retrieved with *SQLGetData*() if the
11027 SQL_GETDATA_EXTENSIONS option of *SQLGetInfo*() is SQL_GD_BLOCK.

11028 For each bound column in a row, *SQLFetch*() does the following:

11029 1. Sets the length/indicator buffer to SQL_NULL_DATA and proceeds to the next column if
11030 the data is NULL. If the data is NULL and no length/indicator buffer was bound,
11031 *SQLFetch*() returns SQLSTATE 22002 (Indicator variable required but not supplied) for the
11032 row and proceeds to the next row. For information about how to determine the address of
11033 the length/indicator buffer, see **Buffer Addresses** on page 217.

11034 If the data for the column is not NULL, *SQLFetch*() proceeds to step 2.

11035 2. If the SQL_ATTR_MAX_LENGTH statement attribute is implemented and has a nonzero
11036 value and the column contains character or binary data, the data is truncated to
11037 SQL_ATTR_MAX_LENGTH octets. (SQL_ATTR_MAX_LENGTH is intended to reduce
11038 network traffic. It is generally implemented by the data source, which truncates the data
11039 before returning it across the network. To guarantee that data is truncated to a particular
11040 size, an application should allocate a buffer of that size and specify the size in the
11041 *ValueMax* argument in *SQLBindCol*().)

11042 3. Converts the data to the type specified by *TargetType* in *SQLBindCol*().

11043 4. If the data was converted to a variable-length data type, such as character or binary, and if
11044 the length of the character data (including the null terminator), *SQLFetch*() truncates the

11045 data to the length of the data buffer less the length of a null terminator. It then null-
11046 terminates the data. If the length of binary data exceeds the length of the data buffer,
11047 *SQLFetch*() truncates it to the length of the data buffer. The length of the data buffer is
11048 specified with *BufferLength* in *SQLBindCol*().

11049 *SQLFetch*() never truncates data converted to fixed-length data types; it always assumes
11050 that the length of the data buffer is the size of the data type.

11051 5. Places the converted (and possibly truncated) data in the data buffer. For information
11052 about how to determine the address of the data buffer, see **Buffer Addresses** on page 217.

11053 6. Places the length of the data in the length/indicator buffer. If the indicator pointer and the
11054 length pointer were both set to the same buffer (as a call to *SQLBindCol*() does), the length
11055 is written in the buffer for valid data and SQL_NULL_DATA is written in the buffer for
11056 NULL data. If no length/indicator buffer was bound, *SQLFetch*() does not return the
11057 length.

11058 — For character or binary data, this is the length of the data after conversion and before
11059 truncation due to the data buffer being too small. If the implementation cannot
11060 determine the length of the data after conversion, as is sometimes the case with long
11061 data, it sets the length to SQL_NO_TOTAL. If data was truncated due to the
11062 SQL_ATTR_MAX_LENGTH statement attribute, the value of this attribute (as opposed
11063 to the actual length) is placed in the length/indicator buffer. This is because this
11064 attribute is designed to truncate data on the server before conversion, so the
11065 implementation has no way of figuring out what the actual length is.

11066 — For all other data types, this is the length of the data after conversion; that is, it is the
11067 size of the type to which the data was converted.

11068 For information about how to determine the address of the length/indicator buffer, see
11069 **Buffer Addresses** on page 217.

11070 7. If the data is truncated during conversion without a loss of significant digits (for example,
11071 the real number 1.234 is truncated when converted to the integer 1) or because the length
11072 of the data buffer is too small (for example, the string ''abcdef'' is placed in a 4-octet
11073 buffer), *SQLFetch*() returns SQLSTATE 01004 (Data truncated) and
11074 SQL_SUCCESS_WITH_INFO. If data is truncated due to the SQL_ATTR_MAX_LENGTH
11075 statement attribute, *SQLFetch*() returns SQL_SUCCESS and does not return SQLSTATE
11076 01004 (Data truncated). If data is truncated during conversion with a loss of significant
11077 digits (for example, if a SQL_INTEGER value greater than 100,000 were converted to a
11078 SQL_C_TINYINT), *SQLFetch*() returns SQLSTATE 22003 (Numeric value out of range) and
11079 SQL_ERROR. (For a multi-row fetch, diagnostics are reported as specified in **Error**
11080 **Handling** on page 313.)

11081 The contents of the bound data buffer and the length/indicator buffer are undefined if
11082 *SQLFetch*() does not return SQL_SUCCESS or SQL_SUCCESS_WITH_INFO.

11083 **Row Status Array**

11084 The row status array is used to return the status of each row in the row-set. The address of this
11085 array is specified with the SQL_ATTR_ROW_STATUS_PTR statement attribute. The array is
11086 allocated by the application and must have as many elements as are specified by the
11087 SQL_ATTR_ROW_ARRAY_SIZE statement attribute. Its values are set by *SQLBulkOperations*(),
11088 *SQLFetch*(), *SQLFetchScroll*(), and *SQLSetPos*(). If the value of the
11089 SQL_ATTR_ROW_STATUS_PTR statement attribute is a null pointer, these functions do not
11090 return the row status.

The contents of the row status array buffer are undefined if *SQLFetch*() does not return SQL_SUCCESS or SQL_SUCCESS_WITH_INFO.

The following values are returned in the row status array.

Row status array value	Description
SQL_ROW_SUCCESS	The row was successfully fetched and has not changed since it was last fetched from this result set.
SQL_ROW_SUCCESS_WITH_INFO	The row was successfully fetched and has not changed since it was last fetched from this result set. However, a warning was returned about the row.
SQL_ROW_ERROR	An error occurred while fetching the row.
SQL_ROW_UPDATED [1, 2, 3]	The row was successfully fetched and has changed since it was last fetched from this result set. If the row is fetched again from this result set, or is refreshed by *SQLSetPos*(), the status changed to the row's new status.
SQL_ROW_DELETED [3]	The row has been deleted since it was last fetched from this result set.
SQL_ROW_ADDED [4]	The row was inserted by *SQLBulkOperations*(). If the row is fetched again from this result set, or is refreshed by *SQLSetPos*(), its status is SQL_ROW_SUCCESS.
SQL_ROW_NOROW	The row-set overlapped the end of the result set and no row was returned that corresponded to this element of the row status array.

1 For keyset, mixed, and dynamic cursors, if a key value is updated, the row of data is considered to have been deleted and a new row added.

2 Some implementations cannot detect updates to data and therefore cannot return this value. To determine whether an implementation can detect updates to refetched rows, an application calls *SQLGetInfo*() with the SQL_ROW_UPDATES option.

3 *SQLFetch*() can return this value only when it is intermixed with calls to *SQLFetchScroll*(). The reason for this is that *SQLFetch*() moves forward through the result set and, when used exclusively, does not refetch any rows. Because no rows are refetched, *SQLFetch*() does not detect changes made to previously fetched rows. However, if *SQLFetchScroll*() positions the cursor before any previously fetched rows and *SQLFetch*() is used to fetch those rows, *SQLFetch*() can detect any changes to those rows.

4 Returned by *SQLBulkOperations*() only. Not set by *SQLFetch*() or *SQLFetchScroll*().

Rows Fetched Buffer

The rows fetched buffer is used to return the number of rows fetched, including those rows for which no data was returned because an error occurred while they were being fetched. In other words, it is the number of rows for which the value in the row status array is not SQL_ROW_NOROW. The address of this buffer is specified with the SQL_ATTR_ROWS_FETCHED_PTR statement attribute. The buffer is allocated by the application. It is set by *SQLFetch*() and *SQLFetchScroll*(). If the value of the SQL_ATTR_ROWS_FETCHED_PTR statement attribute is a null pointer, these functions do not return the number of rows fetched. To determine the number of the current row in the result set, an application can call *SQLGetStmtAttr*() with the SQL_ATTR_ROW_NUMBER attribute.

11135 The contents of the rows fetched buffer are undefined if *SQLFetch*() does not return
11136 SQL_SUCCESS or SQL_SUCCESS_WITH_INFO.

11137 **Error Handling**

11138 Diagnostics can apply to individual rows or to the entire function. For more information about
11139 diagnostic records, see Chapter 15, and *SQLGetDiagField*().

11140 Diagnostics on the Entire Function

11141 If an error applies to the entire function, such as SQLSTATE HYT00 (Timeout expired) or
11142 SQLSTATE 24000 (Invalid cursor state), *SQLFetch*() returns SQL_ERROR and the applicable
11143 SQLSTATE. The contents of the row-set buffers are undefined and the cursor position is
11144 unchanged.

11145 If a warning applies to the entire function, *SQLFetch*() returns SQL_SUCCESS_WITH_INFO and
11146 the applicable SQLSTATE. The status records for warnings that apply to the entire function are
11147 returned before the status records that apply to individual rows.

11148 Diagnostics in Individual Rows

11149 If an error (such as SQLSTATE 22012 (Division by zero)) or a warning (such as SQLSTATE 01004
11150 (Data truncated)) applies to a single row, *SQLFetch*():

11151 • Sets the corresponding element of the row status array to SQL_ROW_ERROR for errors or
11152 SQL_ROW_SUCCESS_WITH_INFO for warnings.

11153 • Adds zero or more status records containing SQLSTATEs for the diagnostic.

11154 • Sets the row and column number fields in the status records. If *SQLFetch*() cannot determine
11155 a row or column number, it sets that number to SQL_ROW_NUMBER_UNKNOWN or
11156 SQL_COLUMN_NUMBER_UNKNOWN respectively. If the status record does not apply to
11157 a particular column, *SQLFetch*() sets the column number to SQL_NO_COLUMN_NUMBER.

11158 *SQLFetch*() continues fetching rows until it has fetched all of the rows in the row-set. It returns
11159 SQL_SUCCESS_WITH_INFO unless an error occurs in every row of the row-set (not counting
11160 rows with status SQL_ROW_NOROW), in which case it returns SQL_ERROR. In particular, if
11161 the row-set size is 1 and an error occurs in that row, *SQLFetch*() returns SQL_ERROR.

11162 *SQLFetch*() returns the status records in row number order. That is, it returns all status records
11163 for unknown rows (if any), then all status records for the first row (if any), then all status records
11164 for the second row (if any), and so on. The status records for each individual row are ordered
11165 according to the normal rules for ordering status records in **Sequence of Status Records** on page
11166 196.

11167 **Descriptors and SQLFetch()**

11168 The following sections describe how *SQLFetch*() interacts with descriptors.

11169 **Argument Mappings**

11170 The implementation does not set any descriptor fields based on the arguments of *SQLFetch*().

11171 **Other Descriptor Fields**

11172 The following descriptor fields are used by *SQLFetch*():

11173 SQL_DESC_ARRAY_SIZE (header field in ARD)
11174 SQL_ATTR_ROW_ARRAY_SIZE statement attribute

11175 SQL_DESC_ARRAY_STATUS_PTR (header field in IRD)
11176 SQL_ATTR_ROW_STATUS_PTR statement attribute

11177 SQL_DESC_BIND_OFFSET_PTR (header field in ARD)
11178 SQL_ATTR_ROW_BIND_OFFSET_PTR statement attribute

11179 SQL_DESC_BIND_TYPE (header field of ARD)
11180 SQL_ATTR_ROW_BIND_TYPE statement attribute

11181 SQL_DESC_COUNT (header field of ARD)
11182 *ColumnNumber* argument of *SQLBindCol*()

11183 SQL_DESC_DATA_PTR (in records of ARD)
11184 *TargetValuePtr* argument of *SQLBindCol*()

11185 SQL_DESC_INDICATOR_PTR (in records of ARD)
11186 *StrLen_or_IndPtr* argument of *SQLBindCol*()

11187 SQL_DESC_OCTET_LENGTH (in records of ARD)
11188 *BufferLength* argument of *SQLBindCol*()

11189 SQL_DESC_OCTET_LENGTH_PTR (in records of ARD)
11190 *StrLen_or_IndPtr* argument of *SQLBindCol*()

11191 SQL_DESC_ROWS_PROCESSED_PTR (header of IRD)
11192 SQL_ATTR_ROWS_FETCHED_PTR statement attribute

11193 SQL_DESC_TYPE (in records of ARD)
11194 *TargetType* argument of *SQLBindCol*()

11195 All descriptor fields can also be set through *SQLSetDescField*().

11196 **Additional Comments**

11197 Applications can bind a single buffer or two separate buffers to be used for length and indicator
11198 values. When an application calls *SQLBindCol*(), the implementation sets the
11199 SQL_DESC_OCTET_LENGTH_PTR and SQL_DESC_INDICATOR_PTR fields of the ARD to the
11200 same address, which is passed in *StrLen_or_IndPtr*. When an application calls *SQLSetDescField*()
11201 or *SQLSetDescRecord*(), it can set these two fields to different addresses. Therefore, *SQLFetch*()
11202 must check these descriptor fields individually to determine where to return length and
11203 indicator values.

11204 If separate buffers are used for the length and indicator values, *SQLFetch*() sets the indicator
11205 buffer to 0 when it returns a length in the length buffer. When the data is NULL, the application
11206 sets the indicator buffer to SQL_NULL_DATA, and the length buffer is undefined. It does not
11207 touch the length buffer when it sets the indicator buffer to a non-zero value.

11208 **SEE ALSO**

	For information about	**See**
11209		
11210	Binding a buffer to a column in a result set	*SQLBindCol*()
11211	Canceling statement processing	*SQLCancel*()
11212	Returning information about a column in a result set	*SQLDescribeCol*()
11213	Executing an SQL statement	*SQLExecDirect*()
11214	Executing a prepared SQL statement	*SQLExecute*()
11215	Fetching a block of data or scrolling through a result set	*SQLFetchScroll*()
11216	Freeing a statement handle	*SQLFreeStmt*()

Fetching part or all of a column of data	*SQLGetData*()
Returning the number of result set columns	*SQLNumResultCols*()
Preparing a statement for execution	*SQLPrepare*()

CHANGE HISTORY

Version 2

Revised generally. See **Alignment with Popular Implementations** on page 2.

NAME

SQLFetchScroll — Fetch the specified row-set of data from the result set and return data for all bound columns.

SYNOPSIS

```
SQLRETURN SQLFetchScroll(
  SQLHSTMT StatementHandle,
  SQLSMALLINT FetchOrientation,
  SQLINTEGER FetchOffset);
```

ARGUMENTS

StatementHandle [Input]
Statement handle.

FetchOrientation [Input]
Type of fetch:

SQL_FETCH_NEXT
SQL_FETCH_PRIOR
SQL_FETCH_FIRST
SQL_FETCH_LAST
SQL_FETCH_ABSOLUTE
SQL_FETCH_RELATIVE
SQL_FETCH_BOOKMARK

For more information, see **Positioning the Cursor** on page 319.

FetchOffset [Input]
Number of the row to fetch. The interpretation of this argument depends on the value of *FetchOrientation* argument. For more information, see **Positioning the Cursor** on page 319.

RETURN VALUE

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NO_DATA, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE.

DIAGNOSTICS

When *SQLFetchScroll*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling *SQLGetDiagRec*() with a *HandleType* of SQL_HANDLE_STMT and a *Handle* of *StatementHandle*. The following SQLSTATE values are commonly returned by *SQLFetchScroll*().

The return code associated with each SQLSTATE value is SQL_ERROR, except that for SQLSTATE values in class 01, the return code is SQL_SUCCESS_WITH_INFO, and except that, if the row-set size is greater than 1 and the operation was applied to at least one row successfully, the return code is SQL_SUCCESS_WITH_INFO.

If an error occurs on a single column, *SQLGetDiagField*() can be called with a *DiagIdentifier* of SQL_DIAG_COLUMN_NUMBER to determine the column the error occurred on; and *SQLGetDiagField*() can be called with a *DiagIdentifier* of SQL_DIAG_ROW_NUMBER to determine the row containing that column.

01000 — General warning
Implementation-defined informational message.

01004 — String data, right truncation
String or binary data returned for a column resulted in the truncation of non-blank character or non-NULL binary data. String values are right truncated.

11268 01S06 — Attempt to fetch before the result set returned the first row-set
11269 The call tried to move the cursor backward before the start of the result set, but less than the
11270 size of one row-set. *SQLFetchScroll*() returns the first row-set in the result set. (Attempts to
11271 move the cursor a full row-set before the start of the result set, or further backward, cause
11272 *SQLFetchScroll*() to return SQL_NO_DATA.)

11273 01S07 — Fractional truncation
11274 The data returned for a column was truncated. For numeric data types, the fractional part of
11275 the number was truncated. For time, timestamp, and interval data types containing a time
11276 component, the fractional portion of the time was truncated.

11277 07006 — Restricted data type attribute violation
11278 A data value of a column in the result set could not be converted to the C data type
11279 specified by *TargetType*in *SQLBindCol*().

11280 Column 0 was bound with a data type of SQL_C_VARBOOKMARK and the
11281 SQL_ATTR_USE_BOOKMARKS statement option was not set to SQL_UB_VARIABLE.

11282 08S01 — Communication link failure
11283 The communication link to the data source failed before the function completed processing.

11284 22001 — String data, right truncation
11285 A bookmark returned for a column was truncated.

11286 22002 — Indicator variable required but not supplied
11287 A null value was fetched into a column whose pointer (the *StrLen_or_IndValue* argument to
11288 *SQLBindCol*() or SQL_DESC_INDICATOR_PTR set by *SQLSetDescField*() or
11289 *SQLSetDescRec*()) was a null pointer.

11290 22003 — Numeric value out of range
11291 Returning the numeric value (as numeric or string) for one or more bound columns would
11292 have caused the whole (as opposed to fractional) part of the number to be truncated.

11293 For more information, see Appendix D.

11294 22007 — Invalid date/time format
11295 A character column in the result set was bound to a date, time, or timestamp C structure,
11296 and a value in the column was, respectively, an invalid date, time, or timestamp.

11297 22012 — Division by zero
11298 A value from an arithmetic expression was returned which resulted in division by zero.

11299 22015 — Interval field overflow
11300 An exact numeric column in the result set was bound to an interval C structure and
11301 returning the data caused a loss of significant digits.

11302 An interval column in the result set was bound to an interval C structure and returning the
11303 data caused a loss of significant digits.

11304 Data in the result set was bound to an interval C structure and there was no representation
11305 of the data in the interval C structure.

11306 22018 — Invalid character value for cast specification
11307 A character column in the result set was bound to a character C buffer and the column
11308 contained a character for which there was no representation in the character set of the
11309 buffer.

11310 A character column in the result set was bound to an approximate numeric C buffer and a
11311 value in the column could not be cast to a valid approximate numeric value.

11312 A character column in the result set was bound to an exact numeric C buffer and a value in
11313 the column could not be cast to a valid exact numeric value.

11314 A character column in the result set was bound to a date/time or interval C buffer and a
11315 value in the column could not be cast to a valid date/time or interval value.

11316 24000 — Invalid cursor state
11317 *StatementHandle* was in an executed state but no result set was associated with
11318 *StatementHandle*.

11319 40001 — Serialization failure
11320 The transaction in which the fetch was executed was terminated to prevent deadlock.

11321 HY000 — General error
11322 An error occurred for which there was no specific SQLSTATE and for which no
11323 implementation-specific SQLSTATE was defined. The error message returned by
11324 *SQLGetDiagRec*() in the **MessageText* buffer describes the error and its cause.

11325 HY001 — Memory allocation error
11326 The implementation failed to allocate memory required to support execution or completion
11327 of the function.

11328 HY008 — Operation canceled
11329 Asynchronous processing was enabled for *StatementHandle*. The function was called and
11330 before it completed execution, *SQLCancel*() was called on *StatementHandle*. The function
11331 was then called again on *StatementHandle*.

11332 The function was called and, before it completed execution, *SQLCancel*() was called on
11333 *StatementHandle* from a different thread in a multithread application.

11334 HY010 — Function sequence error
11335 *StatementHandle* was not in an executed state. The function was called without first calling
11336 *SQLExecDirect*(), *SQLExecute*(), or a catalog function.

11337 An asynchronously executing function (not this one) was called for *StatementHandle* and
11338 was still executing when this function was called.

11339 *SQLBulkOperations*(), *SQLExecDirect*(), *SQLExecute*(), or *SQLSetPos*() was called for
11340 *StatementHandle* and returned SQL_NEED_DATA. This function was called before data was
11341 sent for all data-at-execution parameters or columns.

11342 HY106 — Fetch type out of range
11343 *FetchOrientation* was invalid.

11344 *FetchOrientation* was SQL_FETCH_BOOKMARK, and the SQL_ATTR_USE_BOOKMARKS
11345 statement attribute was set to SQL_UB_OFF.

11346 The value of the SQL_CURSOR_TYPE statement attribute was
11347 SQL_CURSOR_FORWARD_ONLY and *FetchOrientation* was not SQL_FETCH_NEXT.

11348 HY107 — Row value out of range
11349 The value specified with the SQL_ATTR_CURSOR_TYPE statement attribute was
11350 SQL_CURSOR_KEYSET_DRIVEN, but the value specified with the
11351 SQL_ATTR_KEYSET_SIZE statement attribute was greater than 0 and less than the value
11352 specified with the SQL_ATTR_ROW_ARRAY_SIZE statement attribute.

11353 HY111 — Invalid bookmark value
11354 *FetchOrientation* was SQL_FETCH_BOOKMARK and the bookmark pointed to by the value
11355 in the SQL_ATTR_FETCH_BOOKMARK_PTR statement attribute was not valid or was a
11356 null pointer.

11357 HYC00 — Optional feature not implemented
11358 The data source does not support the specified fetch type.

11359 The data source does not support the conversion specified by the combination of *TargetType*
11360 in *SQLBindCol*() and the SQL data type of the corresponding column.

11361 HYT01 — Connection timeout expired
11362 The connection timeout period expired before the data source responded to the request. The
11363 connection timeout period is set through *SQLSetConnectAttr*(),
11364 SQL_ATTR_CONNECTION_TIMEOUT.

11365 IM001 — Function not supported
11366 The function is not supported on the current connection to the data source.

11367 **COMMENTS**

11368 *SQLFetchScroll*() returns a specified row-set from the result set. Row-sets can be specified by
11369 absolute or relative position or by bookmark. *SQLFetchScroll*() can be called only while a result
11370 set exists — that is, after a call that creates a result set and before the cursor over that result set is
11371 closed. If any columns are bound, it returns the data in those columns. If the application has
11372 specified a pointer to a row status array or a buffer in which to return the number of rows
11373 fetched, *SQLFetchScroll*() returns this information as well. Calls to *SQLFetchScroll*() can be mixed
11374 with calls to *SQLFetch*().

11375 **Positioning the Cursor**

11376 When the result set is created, the cursor is positioned before the start of the result set.
11377 *SQLFetchScroll*() positions the cursor based on the values of the *FetchOrientation* and *FetchOffset*
11378 arguments as shown in the following table. The exact rules for determining the start of the new
11379 row-set are shown in the next section.

	FetchOrientation	**Meaning**
11380–11382	SQL_FETCH_NEXT	Return the next row-set. This is equivalent to calling *SQLFetch.*() *SQLFetchScroll*() ignores the value of *FetchOffset*.
11383–11384	SQL_FETCH_PRIOR	Return the prior row-set. *SQLFetchScroll*() ignores the value of *FetchOffset*.
11385–11386	SQL_FETCH_RELATIVE	Return the row-set *FetchOffset* from the start of the current row-set.
11387	SQL_FETCH_ABSOLUTE	Return the row-set starting at row *FetchOffset*.
11388–11389	SQL_FETCH_FIRST	Return the first row-set in the result set. *SQLFetchScroll*() ignores the value of *FetchOffset*.
11390–11391	SQL_FETCH_LAST	Return the last complete row-set in the result set. *SQLFetchScroll*() ignores the value of *FetchOffset*.
11392–11394	SQL_FETCH_BOOKMARK	Return the row-set *FetchOffset* rows from the bookmark specified by the SQL_ATTR_FETCH_BOOKMARK_PTR statement attribute.

11395 It is implementation-defined which fetch orientations are supported. An application can
11396 determine which fetch orientations are supported in conjunction with various types of cursor by
11397 calling *SQLGetInfo*() as described in **Detecting Cursor Capabilities with SQLGetInfo()** on page
11398 402. Furthermore, if the cursor is forward-only and *FetchOrientation* is not SQL_FETCH_NEXT,
11399 *SQLFetchScroll*() returns SQLSTATEHY106 (Fetch type out of range).

The SQL_ATTR_ROW_ARRAY_SIZE statement attribute specifies the number of rows in the row-set. If the row-set being fetched by *SQLFetchScroll*() overlaps the end of the result set, *SQLFetchScroll*() returns a partial row-set. That is, if S+R–1 is greater than L, where S is the starting row of the row-set being fetched, R is the row-set size, and L is the last row in the result set, then only the first L–S+1 rows of the row-set are valid. The remaining rows are empty and have a status of SQL_ROW_NOROW.

After *SQLFetchScroll*() returns, the cursor is positioned on the first row of the result set.

Cursor Positioning Rules

The following sections describe the exact rules for each value of *FetchOrientation*. These rules use the following notation:

Notation	Meaning
Before start	The cursor is positioned before the start of the result set. If the first row of the new row-set is before the start of the result set, *SQLFetchScroll*() returns SQL_NO_DATA.
After end	The cursor is positioned after the end of the result set. If the first row of the new row-set is after the end of the result set, *SQLFetchScroll*() returns SQL_NO_DATA.
CurrRowsetStart	The number of the first row in the current row-set.
LastResultRow	The number of the last row in the result set.
RowsetSize	The row-set size.
FetchOffset	The value of the *FetchOffset* argument.
BookmarkRow	The row corresponding to the bookmark specified by the SQL_ATTR_FETCH_BOOKMARK_PTR statement attribute.

SQL_FETCH_NEXT

The following rules apply:

Condition	First row of new row-set
Before start	1
CurrRowsetStart – *RowsetSize* ≤ *LastResultRow* [1]	*CurrRowsetStart* + *RowsetSize* [1]
CurrRowsetStart – *RowsetSize* > *LastResultRow* [1]	*After end*
After end	*After end*

[1] If the row-set size is changed since the previous call to fetch rows, this is the row-set size that was used with the previous call.

SQL_FETCH_PRIOR

The following rules apply:

Condition	**First row of new row-set**
Before start	*Before start*
CurrRowsetStart = 1	*Before start*
1 < *CurrRowsetStart* ≤ *RowsetSize* [2]	1 [1]
CurrRowsetStart > *RowsetSize* [2]	*CurrRowsetStart* – *RowsetSize* [2]
After end AND *LastResultRow* < *RowsetSize* [2]	1 [1]
After end AND *LastResultRow* ≥ *RowsetSize* [2]	*LastResultRow* – *RowsetSize* [2] + 1

[1] *SQLFetchScroll*() returns SQLSTATE 01S06 (Attempt to fetch before the result set returned the first row-set) and SQL_SUCCESS_WITH_INFO.

[2] If the row-set size has been changed since the previous call to fetch rows, this is the new row-set size.

SQL_FETCH_RELATIVE

The following rules apply:

Condition	**First row of new row-set**
(*Before start* AND *FetchOffset* > 0) OR (*After end* AND *FetchOffset* < 0)	— [1]
BeforeStart AND *FetchOffset* ≤ 0	*Before start*
CurrRowsetStart = 1 AND *FetchOffset* < 0	*Before start*
CurrRowsetStart + *FetchOffset* < 1 AND ∣ *FetchOffset* ∣ > *RowsetSize* [3]	*Before start*
CurrRowsetStart + *FetchOffset* < 1 AND ∣ *FetchOffset* ∣ ≤ *RowsetSize* [3]	1 [2]
1 ≤ *CurrRowsetStart* + *FetchOffset* ≤ *LastResultRow*	*CurrRowsetStart* + *FetchOffset*
CurrRowsetStart + *FetchOffset* > *LastResultRow*	*After end*
After end AND *FetchOffset* ≥ 0	*After end*

[1] *SQLFetchScroll*() returns the same row-set as if it was called with *FetchOrientation* set to SQL_FETCH_ABSOLUTE. For more information, see the SQL_FETCH_ABSOLUTE table below.

[2] *SQLFetchScroll*() returns SQLSTATE 01S06 (Attempt to fetch before the result set returned the first row-set) and SQL_SUCCESS_WITH_INFO.

[3] If the row-set size has been changed since the previous call to fetch rows, this is the new row-set size.

SQL_FETCH_ABSOLUTE

The following rules apply:

Condition	First row of new row-set
$FetchOffset < 0$ AND $\lvert FetchOffset \rvert \leq LastResultRow$	$LastResultRow + FetchOffset + 1$
$FetchOffset < 0$ AND $\lvert FetchOffset \rvert > LastResultRow$ AND $\lvert FetchOffset \rvert > RowsetSize$ [2]	*Before start*
$FetchOffset < 0$ AND $\lvert FetchOffset \rvert > LastResultRow$ AND $\lvert FetchOffset \leq RowsetSize$ [2]	1 [1]
$FetchOffset = 0$	*Before start*
$1 \leq FetchOffset \leq LastResultRow$	*FetchOffset*
$FetchOffset > LastResultRow$	*After end*

[1] *SQLFetchScroll() returns SQLSTATE01S06 (Attempt to fetch before the result set returned the first row-set) and SQL_SUCCESS_WITH_INFO.*

[2] *If the row-set size has been changed since the previous call to fetch rows, this is the new row-set size.*

An absolute fetch performed against a dynamic cursor may not provide the anticipated result because row positions in a dynamic cursor are undetermined. Such an operation is equivalent to a fetch first followed by a fetch relative; it is not an atomic operation, as an absolute fetch on a static cursor is.

SQL_FETCH_FIRST

The following rules apply:

Condition	First row of new row-set
Any	*1*

SQL_FETCH_LAST

The following rules apply:

Condition	First row of new row-set
$RowsetSize$ [1] $\leq LastResultRow$	$LastResultRow - RowsetSize$ [1] $+ 1$
$RowsetSize$ [1] $> LastResultRow$	*1*

[1] *If the row-set size has been changed since the previous call to fetch rows, this is the new row-set size.*

SQL_FETCH_BOOKMARK

The following rules apply:

Condition	First row of new row-set
$BookmarkRow + FetchOffset < 1$	*Before start*
$1 \leq BookmarkRow + Fetchoffset \leq LastResultRow$	$BookmarkRow + FetchOffset$
$BookmarkRow + FetchOffset > LastResultRow$	*After end*

For information about bookmarks, see Section 11.2.4 on page 154.

Effect of Deleted, Added, and Error Rows on Cursor Movement

Static and keyset-driven cursors sometimes detect rows added to the result set and remove rows deleted from the result set. An application determines the effect for various types of cursor by calling SQLGetInfo() as described in **Detecting Cursor Capabilities with SQLGetInfo()** *on page 402. For data sources that can detect deleted rows and remove them, the following paragraphs describe the effects of this behavior. For data sources that can detect deleted rows but cannot remove them, deletions have no effect on cursor movements, and the following paragraphs do not apply.*

If the cursor detects rows added to the result set or removes rows deleted from the result set, it appears as if it detects these changes only when it fetches data. This includes the case when SQLFetchScroll() is called with FetchOrientation set to SQL_FETCH_RELATIVE and FetchOffset set to 0 to refetch the same row-set but does not include the case when SQLSetPos() is called with fOption set to SQL_REFRESH. In the latter case, the data in the row-set buffers is refreshed, but not refetched, and deleted rows are not removed from the result set. Thus, when a row is deleted from or inserted into the current row-set, the cursor does not modify the row-set buffers. Instead, it detects the change when it fetches any row-set that previously included the deleted row or now includes the inserted row.

For example:

```
// Fetch the next row-set
SQLFetchScroll(hstmt, SQL_FETCH_NEXT, 0);

// Delete third row of the row-set. Does not modify the row-set buffers.
SQLSetPos(hstmt, 3, SQL_DELETE, SQL_LOCK_NO_CHANGE);

// The third row has a status of SQL_ROW_DELETED after this call.
SQLSetPos(hstmt, 3, SQL_REFRESH, SQL_LOCK_NO_CHANGE);

// Refetch the same row-set. The third row is removed, replaced by what
// was previously the fourth row.
SQLFetchScroll(hstmt, SQL_FETCH_RELATIVE, 0);
```

When *SQLFetchScroll*() returns a new row-set that has a position relative to the current row-set — that is, *FetchOrientation* is SQL_FETCH_NEXT, SQL_FETCH_PRIOR, or SQL_FETCH_RELATIVE — it does not include changes to the current row-set when calculating the starting position of the new row-set. However, it does include changes outside the current row-set if it is capable of detecting them. Furthermore, when *SQLFetchScroll*() returns a new row-set that has a position independent of the current row-set — that is, *FetchOrientation* is SQL_FETCH_FIRST, SQL_FETCH_LAST, SQL_FETCH_ABSOLUTE, or SQL_FETCH_BOOKMARK — it includes all changes it is capable of detecting, even if they are in the current row-set.

When determining whether newly added rows are inside or outside the current row-set, a partial row-set is considered to end at the last valid row; that is, the last row for which the row status is not SQL_ROW_NOROW. For example, suppose the cursor is capable of detecting newly added rows, the current row-set is a partial row-set, the application adds new rows, and the cursors adds these rows to the end of the result set. If the application calls *SQLFetchScroll*() with FetchOrientation set to SQL_FETCH_NEXT, *SQLFetchScroll*() returns the row-set starting with the first newly added row.

For example, suppose the row-set size is 10, the current row-set comprises rows 21 to 30, the cursor removes rows deleted from the result set, and the cursor detects rows added to the result set. The following table shows the rows that *SQLFetchScroll*() returns in various situations:

Change	Fetch Type	FetchOffset	New Row-set [1]

Delete row 21	NEXT	0	31 to 40
Delete row 31	NEXT	0	32 to 41
Insert row between rows 21 and 22	NEXT	0	31 to 40
Insert row between rows 30 and 31	NEXT	0	Inserted row, 31 to 39
Delete row 21	PRIOR	0	11 to 20
Delete row 20	PRIOR	0	10 to 19
Insert row between rows 21 and 22	PRIOR	0	11 to 20
Insert row between rows 20 and 21	PRIOR	0	12 to 20, inserted row
Delete row 21	RELATIVE	0	22 to 31 [2]
Delete row 21	RELATIVE	1	22 to 31
Insert row between rows 21 and 22	RELATIVE	0	21, inserted row, 22 to 29
Insert row between rows 21 and 22	RELATIVE	1	22 to 31
Delete row 21	ABSOLUTE	21	22 to 31 [2]
Delete row 22	ABSOLUTE	21	21, 23 to 31
Insert row between rows 21 and 22	ABSOLUTE	22	Inserted row, 22 to 29

[1] This column uses the row numbers before any rows were inserted or deleted.

[2] In this case, the cursor attempts to return rows starting with row 21. Because row 21 has been deleted, the first row it returns is row 22.

Error rows (that is, rows with a status of SQL_ROW_ERROR) do not affect cursor movement. For example, if the current row-set starts with row 11 and the status of row 11 is SQL_ROW_ERROR, calling *SQLFetchScroll*() with *FetchOrientation* set to SQL_FETCH_RELATIVE and *FetchOffset* set to 5 returns the row-set starting with row 16, just as it would if the status for row 11 was SQL_SUCCESS.

Returning Data in Bound Columns

SQLFetchScroll() returns data in bound columns in the same way as *SQLFetch*(). For more information, see **Returning Data in Bound Columns** on page 310.

If no columns are bound, *SQLFetchScroll*() does not return data but does move the cursor to the specified position. It is implementation-defined whether data can be retrieved from unbound columns with *SQLGetData*(). An application determines whether it can return data from unbound columns with *SQLGetData*() only if *SQLGetInfo*() returns the SQL_GD_BLOCK bit for the SQL_GETDATA_EXTENSIONS option.

Buffer Addresses

SQLFetchScroll() uses the same formula to determine the address of data and length/indicator buffers as *SQLFetch*(). For more information, see **Buffer Addresses** on page 217.

Row Status Array

SQLFetchScroll() sets values in the row status array in the same manner as *SQLFetch*(). For more information, see Section 10.4.3 on page 134.

Rows Fetched Buffer

SQLFetchScroll() returns the number of rows fetched in the rows fetched buffer in the same manner as *SQLFetch*(). For more information, see **Rows Fetched Buffer** on page 312.

11585 **SQLFetchScroll() and Optimistic Concurrency**

11586 If a cursor uses optimistic concurrency — that is, the SQL_ATTR_CONCURRENCY statement
11587 attribute has a value of SQL_CONCUR_VALUES or SQL_CONCUR_ROWVER —
11588 *SQLFetchScroll*() updates the optimistic concurrency values used by the data source to detect
11589 whether a row has changed. This happens whenever *SQLFetchScroll*() fetches a new row-set,
11590 including when it it refetches the current row-set (it is called with FetchOrientation set to
11591 SQL_FETCH_RELATIVE and FetchOffset set to 0). **Descriptors and SQLFetchScroll()**

11592 *SQLFetchScroll*() interacts with descriptors in the same manner as *SQLFetch*(). For more
11593 information, see **Descriptors and SQLFetch()** on page 313.

11594 **SEE ALSO**

11595	For information about	See
11596	Binding a buffer to a column in a result set	*SQLBindCol*()
11597	Performing bulk insert or update operations	*SQLBulkOperations*()
11598	Canceling statement processing	*SQLCancel*()
11599	Returning information about a column in a result set	*SQLDescribeCol*()
11600	Executing an SQL statement	*SQLExecDirect*()
11601	Executing a prepared SQL statement	*SQLExecute*()
11602 11603	Fetching a single row or a block of data in a forward-only direction	*SQLFetch*()
11604	Returning the number of result set columns	*SQLNumResultCols*()
11605 11606	Positioning the cursor, refreshing data in the row-set, or updating or deleting data in the result set	*SQLSetPos*()
11607	Setting a statement attribute	*SQLSetStmtAttr*()

11608 **CHANGE HISTORY**

11609 **Version 2**

11610 Revised generally. See **Alignment with Popular Implementations** on page 2.

11611 **NAME**

11612 SQLForeignKeys — Return a list of foreign keys for a specified table.

11613 **SYNOPSIS**

```
11614 SQLRETURN SQLForeignKeys(
11615   SQLHSTMT StatementHandle,
11616   SQLCHAR * PKCatalogName,
11617   SQLSMALLINT NameLength1,
11618   SQLCHAR * PKSchemaName,
11619   SQLSMALLINT NameLength2,
11620   SQLCHAR * PKTableName,
11621   SQLSMALLINT NameLength3,
11622   SQLCHAR * FKCatalogName,
11623   SQLSMALLINT NameLength4,
11624   SQLCHAR * FKSchemaName,
11625   SQLSMALLINT NameLength5,
11626   SQLCHAR * FKTableName,
11627   SQLSMALLINT NameLength6);
```

11628 **ARGUMENTS**

11629 *StatementHandle* [Input]
11630 Statement handle.

11631 *PKCatalogName* [Input]
11632 Primary key table catalog name. If a data source supports catalogs, an empty string denotes
11633 those tables that do not have catalogs.

11634 If the SQL_ATTR_METADATA_ID statement attribute is SQL_TRUE, this argument is
11635 interpreted as specified in **Identifier (ID) Arguments** on page 72. If it is SQL_FALSE, this
11636 argument is interpreted as specified in **Ordinary Arguments (OA)** on page 71.

11637 *NameLength1* [Input]
11638 Length of **PKCatalogName*, in octets.

11639 *PKSchemaName* [Input]
11640 Primary key table schema name. If a data source supports schemas, an empty string denotes
11641 those tables that do not have schemas.

11642 If the SQL_ATTR_METADATA_ID statement attribute is SQL_TRUE, this argument is
11643 interpreted as specified in **Identifier (ID) Arguments** on page 72. If it is SQL_FALSE, this
11644 argument is interpreted as specified in **Ordinary Arguments (OA)** on page 71.

11645 *NameLength2* [Input]
11646 Length of **PKSchemaName*, in octets.

11647 *PKTableName* [Input]
11648 Primary key table name.

11649 If the SQL_ATTR_METADATA_ID statement attribute is SQL_TRUE, this argument is
11650 interpreted as specified in **Identifier (ID) Arguments** on page 72. If it is SQL_FALSE, this
11651 argument is interpreted as specified in **Ordinary Arguments (OA)** on page 71.

11652 *NameLength3* [Input]
11653 Length of **PKTableName*.

11654 *FKCatalogName* [Input]
11655 Foreign key table catalog name. If a data source supports catalogs, an empty string denotes
11656 those tables that do not have catalog.

11657 If the SQL_ATTR_METADATA_ID statement attribute is SQL_TRUE, this argument is
11658 interpreted as specified in **Identifier (ID) Arguments** on page 72. If it is SQL_FALSE, this
11659 argument is interpreted as specified in **Ordinary Arguments (OA)** on page 71.

11660 *NameLength4* [Input]
11661 Length of **FKCatalogName*.

11662 *FKSchemaName* [Input]
11663 Foreign key table schema name. If a data source supports schemas, an empty string denotes
11664 those tables that do not have schemas.

11665 If the SQL_ATTR_METADATA_ID statement attribute is SQL_TRUE, this argument is
11666 interpreted as specified in **Identifier (ID) Arguments** on page 72. If it is SQL_FALSE, this
11667 argument is interpreted as specified in **Ordinary Arguments (OA)** on page 71.

11668 *NameLength5* [Input]
11669 Length of **FKSchemaName*.

11670 *FKTableName* [Input]
11671 Foreign key table name.

11672 If the SQL_ATTR_METADATA_ID statement attribute is SQL_TRUE, this argument is
11673 interpreted as specified in **Identifier (ID) Arguments** on page 72. If it is SQL_FALSE, this
11674 argument is interpreted as specified in **Ordinary Arguments (OA)** on page 71.

11675 *NameLength6* [Input]
11676 Length of **FKTableName*.

11677 **RETURN VALUE**

11678 SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or
11679 SQL_INVALID_HANDLE.

11680 **DIAGNOSTICS**

11681 When *SQLForeignKeys*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated
11682 SQLSTATE value can be obtained by calling *SQLGetDiagRec*() with a *HandleType* of
11683 SQL_HANDLE_STMT and a *Handle* of *StatementHandle*. The following SQLSTATE values are
11684 commonly returned by *SQLForeignKeys*(). The return code associated with each SQLSTATE
11685 value is SQL_ERROR, except that for SQLSTATE values in class 01, the return code is
11686 SQL_SUCCESS_WITH_INFO.

11687 01000 — General warning
11688 Implementation-defined informational message.

11689 08S01 — Communication link failure
11690 The communication link to the data source failed before the function completed processing.

11691 24000 — Invalid cursor state
11692 A cursor was open on *StatementHandle*.

11693 HY000 — General error
11694 An error occurred for which there was no specific SQLSTATE and for which no
11695 implementation-specific SQLSTATE was defined. The error message returned by
11696 *SQLGetDiagRec*() in the **MessageText* buffer describes the error and its cause.

11697 HY001 — Memory allocation error
11698 The implementation failed to allocate memory required to support execution or completion
11699 of the function.

11700 HY008 — Operation canceled
11701 Asynchronous processing was enabled for *StatementHandle*. The function was called and
11702 before it completed execution, *SQLCancel*() was called on *StatementHandle*. The function

11703 was then called again on *StatementHandle*.

11704 The function was called and, before it completed execution, *SQLCancel*() was called on
11705 *StatementHandle* from a different thread in a multithread application.

11706 HY009 — Invalid use of null pointer
11707 *PKTableName* and *FKTableName* were both null pointers.

11708 The SQL_ATTR_METADATA_ID statement attribute was set to SQL_TRUE, *FKCatalogName*
11709 or *PKCatalogName* was a null pointer, and the SQL_CATALOG_NAME option of
11710 *SQLGetInfo*() returns that catalog names are supported.

11711 The SQL_ATTR_METADATA_ID statement attribute was set to SQL_TRUE, and
11712 *FKSchemaName*, *PKSchemaName*, *FKTableName*, or *PKTableName* was a null pointer.

11713 HY010 — Function sequence error
11714 An asynchronously executing function (not this one) was called for *StatementHandle* and
11715 was still executing when this function was called.

11716 *SQLBulkOperations*(), *SQLExecDirect*(), *SQLExecute*(), or *SQLSetPos*() was called for
11717 *StatementHandle* and returned SQL_NEED_DATA. This function was called before data was
11718 sent for all data-at-execution parameters or columns.

11719 HY090 — Invalid string or buffer length
11720 The value of one of the name length arguments was less than 0, but not equal to SQL_NTS.

11721 The value of one of the name length arguments exceeded the maximum length value for the
11722 corresponding name (see ‘‘Comments’’).

11723 HYC00 — Optional feature not implemented
11724 A catalog name was specified and the implementation does not support catalogs.

11725 A schema name was specified and the implementation does not support schemas.

11726 The data source does not support the combination of the current settings of the
11727 SQL_ATTR_CONCURRENCY and SQL_ATTR_CURSOR_TYPE statement attributes.

11728 The SQL_ATTR_USE_BOOKMARKS statement attribute was set to SQL_UB_VARIABLE,
11729 and the SQL_ATTR_CURSOR_TYPE statement attribute was set to a cursor type for which
11730 the data source does not support bookmarks.

11731 HYT00 — Timeout expired
11732 The query timeout period expired before the data source returned the result set. The
11733 timeout period is set through *SQLSetStmtAttr*(), SQL_ATTR_QUERY_TIMEOUT.

11734 HYT01 — Connection timeout expired
11735 The connection timeout period expired before the data source responded to the request. The
11736 connection timeout period is set through *SQLSetConnectAttr*(),
11737 SQL_ATTR_CONNECTION_TIMEOUT.

11738 IM001 — Function not supported
11739 The function is not supported on the current connection to the data source.

11740 **COMMENTS**

11741 *SQLForeignKeys*() can return:

11742 • A list of foreign keys in the specified table (columns in the specified table that refer to
11743 primary keys in other tables).

11744 • A list of foreign keys in other tables that refer to the primary key in the specified table.

11745 The implementation returns each list as a result set on *StatementHandle*.

For XSQL implementations that do not implement referential integrity constraints, *SQLForeignKeys*() should not be implemented, and *SQLFunctions*() should indicate that *SQLForeignKeys*() is not present. On implementations that span data sources some of which do not implement referential integrity constraints, this behavior should depend on the data source. Implementing *SQLForeignKeys*() and returning a result set with no rows is unadvisable because it misleads the application.

If **PKTableName* contains a table name, *SQLForeignKeys*() returns a result set containing the primary key of the specified table and all of the foreign keys that refer to it.

If **FKTableName* contains a table name, *SQLForeignKeys*() returns a result set containing all of the foreign keys in the specified table and the primary keys (in other tables) to which they refer.

If both **PKTableName* and **FKTableName* contain table names, *SQLForeignKeys*() returns the foreign keys in the table specified in **FKTableName* that refer to the primary key of the table specified in **PKTableName*. This should be one key at most.

If the foreign keys associated with a primary key are requested, the result set is ordered by FKTABLE_CAT, FKTABLE_SCHEM , FKTABLE_NAME, and KEY_SEQ. If the primary keys associated with a foreign key are requested, the result set is ordered by PKTABLE_CAT, PKTABLE_SCHEM, PKTABLE_NAME, and KEY_SEQ. The following table lists the columns in the result set.

The lengths of VARCHAR columns shown in the table are maximums; the actual lengths depend on the data source. To determine the actual lengths of the PKTABLE_CAT or FKTABLE_CAT, PKTABLE_SCHEM or FKTABLE_SCHEM, PKTABLE_NAME or FKTABLE_NAME, and PKCOLUMN_NAME or FKCOLUMN_NAME columns, an application can call *SQLGetInfo*() with the SQL_MAX_CATALOG_NAME_LEN, SQL_MAX_SCHEMA_NAME_LEN, SQL_MAX_TABLE_NAME_LEN, and SQL_MAX_COLUMN_NAME_LEN options.

The following table lists the columns in the result set. Additional columns beyond column 17 (REMARKS) can be defined by the implementation. An application should gain access to implementation-defined columns by counting down from the end of the result set rather than by specifying an explicit ordinal position; see Section 7.3 on page 68.

Column name	Col. No.	Data type	Comments
PKTABLE_CAT	1	Varchar	Primary key table catalog name; NULL if not applicable to the data source. If a data source supports catalogs, it returns an empty string for those tables that do not have catalogs.
PKTABLE_SCHEM	2	Varchar	Primary key table schema name; NULL if not applicable to the data source. If a data source supports schemas, it returns an empty string for those tables that do not have schemas.

PKTABLE_NAME	3	Varchar not NULL	Primary key table identifier.
PKCOLUMN_NAME	4	Varchar not NULL	Primary key column identifier; an empty string for an unnamed column.
FKTABLE_CAT	5	Varchar	Foreign key table catalog name; NULL if not applicable to the data source. If a data source supports catalogs, it returns an empty string for those tables that do not have catalogs.
FKTABLE_SCHEM	6	Varchar	Foreign key table schema name; NULL if not applicable to the data source. If a data source supports schemas, it returns an empty string for those tables that do not have schemas.
FKTABLE_NAME	7	Varchar not NULL	Foreign key table identifier.
FKCOLUMN_NAME	8	Varchar not NULL	Foreign key column identifier; an empty string for an unnamed column.
KEY_SEQ	9	Smallint not NULL	Column sequence number in key (starting with 1).
UPDATE_RULE	10	Smallint	The action to be applied to the foreign key when the SQL operation is UPDATE. The valid values are set out below this table.
DELETE_RULE	11	Smallint	The action to be applied to the foreign key when the SQL operation is DELETE. The valid values are set out below this table.
FK_NAME	12	Varchar	Foreign key identifier. NULL if not applicable to the data source.
PK_NAME	13	Varchar	Primary key identifier. NULL if not applicable to the data source.
DEFERRABILITY	14	Smallint	SQL_INITIALLY_DEFERRED SQL_INITIALLY_IMMEDIATE SQL_NOT_DEFERRABLE

Valid Values for UPDATE_RULE Column

The UPDATE_RULE column of the result set can have any of the following values (The referenced table is the table that has the primary key; the referencing table is the table that has the foreign key).

SQL_CASCADE
When the primary key of the referenced table is updated, the foreign key of the referencing table is also updated.

SQL_NO_ACTION
If an update of the primary key of the referenced table would cause a ''dangling reference'' in the referencing table (that is, rows in the referencing table would have no counterparts in the referenced table), then the update is rejected. If an update of the foreign key of the

11826 referencing table would introduce a value that does not exist as a value of the primary key
11827 of the referenced table, then the update is rejected.

11828 SQL_SET_NULL
11829 When one or more rows in the referenced table are updated such that one or more
11830 components of the primary key are changed, the components of the foreign key in the
11831 referencing table that correspond to the changed components of the primary key are set to
11832 NULL in all matching rows of the referencing table.

11833 SQL_SET_DEFAULT
11834 When one or more rows in the referenced table are updated such that one or more
11835 components of the primary key are changed, the components of the foreign key in the
11836 referencing table that correspond to the changed components of the primary key are set to
11837 the applicable default values in all matching rows of the referencing table.

11838 NULL
11839 If not applicable to the data source.

11840 **Valid Values for DELETE_RULE Column**

11841 The DELETE_RULE column of the result set can have any of the following values (The
11842 referenced table is the table that has the primary key; the referencing table is the table that has
11843 the foreign key):

11844 SQL_CASCADE
11845 When a row in the referenced table is deleted, all the matching rows in the referencing
11846 tables are also deleted.

11847 SQL_NO_ACTION
11848 If a delete of a row in the referenced table would cause a ‘‘dangling reference’’ in the
11849 referencing table (that is, rows in the referencing table would have no counterparts in the
11850 referenced table), then the update is rejected.

11851 SQL_SET_NULL
11852 When one or more rows in the referenced table are deleted, each component of the foreign
11853 key of the referencing table is set to NULL in all matching rows of the referencing table.

11854 SQL_SET_DEFAULT
11855 When one or more rows in the referenced table are deleted, each component of the foreign
11856 key of the referencing table is set to the applicable default in all matching rows of the
11857 referencing table.

11858 NULL
11859 If not applicable to the data source.

11860 **SEE ALSO**

	For information about	See
11861		
11862	Overview of catalog functions	Chapter 7
11863	Binding a buffer to a column in a result set	*SQLBindCol*()
11864	Canceling statement processing	*SQLCancel*()
11865	Fetching a block of data or scrolling through a result set	*SQLFetchScroll*()
11866 11867	Fetching a single row or a block of data in a forward-only direction	*SQLFetch*()

11868 Returning the columns of a primary key *SQLPrimaryKeys*()
11869 Returning table statistics and indexes *SQLStatistics*()

11870 **CHANGE HISTORY**

11871 **Version 2**
11872 Function added in this version.

11873 **NAME**

11874 SQLFreeHandle — Free resources associated with a specific handle.

11875 **SYNOPSIS**

```
11876 SQLRETURN SQLFreeHandle(
11877   SQLSMALLINT HandleType,
11878   SQLHANDLE Handle);
```

11879 **ARGUMENTS**

11880 *HandleType* [Input]

11881 The type of handle to be freed by *SQLFreeHandle*(). Must be one of the following values:

11882 SQL_HANDLE_ENV
11883 SQL_HANDLE_DBC
11884 SQL_HANDLE_STMT
11885 SQL_HANDLE_DESC

11886 If *HandleType* is not one of the above values, *SQLFreeHandle*() returns
11887 SQL_INVALID_HANDLE.

11888 *Handle* [Input]

11889 The handle to be freed.

11890 **RETURN VALUE**

11891 SQL_SUCCESS, SQL_ERROR, or SQL_INVALID_HANDLE.

11892 If *SQLFreeHandle*() returns SQL_ERROR, the handle is still valid.

11893 **DIAGNOSTICS**

11894 When *SQLFreeHandle*() returns SQL_ERROR, an associated SQLSTATE value may be obtained
11895 from the diagnostic data structure for the handle that *SQLFreeHandle*() attempted to free, but
11896 could not. The following table lists the SQLSTATE values commonly returned by
11897 *SQLFreeHandle*(). The return code associated with each SQLSTATE value is SQL_ERROR, unless
11898 noted otherwise.

11899 HY000 — General error

11900 An error occurred for which there was no specific SQLSTATE and for which no
11901 implementation-specific SQLSTATE was defined. The error message returned by
11902 *SQLGetDiagRec*() in the **MessageText* buffer describes the error and its cause.

11903 HY001 — Memory allocation error

11904 The implementation failed to allocate memory required to support execution or completion
11905 of the function.

11906 HY010 — Function sequence error

11907 *HandleType* was SQL_HANDLE_ENV, and at least one connection was in an allocated or
11908 connected state. *SQLDisconnect*() and *SQLFreeHandle*() with a *HandleType* of
11909 SQL_HANDLE_DBC must be called for each connection before calling *SQLFreeHandle*()
11910 with a *HandleType* of SQL_HANDLE_ENV.

11911 *HandleType* was SQL_HANDLE_DBC, and the function was called before calling
11912 *SQLDisconnect*() for the connection.

11913 *HandleType* was SQL_HANDLE_STMT; an asynchronously executing function was called on
11914 the statement handle; and the function was still executing when this function was called.

11915 *HandleType* was SQL_HANDLE_STMT; *SQLBulkOperations*(), *SQLExecDirect*(),
11916 *SQLExecute*(), or *SQLSetPos*() was called with the statement handle, and returned
11917 SQL_NEED_DATA. This function was called before data was sent for all data-at-execution
11918 parameters or columns.

11919 All subsidiary handles and other resources were not released before *SQLFreeHandle*() was
11920 called.

11921 HY013 — Memory management error
11922 *HandleType* was SQL_HANDLE_STMT or SQL_HANDLE_DESC, and the function call
11923 could not be processed because the underlying memory objects could not be accessed,
11924 possibly because of low memory conditions.

11925 HY017 — Invalid use of an automatically allocated descriptor handle.
11926 *Handle* was set to the handle for an automatically-allocated descriptor or an implementation •
11927 descriptor.

11928 HYT01 — Connection timeout expired
11929 The connection timeout period expired before the data source responded to the request. The
11930 connection timeout period is set through *SQLSetConnectAttr*(),
11931 SQL_ATTR_CONNECTION_TIMEOUT.

11932 IM001 — Function not supported
11933 The function is not supported on the current connection to the data source.

11934 **COMMENTS**

11935 *SQLFreeHandle*() is used to free handles for environments, connections, statements, and
11936 descriptors.

11937 **Freeing an Environment Handle**

11938 Before calling *SQLFreeHandle*() with a *HandleType* of SQL_HANDLE_ENV, an application must
11939 call *SQLFreeHandle*() with a *HandleType* of SQL_HANDLE_DBC for all connections allocated
11940 under the environment. Otherwise, the call to *SQLFreeHandle*() returns SQL_ERROR and the
11941 environment and any active connection remains valid.

11942 **Freeing a Connection Handle**

11943 Prior to calling *SQLFreeHandle*() with *HandleType* of SQL_HANDLE_DBC, an application must
11944 call *SQLDisconnect*() for the connection. Otherwise, the call to *SQLFreeHandle*() returns
11945 SQL_ERROR and the connection remains valid.

11946 **Freeing a Statement Handle**

11947 A call to *SQLFreeHandle*() with *HandleType* of SQL_HANDLE_STMT frees all resources that were
11948 allocated by a call to *SQLAllocHandle*() with *HandleType* of SQL_HANDLE_STMT. Any pending
11949 results of the statement are deleted and any result sets are discarded. Freeing a statement handle
11950 also frees all the automatically-generated descriptors associated with that handle.

11951 (A call to *SQLDisconnect*() also drops any statements and descriptors open on the connection.)

11952 **Freeing a Descriptor Handle**

11953 A call to *SQLFreeHandle*() with *HandleType* of SQL_HANDLE_DESC frees the descriptor handle
11954 in *Handle*. The call does not release any memory allocated by the application referenced by a
11955 pointer field (including SQL_DESC_DATA_PTR, SQL_DESC_INDICATOR_PTR, and
11956 SQL_DESC_OCTET_LENGTH_PTR) of any descriptor record of *Handle*. The memory allocated
11957 for fields that are not pointer fields is freed when the handle is freed. When an explicitly-
11958 allocated descriptor handle is freed, all statements that the freed handle had been associated
11959 with revert to their automatically-allocated descriptor handle.

11960 (A call to *SQLDisconnect*() also drops any statements and descriptors open on the connection.)

11961 **SEE ALSO**

For information about	See
Allocating a handle	*SQLAllocHandle*()
Canceling statement processing	*SQLCancel*()
Setting a cursor name	*SQLSetCursorName*()

CHANGE HISTORY

Version 2

Revised generally. See **Alignment with Popular Implementations** on page 2.

11969 **NAME**

11970 SQLFreeStmt — Stop processing associated with a specific statement, close any open cursors
11971 associated with the statement, or discard pending results.

11972 **SYNOPSIS**

```
11973 SQLRETURN SQLFreeStmt(
11974   SQLHSTMT StatementHandle,
11975   SQLUSMALLINT Option);
```

11976 **ARGUMENTS**

11977 *StatementHandle* [Input]
11978 Statement handle

11979 *Option* [Input]
11980 One of the following options:

11981 SQL_CLOSE
11982 Close any cursor associated with *StatementHandle* and discard all pending results. The
11983 application can reopen this cursor later by executing a SELECT statement again with
11984 the same or different parameter values. If no cursor is open, this option has no effect.
11985 (Calling *SQLCloseCursor*() also closes a cursor.)

11986 SQL_UNBIND
11987 Sets the SQL_DESC_COUNT field of the ARD to 0, releasing all column buffers bound
11988 by *SQLBindCol*() for *StatementHandle*. The SQL_DESC_DATA_PTR field of the ARD for
11989 the bookmark column is set to NULL to release any bound bookmark column. If this
11990 operation is performed on an explicitly-allocated descriptor that is shared by more than
11991 one statement, it affects the bindings of all statements that share the descriptor.

11992 SQL_RESET_PARAMS
11993 Sets the SQL_DESC_COUNT field of the APD to 0, releasing all parameter buffers set
11994 by *SQLBindParameter*() for *StatementHandle*. If this operation is performed on an
11995 explicitly-allocated descriptor that is shared by more than one statement, it affects the
11996 bindings of all the statements that share the descriptor.

11997 **RETURN VALUE**

11998 SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.

11999 **DIAGNOSTICS**

12000 When *SQLFreeStmt*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated
12001 SQLSTATE value can be obtained by calling *SQLGetDiagRec*() with a *HandleType* of
12002 SQL_HANDLE_STMT and a *Handle* of *StatementHandle*. The following SQLSTATE values are
12003 commonly returned by *SQLFreeStmt*(). The return code associated with each SQLSTATE value is
12004 SQL_ERROR, except that for SQLSTATE values in class 01, the return code is
12005 SQL_SUCCESS_WITH_INFO.

12006 01000 — General warning
12007 Implementation-defined informational message.

12008 HY000 — General error
12009 An error occurred for which there was no specific SQLSTATE and for which no
12010 implementation-specific SQLSTATE was defined. The error message returned by
12011 *SQLGetDiagRec*() in the **MessageText* buffer describes the error and its cause.

12012 HY001 — Memory allocation error
12013 The implementation failed to allocate memory required to support execution or completion
12014 of the function.

12015 HY010 — Function sequence error
12016 An asynchronously executing function was called for *StatementHandle* and was still
12017 executing when this function was called.

12018 *SQLBulkOperations*(), *SQLExecDirect*(), *SQLExecute*(), or *SQLSetPos*() was called for
12019 *StatementHandle* and returned SQL_NEED_DATA. This function was called before data was
12020 sent for all data-at-execution parameters or columns.

12021 HY092 — Invalid attribute identifier
12022 *Option* was not one of the following:

12023 SQL_CLOSE
12024 SQL_UNBIND
12025 SQL_RESET_PARAMS

12026 HYT01 — Connection timeout expired
12027 The connection timeout period expired before the data source responded to the request. The
12028 connection timeout period is set through *SQLSetConnectAttr*(),
12029 SQL_ATTR_CONNECTION_TIMEOUT.

12030 IM001 — Function not supported
12031 The function is not supported on the current connection to the data source.

12032 **COMMENTS**

12033 Calling *SQLFreeStmt*() with the SQL_CLOSE option is equivalent to calling *SQLCloseCursor*(),
12034 except in the case that no cursor is open on the statement. In this case, *SQLFreeStmt*() has no
12035 effect, but *SQLCloseCursor*() returns SQLSTATE 24000 (Invalid cursor state).

12036 **SEE ALSO**

12037	**For information about**	**See**
12038	Allocating a handle	*SQLAllocHandle*()
12039	Canceling statement processing	*SQLCancel*()
12040	Closing a cursor	*SQLCloseCursor*()
12041	Freeing a handle	*SQLFreeHandle*()
12042	Setting a cursor name	*SQLSetCursorName*()

12043 **CHANGE HISTORY**

12044 **Version 2**

12045 Revised generally. See **Alignment with Popular Implementations** on page 2. The function was
12046 deprecated in Version 1. The former SQL_DROP value of *Option* has been deleted. The other
12047 three values remain and the function is no longer deprecated.

12048 **NAME**

12049 SQLGetConnectAttr — Return the current setting of a connection attribute.

12050 **SYNOPSIS**

```
12051 SQLRETURN SQLGetConnectAttr(
12052   SQLHDBC ConnectionHandle,
12053   SQLINTEGER Attribute,
12054   SQLPOINTER ValuePtr,
12055   SQLINTEGER BufferLength,
12056   SQLINTEGER * StringLengthPtr);
```

12057 **ARGUMENTS**

12058 *ConnectionHandle* [Input]

12059 Connection handle.

12060 *Attribute* [Input]

12061 Attribute to retrieve.

12062 *ValuePtr* [Output]

12063 A pointer to memory in which to return the current value of the attribute specified by
12064 *Attribute*.

12065 *BufferLength*

12066 If *ValuePtr* points to data of variable length, this argument should be the length of **ValuePtr*.
12067 If what is contained in *ValuePtr* is itself a pointer, but not to data of variable length, then
12068 *BufferLength* should have the value SQL_IS_POINTER. If what is contained in *ValuePtr* is
12069 actual data of fixed length, then *BufferLength* should have the value
12070 SQL_IS_NOT_POINTER.

12071 *StringLengthPtr* [Output]

12072 A pointer to a buffer in which to return the total number of octets (excluding the null
12073 terminator) available to return in **ValuePtr*. If *ValuePtr* is a pointer, no length is returned. If
12074 the attribute value is a character string, and the number of octets available to return is
12075 greater than or equal to *BufferLength*, the data in **ValuePtr* is truncated to *BufferLength* minus
12076 the length of a null terminator and is null terminated.

12077 **RETURN VALUE**

12078 SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NO_DATA, SQL_ERROR, or
12079 SQL_INVALID_HANDLE.

12080 **DIAGNOSTICS**

12081 When *SQLGetConnectAttr*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated
12082 SQLSTATE value may be obtained from the diagnostic data structure by calling
12083 *SQLGetDiagRec*() with a *HandleType* of SQL_HANDLE_DBC and a *Handle* of *ConnectionHandle*.
12084 The following SQLSTATE values are commonly returned by *SQLGetConnectAttr*(). The return
12085 code associated with each SQLSTATE value is SQL_ERROR, except that for SQLSTATE values in
12086 class 01, the return code is SQL_SUCCESS_WITH_INFO.

12087 01000 — General warning

12088 Implementation-defined informational message.

12089 01004 — String data, right truncation

12090 The data returned in **ValuePtr* was truncated to be *BufferLength* minus the length of a null
12091 terminator. The length of the untruncated string value is returned in **StringLengthPtr*.

12092 08003 — Connection does not exist

12093 An *Attribute* was specified that required an open connection.

08S01 — Communication link failure
The communication link to the data source failed before the function completed processing.

HY000 — General error
An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned from the diagnostic data structure by *MessageText* in *SQLGetDiagField*() describes the error and its cause.

HY001 — Memory allocation error
The implementation failed to allocate memory required to support execution or completion of the function.

HY010 — Function sequence error
SQLBrowseConnect() was called for *ConnectionHandle* and returned SQL_NEED_DATA. This function was called before *SQLBrowseConnect*() returned SQL_SUCCESS_WITH_INFO or SQL_SUCCESS.

HY092 — Invalid attribute identifier
Attribute was not valid for this connection to this data source.

HYC00 — Optional feature not implemented
Attribute was valid but is not supported by the data source.

HYT01 — Connection timeout expired
The connection timeout period expired before the data source responded to the request. The connection timeout period is set through *SQLSetConnectAttr*(), SQL_ATTR_CONNECTION_TIMEOUT.

IM001 — Function not supported
The function is not supported on the current connection to the data source.

COMMENTS

For a list of attributes that can be set, see *SQLSetConnectAttr*(). If *Attribute* specifies an attribute that returns a string, *ValuePtr* must be a pointer to a buffer for the string. The maximum length of the string, including the null terminator, is *BufferLength* octets.

Depending on the attribute, an application does not need to establish a connection prior to calling *SQLGetConnectAttr*(). However, if *SQLGetConnectAttr*() is called and the specified attribute does not have a default value and has not been set by a prior call to *SQLSetConnectAttr*(), *SQLGetConnectAttr*() returns SQL_NO_DATA.

While an application can set statement attributes using *SQLSetConnectAttr*(), an application cannot use *SQLGetConnectAttr*() to retrieve statement attribute values; it must call *SQLGetStmtAttr*() to retrieve the setting of statement attributes.

The SQL_ATTR_AUTO_IPD connection attribute can be returned by a call to *SQLGetConnectAttr*(), but cannot be set by a call to *SQLSetConnectAttr*().

SEE ALSO

For information about	See
Returning the setting of a statement attribute	*SQLGetStmtAttr*()
Setting a connection attribute	*SQLSetConnectAttr*()
Setting a statement attribute	*SQLSetStmtAttr*()

CHANGE HISTORY

12137 **Version 2**

12138 Revised generally. See **Alignment with Popular Implementations** on page 2. See also the list in
12139 **New Connection Attributes in Version 2** on page 461.

12140 **NAME**

12141 SQLGetCursorName — Return the cursor name associated with a specified statement.

12142 **SYNOPSIS**

```
12143 SQLRETURN SQLGetCursorName(
12144   SQLHSTMT StatementHandle,
12145   SQLCHAR * CursorName,
12146   SQLSMALLINT BufferLength,
12147   SQLSMALLINT * NameLengthPtr);
```

12148 **ARGUMENTS**

12149 *StatementHandle* [Input]

12150 Statement handle.

12151 *CursorName* [Output]

12152 Pointer to a buffer in which to return the cursor name.

12153 *BufferLength* [Input]

12154 Length of **CursorName*, in octets.

12155 *NameLengthPtr* [Output]

12156 Pointer to memory in which to return the total number of octets (excluding the null
12157 terminator) available to return in **CursorName*. If the number of octets available to return is
12158 greater than or equal to *BufferLength*, the cursor name in **CursorName* is truncated to
12159 *BufferLength* minus the length of a null terminator.

12160 **RETURN VALUE**

12161 SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.

12162 **DIAGNOSTICS**

12163 When *SQLGetCursorName*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated
12164 SQLSTATE value can be obtained by calling *SQLGetDiagRec*() with a *HandleType* of
12165 SQL_HANDLE_STMT and a *Handle* of *StatementHandle*. The following SQLSTATE values are
12166 commonly returned by *SQLGetCursorName*(). The return code associated with each SQLSTATE
12167 value is SQL_ERROR, except that for SQLSTATE values in class 01, the return code is
12168 SQL_SUCCESS_WITH_INFO.

12169 01000 — General warning

12170 Implementation-defined informational message.

12171 01004 — String data, right truncation

12172 The buffer **CursorName* was not large enough to return the entire cursor name, so the cursor
12173 name was truncated. The length of the untruncated cursor name is returned in
12174 **NameLengthPtr*.

12175 HY000 — General error

12176 An error occurred for which there was no specific SQLSTATE and for which no
12177 implementation-specific SQLSTATE was defined. The error message returned by
12178 *SQLGetDiagRec*() in the **MessageText* buffer describes the error and its cause.

12179 HY001 — Memory allocation error

12180 The implementation failed to allocate memory required to support execution or completion
12181 of the function.

12182 HY010 — Function sequence error

12183 An asynchronously executing function was called for *StatementHandle* and was still
12184 executing when this function was called.

12185 *SQLBulkOperations*(), *SQLExecDirect*(), *SQLExecute*(), or *SQLSetPos*() was called for
12186 *StatementHandle* and returned SQL_NEED_DATA. This function was called before data was
12187 sent for all data-at-execution parameters or columns.

12188 HY090 — Invalid string or buffer length
12189 *BufferLength* was less than 0.

12190 HYT01 — Connection timeout expired
12191 The connection timeout period expired before the data source responded to the request. The
12192 connection timeout period is set through *SQLSetConnectAttr*(),
12193 SQL_ATTR_CONNECTION_TIMEOUT.

12194 IM001 — Function not supported
12195 The function is not supported on the current connection to the data source.

12196 **COMMENTS**

12197 Cursor names are used only in positioned UPDATE and DELETE statements (for example,
12198 `UPDATE` *`table-name`* `... WHERE CURRENT OF` *`cursor-name`*). If the application does not
12199 call *SQLSetCursorName*() to define a cursor name, the implementation generates a name when
12200 preparing or executing any statement that produces a result set. This name begins with
12201 SQL_CUR and does not exceed [SQL_MAX_ID_LENGTH] characters in length.

12202 *SQLGetCursorName*() returns the cursor name regardless of whether the name was created
12203 explicitly or implicitly. A cursor name is implicitly generated if *SQLSetCursorName*() is not
12204 called.

12205 A cursor name that is set either explicitly or implicitly remains set until the *StatementHandle* with
12206 which it is associated is dropped, using *SQLFreeHandle*() with a *HandleType* of
12207 SQL_HANDLE_STMT.

12208 **SEE ALSO**

12209 **For information about**	**See**
12210 Executing an SQL statement	*SQLExecDirect*()
12211 Executing a prepared SQL statement	*SQLExecute*()
12212 Preparing a statement for execution	*SQLPrepare*()
12213 Setting a cursor name	*SQLSetCursorName*()

12214 **CHANGE HISTORY**

12215 **Version 2**

12216 Revised generally. See **Alignment with Popular Implementations** on page 2.

12217 **NAME**

12218 SQLGetData — Retrieve data for a single column in the result set.

12219 **SYNOPSIS**

```
12220 SQLRETURN SQLGetData(
12221   SQLHSTMT StatementHandle,
12222   SQLUSMALLINT ColumnNumber,
12223   SQLSMALLINT TargetType,
12224   SQLPOINTER TargetValuePtr,
12225   SQLINTEGER BufferLength,
12226   SQLINTEGER * StrLen_or_IndPtr);
```

12227 **ARGUMENTS**

12228 *StatementHandle* [Input]

12229 Statement handle.

12230 *ColumnNumber* [Input]

12231 Number of the column for which to return data. Result set columns are numbered from left
12232 to right starting at 1. The bookmark column is column number 0; this can be specified only
12233 if bookmarks are used.

12234 *TargetType* [Input]

12235 The type identifier of the C data type of the **TargetValuePtr* buffer. For a list of valid C data
12236 types and type identifiers, see Section D.2 on page 560. If *TargetType* is SQL_ARD_TYPE, the
12237 implementation uses the type identifier specified in the SQL_DESC_TYPE field of the ARD.
12238 If it is SQL_C_DEFAULT, the implementation selects a buffer type based upon the SQL data
12239 type of the source.

12240 If *TargetType* is a SQL_C_NUMERIC data type, the precision and default scale fields of the
12241 SQL_C_NUMERIC structure are used by default. The SQL_DESC_PRECISION and
12242 SQL_DESC_SCALE fields of the ARD are set to the same value. If the default precision or
12243 scale is not appropriate, the application should explicitly set the descriptor field by a call to
12244 *SQLSetDescField*() or *SQLSetDescRec*(). It should set the SQL_DESC_CONCISE_TYPE field
12245 to SQL_C_NUMERIC, and call *SQLGetData*() with a *TargetType* of SQL_ARD_TYPE, which
12246 causes the precision and scale values in the descriptor fields to be used.

12247 *TargetValuePtr* [Output]

12248 Pointer to the buffer in which to return the data.

12249 *BufferLength* [Input]

12250 Length of the **TargetValuePtr* buffer in octets.

12251 The implementation uses *BufferLength* to avoid writing past the end of the **TargetValuePtr*
12252 buffer when returning variable-length data, such as character or binary data. The
12253 implementation counts the null terminator when returning character data to
12254 **TargetValuePtr*. **TargetValuePtr* must therefore contain space for the null terminator or the
12255 implementation truncates the data.

12256 When the data source returns fixed-length data, such as an integer or a date structure, the
12257 implementation ignores *BufferLength* and assumes the buffer is large enough to hold the
12258 data. It is therefore important for the application to allocate a large enough buffer for fixed-
12259 length data or the implementation writes past the end of the buffer.

12260 *SQLGetData*() returns SQLSTATE HY090 (Invalid string or buffer length) when *BufferLength*
12261 is less than 0 but not when *BufferLength* is 0. However, if *TargetType* specifies a character
12262 type, an application should not set BufferLength to 0, because the buffer must have space
12263 for the null terminator.

12264 If *TargetValuePtr* is set to a null pointer, *BufferLength* is ignored.

12265 *StrLen_or_IndPtr* [Output]
12266 Pointer to the buffer in which to return the length or indicator value. If this is a null pointer,
12267 no length or indicator value is returned. This returns an error when the data being fetched is
12268 NULL.

12269 *SQLGetData*() can return the following values in the length/indicator buffer:

12270 • The length of the data available to return
12271 • SQL_NO_TOTAL
12272 • SQL_NULL_DATA

12273 For more information, see Section 4.3.5 on page 42 and the "Comments" section.

12274 **RETURN VALUE**
12275 SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NO_DATA, SQL_STILL_EXECUTING,
12276 SQL_ERROR, or SQL_INVALID_HANDLE.

12277 **DIAGNOSTICS**
12278 When *SQLGetData*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated
12279 SQLSTATE value can be obtained by calling *SQLGetDiagRec*() with a *HandleType* of
12280 SQL_HANDLE_STMT and a *Handle* of *StatementHandle*. The following table lists the SQLSTATE
12281 values commonly returned by *SQLGetData*(). The return code associated with each SQLSTATE
12282 value is SQL_ERROR, except that for SQLSTATE values in class 01, the return code is
12283 SQL_SUCCESS_WITH_INFO.

12284 01000 — General warning
12285 Implementation-defined informational message.

12286 01004 — String data, right truncation
12287 All of the data for the column specified by *ColumnNumber* could not be retrieved in a single
12288 call to the function. The length of the data remaining in the specified column prior to the
12289 current call to *SQLGetData*() is returned in **StrLen_or_IndPtr*.

12290 *TargetValuePtr* was a null pointer and more data was available to return. Following this
12291 diagnostic, the application can retrieve truncated text as described in **Retrieving Variable-**
12292 **Length Data in Parts** on page 347.

12293 01S07 — Fractional truncation
12294 The data returned for one or more columns was truncated. For numeric data types, the
12295 fractional part of the number was truncated. For time, timestamp, and interval data types
12296 containing a time component, the fractional portion of the time was truncated.

12297 07006 — Restricted data type attribute violation
12298 The data value of a column in the result set cannot be converted to the C data type specified
12299 by *TargetType*.

12300 07009 — Invalid descriptor index
12301 *ColumnNumber* was 0 and the SQL_ATTR_USE_BOOKMARKS statement attribute was set
12302 to SQL_UB_OFF.

12303 *ColumnNumber* was greater than the number of columns in the result set.

12304 *ColumnNumber* was less than 0.

12305 The specified column was bound. This description does not apply to implementations that
12306 return the SQL_GD_BOUND bitmask for the SQL_GETDATA_EXTENSIONS option in
12307 *SQLGetInfo*().

12308 The number of the specified column was less than or equal to the number of the highest
12309 bound column. This description does not apply to implementations that return the
12310 SQL_GD_ANY_COLUMN bitmask for the SQL_GETDATA_EXTENSIONS option in
12311 *SQLGetInfo*().

12312 The application has already called *SQLGetData*() for the current row; the number of the
12313 column specified in the current call was less than the number of the column specified in the
12314 preceding call; and the implementation does not return the SQL_GD_ANY_ORDER
12315 bitmask for the SQL_GETDATA_EXTENSIONS option in *SQLGetInfo*().

12316 *TargetType* was SQL_ARD_TYPE and the *ColumnNumber* descriptor record failed the
12317 consistency check.

12318 *TargetType* was SQL_ARD_TYPE and no records existed in the ARD.

12319 08S01 — Communication link failure
12320 The communication link to the data source failed before the function completed processing.

12321 22002 — Indicator variable required but not supplied
12322 *StrLen_or_IndPtr* was a null pointer and NULL data was retrieved.

12323 22003 — Numeric value out of range
12324 Returning the numeric value (as numeric or string) for the column would have caused the
12325 whole (as opposed to fractional) part of the number to be truncated.

12326 For more information, see Appendix D.

12327 22007 — Invalid date/time format
12328 The character column in the results set was bound to a C date, time, or timestamp structure,
12329 and the value in the column was an invalid date, time, or timestamp, respectively. For more
12330 information, see Appendix D.

12331 22012 — Division by zero
12332 A value from an arithmetic expression that resulted in division by zero was returned.

12333 22015 — Interval field overflow
12334 An exact numeric column in the result set was bound to an interval C structure and
12335 returning the data caused a loss of significant digits.

12336 An interval column in the result set was bound to an interval C structure and returning the
12337 data caused a loss of significant digits.

12338 Data in the result set was bound to an interval C structure and there was no representation
12339 of the data in the interval C structure.

12340 22018 — Invalid character value
12341 The character column in the result set was bound to a character C buffer and the column
12342 contained a character for which there was no representation in the character set of the C
12343 buffer.

12344 A character column in the result set was bound to an approximate numeric C buffer and a
12345 value in the column could not be cast to a valid approximate numeric value.

12346 A character column in the result set was bound to an exact numeric C buffer and a value in
12347 the column could not be cast to a valid exact numeric value.

12348 A character column in the result set was bound to a date/time or interval C buffer and a
12349 value in the column could not be cast to a valid date/time or interval value.

12350 24000 — Invalid cursor state
12351 *StatementHandle* was in an executed state but no result set was associated with it.

12352 A cursor was open on *StatementHandle* and *SQLFetch*() or *SQLFetchScroll*() had been called,
12353 but the cursor was positioned before the start of the result set or after the end of the result
12354 set.

12355 HY000 — General error
12356 An error occurred for which there was no specific SQLSTATE and for which no
12357 implementation-specific SQLSTATE was defined. The error message returned by
12358 *SQLGetDiagRec*() in the MessageText buffer describes the error and its cause.

12359 HY001 — Memory allocation error
12360 The implementation failed to allocate memory required to support execution or completion
12361 of the function.

12362 HY003 — Invalid application buffer type
12363 *TargetType* was neither a valid data type nor SQL_C_DEFAULT.

12364 *ColumnNumber* was 0 and *TargetType* was not SQL_C_VARBOOKMARK.

12365 HY008 — Operation canceled
12366 Asynchronous processing was enabled for *StatementHandle*. The function was called and
12367 before it completed execution, *SQLCancel*() was called on *StatementHandle*. The function
12368 was then called again on *StatementHandle*.

12369 The function was called and, before it completed execution, *SQLCancel*() was called on
12370 *StatementHandle* from a different thread in a multithread application.

12371 HY010 — Function sequence error
12372 *StatementHandle* was not in an executed state. The function was called without first calling
12373 *SQLExecDirect*(), *SQLExecute*(), or a catalog function.

12374 An asynchronously executing function (not this one) was called for *StatementHandle* and
12375 was still executing when this function was called.

12376 *SQLBulkOperations*(), *SQLExecDirect*(), *SQLExecute*(), or *SQLSetPos*() was called for
12377 *StatementHandle* and returned SQL_NEED_DATA. This function was called before data was
12378 sent for all data-at-execution parameters or columns.

12379 The function was called without first calling *SQLFetch*() or *SQLFetchScroll*().

12380 *StatementHandle* was in an executed state but no result set was associated with
12381 *StatementHandle.*

12382 HY090 — Invalid string or buffer length
12383 *BufferLength* was less than 0.

12384 HY109 — Invalid cursor position
12385 The cursor was positioned (by *SQLSetPos*() or *SQLFetchScroll*()) on a row that had been
12386 deleted or could not be fetched.

12387 The cursor was a forward-only cursor and the row-set size was greater than one.

12388 HYC00 — Optional feature not implemented
12389 The data source does not support use of *SQLGetData*() with multiple rows in
12390 *SQLFetchScroll*(). This description does not apply to data sources that return the
12391 SQL_GD_BLOCK bitmask for the SQL_GETDATA_EXTENSIONS option in *SQLGetInfo*().

12392 The implementation does not support the conversion specified by the combination of
12393 *TargetType* and the SQL data type of the corresponding column. This error only applies
12394 when the SQL data type of the column was mapped to an implementation-defined SQL
12395 data type.

12396 HYT01 — Connection timeout expired
12397 The connection timeout period expired before the data source responded to the request. The
12398 connection timeout period is set through *SQLSetConnectAttr*(),
12399 SQL_ATTR_CONNECTION_TIMEOUT.

12400 IM001 — Function not supported
12401 The function is not supported on the current connection to the data source.

12402 **COMMENTS**

12403 *SQLGetData*() returns the data in a specified column. *SQLGetData*() can only be called after one
12404 or more rows have been fetched from the result set by *SQLFetch*(), *SQLFetchScroll*(), or
12405 *SQLSetPos*(). If variable-length data is too large to be returned in a single call to *SQLGetData*()
12406 (due to a limitation in the application), *SQLGetData*() can retrieve it in parts. It is possible to bind
12407 some columns in a row and call *SQLGetData*() for others, although this is subject to some
12408 restrictions.

12409 **Restrictions on Use of SQLGetData()**

12410 Portable applications should only use *SQLGetData*(), to retrieve data for unbound columns with
12411 a number greater than that of the last bound column. (However, any implementation that
12412 supports bookmarks allows calls to *SQLGetData*() for column 0, which retrieves the bookmark.)
12413 Furthermore, within a row of data, the value of *ColumnNumber* in each call to *SQLGetData*()
12414 should be greater than or equal to the value of *ColumnNumber* in the previous call; that is, data
12415 should be retrieved in increasing column number order. Finally, *SQLGetData*() should not be
12416 called if the row-set size is greater than 1.

12417 Implementations can relax the above restrictions. To determine what additional operations an
12418 implementation supports, an application calls *SQLGetInfo*() with an *InfoItem* of
12419 SQL_GETDATA_EXTENSIONS.

12420 Regardless of what extensions an implementation allows, applications should not call
12421 *SQLGetData*() for a forward-only cursor when the row-set size is greater than 1 because the row
12422 position is undefined.

12423 *SQLGetData*() cannot be used to retrieve the bookmark for a row just inserted by calling
12424 *SQLBulkOperations*() with the SQL_ADD option, because the cursor is not positioned on the row.
12425 An application can retrieve the bookmark for such a row by binding column 0 before calling
12426 *SQLBulkOperations*() with SQL_ADD, in which case *SQLBulkOperations*() returns the bookmark
12427 in the bound buffer. *SQLSetPos*() can then be called with SQL_POSITION to reposition the
12428 cursor on that row, at which point *SQLGetData*() can be called to retrieve the bookmark.

12429 **Retrieving Variable-Length Data in Parts**

12430 *SQLGetData*() can be used to retrieve data from a column that contains variable-length data in
12431 parts — that is, when the identifier of the SQL data type of the column is SQL_CHAR,
12432 SQL_VARCHAR, SQL_LONGVARCHAR, SQL_BINARY, SQL_VARBINARY,
12433 SQL_LONGVARBINARY, or an implementation-defined identifier for a variable-length type.

12434 To retrieve data from a column in parts, the application calls *SQLGetData*() multiple times in
12435 succession for the same column. On each call, *SQLGetData*() returns the next part of the data.
12436 The application has to reassemble the parts, removing any null terminators from intermediate
12437 parts of character data. If there is more data to return, *SQLGetData*() returns SQLSTATE 01004
12438 (Data truncated) and SQL_SUCCESS_WITH_INFO. When it returns the last part of the data,
12439 *SQLGetData*() returns SQL_SUCCESS. SQL_NO_TOTAL or 0 are not returned on the last valid
12440 call to retrieve data from a column, because the application would then have no way of knowing
12441 how much of the data in the application buffer is valid. If *SQLGetData*() is called after this, it
12442 returns SQL_NO_DATA (see below).

12443 *SQLGetData*() can return bookmarks in parts. As with other data, a call to *SQLGetData*() returns
12444 SQLSTATE 01004 (String data, right truncation) and SQL_SUCCESS_WITH_INFO when there is
12445 more data to be returned. This is different from the case when a bookmark is truncated by a call
12446 to *SQLFetch*() or *SQLFetchScroll*(), which returns SQL_ERROR and SQLSTATE 22001 (String
12447 data, right truncation).

12448 *SQLGetData*() cannot be used to return fixed-length data in parts. If *SQLGetData*() is called more
12449 than once for a column containing fixed-length data, it returns SQL_NO_DATA for all calls after
12450 the first.

12451 **Retrieving Data with SQLGetData**

12452 To return data for the specified column, *SQLGetData*() performs the following sequence of steps:

12453 1. Returns SQL_NO_DATA if it has already returned all of the data for the column.

12454 2. Sets **StrLen_or_IndPtr* to SQL_NULL_DATA if the data is NULL. If the data is NULL and
12455 *StrLen_or_IndPtr* was a null pointer, *SQLGetData*() returns SQLSTATE 22002 (Indicator
12456 variable required but not supplied).

12457 If the data for the column is not NULL, *SQLGetData*() proceeds to step 3.

12458 3. If the SQL_ATTR_MAX_LENGTH statement attribute is set to a nonzero value, the
12459 column contains character or binary data, and *SQLGetData*() has not previously been
12460 called for the column, the data is truncated to SQL_ATTR_MAX_LENGTH octets. (The
12461 SQL_ATTR_MAX_LENGTH statement attribute is intended to reduce network traffic. It is
12462 generally implemented by the data source, which truncates the data before returning it
12463 across the network. Implementations are not required to support it. Therefore, to
12464 guarantee that data is truncated to a particular size, an application should allocate a buffer
12465 of that size and specify the size in the *BufferLength* argument.)

12466 4. Converts the data to the type specified in *TargetType*. The data is given the default
12467 precision and scale for that data type. If *TargetType* is SQL_ARD_TYPE, the data type in
12468 the SQL_DESC_CONCISE_TYPE field of the ARD is used. If *TargetType* is either
12469 SQL_ARD_TYPE or SQL_C_DEFAULT, the data is given the precision and scale in the
12470 SQL_DESC_DATETIME_INTERVAL_PRECISION, SQL_DESC_PRECISION, and
12471 SQL_DESC_SCALE fields of the ARD, depending on the data type in the
12472 SQL_DESC_CONCISE_TYPE field.

12473 5. If the data was converted to a variable-length data type, such as character or binary,
12474 *SQLGetData*() checks whether the length of the data exceeds *BufferLength*. If the length of
12475 character data (including the null terminator) exceeds *BufferLength*, *SQLGetData*()
12476 truncates the data to *BufferLength* less the length of a null terminator. It then null-
12477 terminates the data. If the length of binary data exceeds the length of the data buffer,
12478 *SQLGetData*() truncates it to *BufferLength* octets.

12479 *SQLGetData*() never truncates data converted to fixed-length data types; it always assumes
12480 that the length of **TargetValuePtr* is the size of the data type.

12481 6. Places the converted (and possibly truncated) data in **TargetValuePtr*.

12482 7. Places the length of the data in **StrLen_or_IndPtr*. If *StrLen_or_IndPtr* was a null pointer,
12483 *SQLGetData*() does not return the length.

12484 — For character or binary data, this is the length of the data after conversion and before
12485 truncation due to *BufferLength*. If the implementation cannot determine the length of
12486 the data after conversion, as is sometimes the case with long data, it returns
12487 SQL_SUCCESS_WITH_INFO and sets the length to SQL_NO_TOTAL. (The last call to
12488 *SQLGetData*() must always return the length of the data, not SQL_NO_TOTAL.) If data

12489 was truncated due to the SQL_ATTR_MAX_LENGTH statement attribute, the value of
12490 this attribute-as opposed to the actual length-is placed in **StrLen_or_IndPtr*. This is
12491 because this attribute is designed to truncate data on the server before conversion, so
12492 the implementation cannot determine the actual length. When *SQLGetData*() is called
12493 multiple times in succession for the same column, this is the length of the data available
12494 at the start of the current call; that is, the length decreases with each subsequent call.

12495 — For all other data types, this is the length of the data after conversion; that is, it is the
12496 size of the type to which the data was converted.

12497 8. If the data is truncated without loss of significance during conversion (for example, the
12498 real number 1.234 is truncated when converted to the integer 1) or because *BufferLength* is
12499 too small (for example, the string 'abcdef' is placed in a 4-octet buffer), *SQLGetData*()
12500 returns SQLSTATE 01004 (Data truncated) and SQL_SUCCESS_WITH_INFO. If data is
12501 truncated without loss of significance due to the SQL_ATTR_MAX_LENGTH statement
12502 attribute, *SQLGetData*() returns SQL_SUCCESS and does not return SQLSTATE 01004
12503 (Data truncated).

12504 The contents of the bound data buffer (if *SQLGetData*() is called on a bound buffer) and the
12505 length/indicator buffer are undefined if *SQLGetData*() does not return SQL_SUCCESS or
12506 SQL_SUCCESS_WITH_INFO.

12507 **Descriptors and SQLGetData**

12508 *SQLGetData*() does not interact directly with any descriptor fields unless SQL_ARD_TYPE is
12509 specified, in which case it examines the descriptor record specified by *ColumnNumber* to
12510 determine attributes of the C buffer.

12511 **SEE ALSO**

12512	For information about	See
12513	Assigning storage for a column in a result set	*SQLBindCol*()
12514	Canceling statement processing	*SQLCancel*()
12515	Executing an SQL statement	*SQLExecDirect*()
12516	Executing a prepared SQL statement	*SQLExecute*()
12517	Fetching a block of data or scrolling through a result set	*SQLFetchScroll*()
12518 12519	Fetching a single row of data or a block of data in a forward-only direction	*SQLFetch*()
12520	Sending parameter data at execution time	*SQLPutData*()

12521 **CHANGE HISTORY**

12522 **Version 2**

12523 Revised generally. See **Alignment with Popular Implementations** on page 2.

12524 **NAME**

12525 SQLGetDescField — Return the current settings of a single field of a descriptor record.

12526 **SYNOPSIS**

```
12527 SQLRETURN SQLGetDescField(
12528   SQLHDESC DescriptorHandle,
12529   SQLSMALLINT RecNumber,
12530   SQLSMALLINT FieldIdentifier,
12531   SQLPOINTER ValuePtr,
12532   SQLINTEGER BufferLength,
12533   SQLINTEGER * StringLengthPtr);
```

12534 **ARGUMENTS**

12535 *DescriptorHandle* [Input]

12536 Descriptor handle.

12537 *RecNumber* [Input]

12538 Indicates the descriptor record from which the application seeks information. Descriptor
12539 records are numbered from 0, with record number 0 being the bookmark record. If
12540 *FieldIdentifier* indicates a field of the descriptor header record, *RecNumber* must be 0. If
12541 *RecNumber* is less than SQL_DESC_COUNT, but the row does not contain data for a column
12542 or parameter, a call to *SQLGetDescField*() returns the default values of the fields (for more
12543 information, see **Initialization of Descriptor Fields** on page 467).

12544 *FieldIdentifier* [Input]

12545 Indicates the field of the descriptor whose value is to be returned. For a list of valid values,
12546 see *SQLSetDescField*().

12547 *ValuePtr* [Output]

12548 Pointer to a buffer in which to return the descriptor information. The data type depends on
12549 the value of *FieldIdentifier*.

12550 *BufferLength* [Input]

12551 If *ValuePtr* points to data of variable length, this argument should be the length of **ValuePtr*.
12552 If what is contained in *ValuePtr* is itself a pointer, but not to data of variable length, then
12553 *BufferLength* should have the value SQL_IS_POINTER. If what is contained in *ValuePtr* is
12554 actual data of fixed length, then *BufferLength* should have the value
12555 SQL_IS_NOT_POINTER.

12556 *StringLengthPtr* [Output]

12557 Pointer to the total number of octets (excluding the number of octets required for the null
12558 terminator) available to return in **ValuePtr*.

12559 **RETURN VALUE**

12560 SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, SQL_NO_DATA, or
12561 SQL_INVALID_HANDLE.

12562 SQL_NO_DATA is returned if *RecNumber* is greater than the number of descriptor records.

12563 **DIAGNOSTICS**

12564 When *SQLGetDescField*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated
12565 SQLSTATE value can be obtained by calling *SQLGetDiagRec*() with a *HandleType* of
12566 SQL_HANDLE_STMT and a *Handle* of *StatementHandle*. The following table lists the SQLSTATE
12567 values commonly returned by *SQLGetDescField*(). The return code associated with each
12568 SQLSTATE value is SQL_ERROR, except that for SQLSTATE values in class 01, the return code is
12569 SQL_SUCCESS_WITH_INFO.

12570 01000 — General warning
12571 Implementation-defined informational message.

12572 01004 — String data, right truncation
12573 The buffer **ValuePtr* was not large enough to return the entire descriptor field, so the field
12574 was truncated. The length of the untruncated descriptor field is returned in **StringLengthPtr*.

12575 07009 — Invalid descriptor index
12576 *RecNumber* was equal to 0, the SQL_ATTR_USE_BOOKMARK statement attribute was
12577 SQL_UB_OFF, and *DescriptorHandle* was an IRD handle. (This error can be returned for an
12578 automatically-allocated descriptor only if the descriptor is associated with a statement
12579 handle.)

12580 *FieldIdentifier* was a record field, *RecNumber* was 0, and *DescriptorHandle* was an IPD handle.

12581 *RecNumber* was less than 0.

12582 08S01 — Communication link failure
12583 The communication link to the data source failed before the function completed processing.

12584 24000 — Invalid cursor state
12585 There was no open cursor, and *DescriptorHandle* was an IRD handle.

12586 HY000 — General error
12587 An error occurred for which there was no specific SQLSTATE and for which no
12588 implementation-specific SQLSTATE was defined. The error message returned by
12589 *SQLGetDiagRec*() in the **MessageText* buffer describes the error and its cause.

12590 HY001 — Memory allocation error
12591 The implementation failed to allocate memory required to support execution or completion
12592 of the function.

12593 HY007 — Associated statement is not prepared
12594 *DescriptorHandle* was associated with an IRD, and the associated statement handle was not
12595 in the prepared or executed state.

12596 HY010 — Function sequence error
12597 *DescriptorHandle* was associated with a statement handle for which an asynchronously
12598 executing function (not this one) was called and was still executing when this function was
12599 called.

12600 *DescriptorHandle* was associated with a statement handle for which *SQLBulkOperations*(),
12601 *SQLExecDirect*(), *SQLExecute*(), or *SQLSetPos*() was called and returned SQL_NEED_DATA.
12602 This function was called before data was sent for all data-at-execution parameters or
12603 columns.

12604 A previous call to the same function returned SQL_STILL_EXECUTING and the present call
12605 specified a different value of *DescriptorHandle*, but one that pertains to the same statement
12606 handle.

12607 HY091 — Invalid descriptor field identifier
12608 *FieldIdentifier* was not an XDBC-defined field nor an implementation-defined value.

12609 *FieldIdentifier* was undefined for *DescriptorHandle*.

12610 *RecNumber* was greater than the value in the SQL_DESC_COUNT field.

12611 HYT01 — Connection timeout expired
12612 The connection timeout period expired before the data source responded to the request. The
12613 connection timeout period is set through *SQLSetConnectAttr*(),
12614 SQL_ATTR_CONNECTION_TIMEOUT.

12615 IM001 — Function not supported
12616 The function is not supported on the current connection to the data source.

12617 When the application calls *SQLGetDescField*() for an IRD, after *SQLPrepare*() and before
12618 *SQLExecute*(), it can return any SQLSTATEthat can be returned by *SQLPrepare*() or *SQLExecute*(),
12619 depending on when the data source evaluates the SQL statement associated with
12620 *StatementHandle* (see **Performance Note**).

12621 **COMMENTS**

12622 An application can call *SQLGetDescField*() to return the value of a single field of a descriptor
12623 record. A call to *SQLGetDescField*() can return the setting of any field in any descriptor type,
12624 including header fields, record fields, and bookmark fields. An application can obtain the
12625 settings of multiple fields in the same or different descriptors, in arbitrary order, by making
12626 repeated calls to *SQLGetDescField*(). *SQLGetDescField*() can also be called to return
12627 implementation-defined descriptor fields.

12628 The settings of multiple fields that describe the name, data type, and storage of column or
12629 parameter data can also be retrieved in a single call to *SQLGetDescRec*().

12630 *SQLGetStmtAttr*() can be called to return the setting of a single field in the descriptor header that
12631 is also a statement attribute.

12632 When an application calls *SQLGetDescField*() to retrieve the value of a field that is undefined for
12633 a particular descriptor type, the function returns SQLSTATE HY091 (Invalid descriptor field
12634 identifier). When an application calls *SQLGetDescField*() to retrieve the value of a field that is
12635 defined for a particular descriptor type, but has no default value and has not been set yet, the
12636 function returns SQL_SUCCESS but the value returned for the field is undefined. For more
12637 information, see **Initialization of Descriptor Fields** on page 467.

12638 The SQL_DESC_ALLOC_TYPE header field is available as read-only. This field is defined for all
12639 types of descriptors.

12640 The following record fields are available as read-only. Each of these fields is defined either for
12641 the IRD only, or for both the IRD and the IPD.

12642	SQL_DESC_AUTO_UNIQUE_VALUE	SQL_DESC_LITERAL_SUFFIX
12643	SQL_DESC_BASE_COLUMN_NAME	SQL_DESC_LOCAL_TYPE_NAME
12644	SQL_DESC_CASE_SENSITIVE	SQL_DESC_SCHEMA_NAME
12645	SQL_DESC_CATALOG_NAME	SQL_DESC_SEARCHABLE
12646	SQL_DESC_DISPLAY_SIZE	SQL_DESC_TABLE_NAME
12647	SQL_DESC_FIXED_PREC_SCALE	SQL_DESC_TYPE_NAME
12648	SQL_DESC_LABEL	SQL_DESC_UNSIGNED
12649	SQL_DESC_LITERAL_PREFIX	SQL_DESC_UPDATABLE

12650 For a description of these fields, and fields that can be set in a descriptor header or record, see
12651 the *SQLSetDescField*() section. For more information on descriptors, see Chapter 13.

12652 Calling *SQLGetDescField*() for an IRD between the preparation and the execution of an SQL
12653 statement has performance implications; see **Performance Note** on page 279.

12654 **SEE ALSO**

	For information about	See
12656	Getting multiple descriptor fields	*SQLGetDescRec*()

12657 Setting a single descriptor field *SQLSetDescField*()
12658 Setting multiple descriptor fields *SQLSetDescRec*()

12659 **CHANGE HISTORY**

12660 **Version 2**
12661 Revised generally. See **Alignment with Popular Implementations** on page 2. Also see the list in
12662 **Descriptor Fields Added in Version 2** on page 483.

12663 **NAME**

12664 SQLGetDescRec — Return the current settings of multiple fields of a descriptor record.

12665 **SYNOPSIS**

```
12666 SQLRETURN SQLGetDescRec(
12667   SQLHDESC DescriptorHandle,
12668   SQLSMALLINT RecNumber,
12669   SQLCHAR * Name,
12670   SQLSMALLINT BufferLength,
12671   SQLSMALLINT * StringLengthPtr,
12672   SQLSMALLINT * TypePtr,
12673   SQLSMALLINT * SubTypePtr,
12674   SQLINTEGER * LengthPtr,
12675   SQLSMALLINT * PrecisionPtr,
12676   SQLSMALLINT * ScalePtr,
12677   SQLSMALLINT * NullablePtr);
```

12678 **ARGUMENTS**

12679 *DescriptorHandle* [Input] |
12680 Descriptor handle. |

12681 *RecNumber* [Input]
12682 Indicates the descriptor record from which the application seeks information. Descriptor
12683 records are numbered from 0, with record number 0 being the bookmark record. *RecNumber*
12684 must be less than or equal to the value of SQL_DESC_COUNT. If *RecNumber* is less than
12685 SQL_DESC_COUNT, but the row does not contain data for a column or parameter,
12686 *SQLGetDescRec*() returns the default values of the fields (for more information, see |
12687 **Initialization of Descriptor Fields** on page 467). |

12688 *Name* [Output]
12689 A pointer to a buffer in which to return the SQL_DESC_NAME field for the descriptor
12690 record. |

12691 *BufferLength* [Input] |
12692 Length of the **Name* buffer, in octets. |

12693 *StringLengthPtr* [Output] |
12694 A pointer to a buffer in which to return the number of octets of data available to return in |
12695 the **Name* buffer, excluding the null terminator. If the number of octets was greater than or |
12696 equal to *BufferLength*, the data in **Name* is truncated to *BufferLength* minus the length of a |
12697 null terminator, and is null-terminated. |

12698 *TypePtr* [Output]
12699 A pointer to a buffer in which to return the value of the SQL_DESC_TYPE field for the
12700 descriptor record. |

12701 *SubTypePtr* [Output]
12702 For records whose type is SQL_DATETIME or SQL_INTERVAL, this is a pointer to a buffer
12703 in which to return the value of the SQL_DESC_DATETIME_INTERVAL_CODEfield. |

12704 *LengthPtr* [Output]
12705 A pointer to a buffer in which to return the value of the SQL_DESC_OCTET_LENGTH field
12706 for the descriptor record. |

12707 *PrecisionPtr* [Output]
12708 A pointer to a buffer in which to return the value of the SQL_DESC_PRECISION field for
12709 the descriptor record. |

12710 *ScalePtr* [Output]
12711 A pointer to a buffer in which to return the value of the SQL_DESC_SCALE field for the
12712 descriptor record.

12713 *NullablePtr* [Output]
12714 A pointer to a buffer in which to return the value of the SQL_DESC_NULLABLE field for
12715 the descriptor record.

12716 **RETURN VALUE**

12717 SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, SQL_NO_DATA, or
12718 SQL_INVALID_HANDLE.

12719 SQL_NO_DATA is returned if *RecNumber* is greater than the number of descriptor records.

12720 **DIAGNOSTICS**

12721 When *SQLGetDescRec*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated
12722 SQLSTATE value can be obtained by calling *SQLGetDiagRec*() with a *HandleType* of
12723 SQL_HANDLE_DESC and a *Handle* of *DescriptorHandle*. The following SQLSTATE values are
12724 commonly returned by *SQLGetDescRec*(). The return code associated with each SQLSTATE
12725 value is SQL_ERROR, except that for SQLSTATE values in class 01, the return code is
12726 SQL_SUCCESS_WITH_INFO.

12727 01000 — General warning
12728 Implementation-defined informational message.

12729 01004 — String data, right truncation
12730 The buffer **Name* was not large enough to return the entire descriptor field, so the field was
12731 truncated. The length of the untruncated descriptor field is returned in **StringLengthPtr*.

12732 07009 — Invalid descriptor index
12733 *FieldIdentifier* was a record field, *RecNumber* was 0 and *DescriptorHandle* argument was an
12734 IPD handle.

12735 *RecNumber* was 0, the SQL_ATTR_USE_BOOKMARKS statement attribute was
12736 SQL_UB_OFF, and *DescriptorHandle* was an IRD handle. (This error can be returned for an
12737 automatically-allocated descriptor only if the descriptor is associated with a statement
12738 handle.)

12739 *RecNumber* was less than 0.

12740 08S01 — Communication link failure
12741 The communication link to the data source failed before the function completed processing.

12742 24000 — Invalid cursor state
12743 There was no open cursor, and *DescriptorHandle* was an IRD handle.

12744 HY000 — General error
12745 An error occurred for which there was no specific SQLSTATE and for which no
12746 implementation-specific SQLSTATE was defined. The error message returned by
12747 *SQLGetDiagRec*() in the **MessageText* buffer describes the error and its cause.

12748 HY001 — Memory allocation error
12749 The implementation failed to allocate the memory required to support execution or
12750 completion of the function.

12751 HY007 — Associated statement is not prepared
12752 *DescriptorHandle* was associated with an IRD, and the associated statement handle was not
12753 in the prepared or executed state.

12754 HY010 — Function sequence error
12755 *DescriptorHandle* was associated with a statement handle for which an asynchronously

executing function (not this one) was called and was still executing when this function was called.

DescriptorHandle was associated with a statement handle for which *SQLBulkOperations*(), *SQLExecDirect*(), *SQLExecute*(), or *SQLSetPos*() was called and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.

A previous call to the same function returned SQL_STILL_EXECUTING and the present call specified a different value of *DescriptorHandle*, but one that pertains to the same statement handle.

HY091 — Invalid descriptor field identifier
A field to be retrieved was not defined for *DescriptorHandle*.

RecNumber was greater than the value in the SQL_DESC_COUNT field.

HYT01 — Connection timeout expired
The connection timeout period expired before the data source responded to the request. The connection timeout period is set through *SQLSetConnectAttr*(), SQL_ATTR_CONNECTION_TIMEOUT.

IM001 — Function not supported
The function is not supported on the current connection to the data source.

COMMENTS

An application can call *SQLGetDescRec*() to retrieve the values of the following fields for a single column or parameter:

- SQL_DESC_NAME
- SQL_DESC_TYPE
- SQL_DESC_DATETIME_INTERVAL_CODE(for date/time or interval records)
- SQL_DESC_OCTET_LENGTH
- SQL_DESC_PRECISION
- SQL_DESC_SCALE
- SQL_DESC_NULLABLE

SQLGetDescRec() does not retrieve header fields.

An application can inhibit the return of a field's setting by setting the argument corresponding to the field to a null pointer.

When an application calls *SQLGetDescRec*() to retrieve the value of a field that is undefined for a particular descriptor type, the function returns SQLSTATE HY091 (Invalid descriptor field identifier). When an application calls *SQLGetDescRec*() to retrieve the value of a field that is defined for a particular descriptor type, but has no default value and has not been set yet, the function returns SQL_SUCCESS but the value returned for the field is undefined. For more information, see **Initialization of Descriptor Fields** on page 467.

The values of all fields can be retrieved individually by a call to *SQLGetDescField*(). For a description of the fields in a descriptor header or record, see the *SQLSetDescField*() section. For more information on descriptors, see Chapter 13.

SEE ALSO

For information about	**See**

12798 Setting multiple descriptor fields	*SQLSetDescRec*()
12799 Getting a descriptor field	*SQLGetDescField*() •
12800 Binding a column	*SQLBindCol*()
12801 Binding a parameter	*SQLBindParam*()

12802 **CHANGE HISTORY**

12803 **Version 2**

12804 Revised generally. See **Alignment with Popular Implementations** on page 2. Also see the list in
12805 **Descriptor Fields Added in Version 2** on page 483.

NAME

SQLGetDiagField — Return the current value of a field of a diagnostic data structure that contains error, warning, and status information.

SYNOPSIS

```
SQLRETURN SQLGetDiagField(
  SQLSMALLINT HandleType,
  SQLHANDLE Handle,
  SQLSMALLINT RecNumber,
  SQLSMALLINT DiagIdentifier,
  SQLPOINTER DiagInfoPtr,
  SQLSMALLINT BufferLength,
  SQLSMALLINT * StringLengthPtr);
```

ARGUMENTS

HandleType [Input]
A handle type identifier that describes the type of handle for which diagnostics are required. Must be one of the following:

SQL_HANDLE_ENV
SQL_HANDLE_DBC
SQL_HANDLE_STMT
SQL_HANDLE_DESC

Handle [Input]
A handle for the diagnostic data structure, of the type indicated by *HandleType*.

RecNumber [Input]
Indicates the status record from which the application seeks information. Status records are numbered from 1. If *DiagIdentifier* indicates any field of the diagnostics header record, *RecNumber* must be 0. If not, it should be greater than 0.

DiagIdentifier [Input]
Indicates the field of the diagnostic data structure whose value is to be returned. For more information, see **DiagIdentifier Argument** on page 360.

DiagInfoPtr [Output]
Pointer to a buffer in which to return the diagnostic information. The data type depends on the value of *DiagIdentifier*.

BufferLength [Input]
If *ValuePtr* points to data of variable length, this argument should be the length of **ValuePtr*. If what is contained in *ValuePtr* is itself a pointer, but not to data of variable length, then *BufferLength* should have the value SQL_IS_POINTER. If what is contained in *ValuePtr* is actual data of fixed length, then *BufferLength* should have the value SQL_IS_NOT_POINTER.

StringLengthPtr [Output]
Pointer to a buffer in which to return the total number of octets (excluding the number of octets required for the null terminator) available to return in **DiagInfoPtr*, for character data. If the number of octets available to return is greater than *BufferLength*, then the text in **DiagInfoPtr* is truncated to *BufferLength* minus the length of a null terminator.

RETURN VALUE

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, SQL_INVALID_HANDLE, or SQL_NO_DATA.

12852 **DIAGNOSTICS**

12853 *SQLGetDiagField*() does not post status records for itself. It uses the following return values to
12854 report the outcome of its own execution:

12855 SQL_SUCCESS
12856 The function successfully returned diagnostic information.

12857 SQL_SUCCESS_WITH_INFO
12858 **DiagInfoPtr* was too small to hold the requested diagnostic field. To determine that a
12859 truncation occurred, the application compares *BufferLength* to the actual number of octets
12860 available, which is written to **StringLengthPtr*.

12861 SQL_INVALID_HANDLE
12862 The handle indicated by *HandleType* and *Handle* was not a valid handle.

12863 SQL_ERROR
12864 One of the following occurred:

12865 • *DiagIdentifier* was not one of the valid values.

12866 • *DiagIdentifier* was SQL_DIAG_CURSOR_ROW_COUNT or SQL_DIAG_ROW_COUNT,
12867 but *Handle* was not a statement handle.

12868 • *RecNumber* was negative or 0 when *DiagIdentifier* indicated a field from a diagnostic
12869 record; or *RecNumber* was nonzero when *DiagIdentifier* indicated header information.

12870 • The value requested was a character string and *BufferLength* was less than zero.

12871 SQL_NO_DATA
12872 *RecNumber* was greater than the number of diagnostic records that existed for *Handle*. The
12873 function also returns SQL_NO_DATA for any positive *RecNumber* if there are no diagnostic
12874 records for *Handle*.

12875 **COMMENTS**

12876 An application typically calls *SQLGetDiagField*() to accomplish one of three goals:

12877 • To obtain specific error or warning information when a function call has returned
12878 SQL_ERROR or SQL_SUCCESS_WITH_INFO.

12879 • To find out the number of rows in the data source that were affected when insert, delete, or
12880 update operations were performed with a call to *SQLBulkOperations*(), *SQLExecDirect*(),
12881 *SQLExecute*(), or *SQLSetPos*() (from the SQL_DIAG_ROW_COUNT header field), or to find
12882 out the number of rows that exist in the current open cursor, if the implementation is able to
12883 provide this information (from the SQL_DIAG_CURSOR_ROW_COUNT header field).

12884 • To determine which function was executed by a call to *SQLExecDirect*() or *SQLExecute*()
12885 (from the SQL_DIAG_DYNAMIC_FUNCTION and
12886 SQL_DIAG_DYNAMIC_FUNCTION_CODE header fields).

12887 Any XDBC function can post zero or more errors each time it is called, so an application can call
12888 *SQLGetDiagField*() after any XDBC function call. *SQLGetDiagField*() retrieves only the diagnostic
12889 information most recently associated with the diagnostic data structure specified in the *Handle*
12890 argument. If the application calls another XDBC function, any diagnostic information from a
12891 previous call with the same handle is lost.

12892 An application can scan all diagnostic records by incrementing *RecNumber*, as long as
12893 *SQLGetDiagField*() returns SQL_SUCCESS. The number of status records is indicated in the
12894 SQL_DIAG_NUMBER header field. No call to *SQLGetDiagField*() modifies the diagnostics area.
12895 The application can call *SQLGetDiagField*() again at a later time to retrieve a field from a record,
12896 as long as a function other than *SQLGetDiagField*() or *SQLGetDiagRec*(), has not been called in
12897 the interim, which would post records on the same handle.

12898 An application can call *SQLGetDiagField*() to return any diagnostic field at any time, except that
12899 a call to retrieve the SQL_DIAG_CURSOR_ROW_COUNT or SQL_DIAG_ROW_COUNT fields
12900 of a handle other than a statement handle returns SQL_ERROR. If any other diagnostic field is
12901 undefined, the call to *SQLGetDiagField*() returns SQL_SUCCESS (provided no other error is
12902 encountered), and an undefined value is returned for the field.

12903 **HandleType Argument**

12904 Each handle type can have diagnostic information associated with it. The *HandleType* argument
12905 denotes the handle type of *Handle*.

12906 Some header and record fields cannot be returned for all types of handles: environment,
12907 connection, statement, and descriptor. Those handles for which a field is not applicable are
12908 indicated in the lists in **Header Fields** and **Record Fields** below.

12909 **DiagIdentifier Argument**

12910 This argument indicates the identifier of the field required from the diagnostic data structure. If
12911 *RecNumber* is greater than or equal to 1, the data in the field describes the diagnostic information
12912 returned by a function. If *RecNumber* is 0, the field is in the header of the diagnostic data
12913 structure, and therefore contains data pertaining to the function call that returned the diagnostic
12914 information, not the specific information.

12915 Additional implementation-defined fields may exist in the diagnostic data structure.

12916 **Header Fields**

12917 The following header fields can be specified as *DiagIdentifier*. The only diagnostic header fields
12918 that are defined for a descriptor handle are SQL_DIAG_NUMBER and
12919 SQL_DIAG_RETURNCODE.

12920 No implementation-specific header diagnostic field should be associated with an environment
12921 handle.

12922 SQL_DIAG_CURSOR_ROW_COUNT (Return type: SQLINTEGER)
12923 This field contains the count of rows in the cursor. It is implementation-defined whether
12924 row counts are available for various cursor types; the application can determine the level of
12925 support as described in **Detecting Cursor Capabilities with SQLGetInfo()** on page 402.
12926 The contents of this field are defined only for statement handles and only after
12927 *SQLExecDirect*(), *SQLExecute*(), or *SQLMoreResults*() has been called. A call to
12928 *SQLGetDiagField*() to obtain this information for a handle other than a statement handle
12929 returns SQL_ERROR.

12930 SQL_DIAG_DYNAMIC_FUNCTION (Return type: CHAR *)
12931 For statement handles, this is a string that describes the SQL statement that the underlying
12932 function executed (see **Values of the Dynamic Function Fields** on page 362). The contents
12933 of this field are defined only after a call to *SQLExecDirect*(), *SQLExecute*(), or
12934 *SQLMoreResults*(). For handles other than statement handles, this is an empty string.

12935 SQL_DIAG_DYNAMIC_FUNCTION_CODE (Return type: SQLINTEGER)
12936 For statement handles, this is a numeric code that describes the SQL statement that was
12937 executed by the underlying function (see **Values of the Dynamic Function Fields** on page
12938 362). The contents of this field are defined only after a call to *SQLExecDirect*(),
12939 *SQLExecute*(), or *SQLMoreResults*(). For handles other than statement handles, the value is
12940 SQL_DIAG_UNKNOWN_STATEMENT.

12941 SQL_DIAG_NUMBER (Return type: SQLINTEGER)
12942 The number of status records that are available for the specified handle.

12943 SQL_DIAG_RETURNCODE (Return type: SQLRETURN)
12944 Return code returned by the function. See Chapter 15 for a list of return codes. If no
12945 function has yet been called on *Handle*, SQL_SUCCESS is returned for
12946 SQL_DIAG_RETURNCODE.

12947 SQL_DIAG_ROW_COUNT (Return type: SQLINTEGER)
12948 For statement handles, the number of rows affected by an insert, delete, or update
12949 performed by *SQLBulkOperations*(), *SQLExecDirect*(), *SQLExecute*(), or *SQLSetPos*(). Its
12950 value is undefined after a cursor specification has been executed. The contents of this field
12951 are defined only for statement handles. The data in this field is returned in *RowCountPtr* of
12952 *SQLRowCount*(). A call to *SQLGetDiagField*() to obtain this information for a handle other
12953 than a statement handle returns SQL_ERROR.

12954 The data in this field is reset after every function call, whereas the row count returned by
12955 *SQLRowCount*() remains the same until the statement is set back to the prepared or
12956 allocated state.

12957 **Record Fields**

12958 The following record fields can be specified as *DiagIdentifier*:

12959 SQL_DIAG_CLASS_ORIGIN (Return type: CHAR *)
12960 A string that indicates the document that defines the class and subclass portion of the
12961 SQLSTATE value in this record. Its value is 'ISO 9075' for all SQLSTATEs defined by the ISO
12962 CLI International Standard. For other SQLSTATEs defined in this specification, its value is
12963 'XDBC'.

12964 SQL_DIAG_COLUMN_NUMBER (Return type: SQLINTEGER)
12965 If the SQL_DIAG_ROW_NUMBER field is a valid row or parameter number, then this field
12966 contains the value that represents the column number in the result set. Result set column
12967 numbers always start at 1; if this status record pertains to a bookmark column, then the field
12968 can be zero. It has the value SQL_NO_COLUMN_NUMBER if the status record is not
12969 associated with a column number. If the implementation cannot determine the column
12970 number that this record is associated with, this field has the value
12971 SQL_COLUMN_NUMBER_UNKNOWN. The contents of this field are defined only for
12972 statement handles.

12973 SQL_DIAG_CONNECTION_NAME (Return type: CHAR *)
12974 A string that indicates the name of the connection that the diagnostic record relates to. For
12975 diagnostic data structures associated with the environment handle and for diagnostics that
12976 do not relate to any server, this field is a zero-length string.

12977 SQL_DIAG_MESSAGE_TEXT (Return type: CHAR *)
12978 An informational message on the error or warning. This field is formatted as described in
12979 Chapter 15.

12980 SQL_DIAG_NATIVE (Return type: SQLINTEGER)
12981 An implementation-defined native error code. If there is no native error code, this is 0.

12982 SQL_DIAG_ROW_NUMBER (Return type: SQLINTEGER)
12983 This field contains the row or parameter number in the row-set or set of parameters with
12984 which the status record is associated. This field has the value SQL_NO_ROW_NUMBER if
12985 this status record is not associated with a row number. If the associated row cannot be
12986 determined, this field has the value SQL_ROW_NUMBER_UNKNOWN.

12987 The contents of this field are defined only for statement handles.

12988 SQL_DIAG_SERVER_NAME (Return type: CHAR *)
12989 A string that indicates the server name that the diagnostic record relates to. It is the same as

12990 the value returned for a call to *SQLGetInfo*() with the SQL_DATA_SOURCE_NAME option.
12991 For diagnostic data structures associated with the environment handle and for diagnostics
12992 that do not relate to any server, this field is a zero-length string.

12993 SQL_DIAG_SQLSTATE (Return type: CHAR *)
12994 A five-character SQLSTATE diagnostic code.

12995 SQL_DIAG_SUBCLASS_ORIGIN (Return type: CHAR *)
12996 A string with the same format and valid values as SQL_DIAG_CLASS_ORIGIN, that
12997 identifies the defining portion of the subclass portion of the SQLSTATE code.

12998 **Values of the Dynamic Function Fields**

12999 The following table describes the values of SQL_DIAG_DYNAMIC_FUNCTION and
13000 SQL_DIAG_DYNAMIC_FUNCTION_CODE that apply to each type of SQL statement executed
13001 by a call to *SQLExecute*() or *SQLExecDirect*(). Implementation-defined values may also exist.

	SQL statement Executed	Value of SQL_DIAG_ DYNAMIC_FUNCTION	Value of SQL_DIAG_DYNAMIC_ FUNCTION_CODE
13002–13003			
13004	*alter-domain-statement*	"ALTER DOMAIN"	SQL_DIAG_ALTER_DOMAIN
13005	*alter-table-statement*	"ALTER TABLE"	SQL_DIAG_ALTER_TABLE
13006	*assertion-definition*	"CREATE ASSERTION"	SQL_DIAG_CREATE_ASSERTION
13007	*character-set-definition*	"CREATE CHARACTER SET"	SQL_DIAG_CREATE_CHARACTER_SET
13008	*collation-definition*	"CREATE COLLATION"	SQL_DIAG_CREATE_COLLATION
13009	*create-index-statement*	"CREATE INDEX"	SQL_DIAG_CREATE_INDEX
13010	*create-table-statement*	"CREATE TABLE"	SQL_DIAG_CREATE_TABLE
13011	*create-view-statement*	"CREATE VIEW"	SQL_DIAG_CREATE_VIEW
13012	*cursor-specification*	"SELECT CURSOR"	SQL_DIAG_SELECT_CURSOR
13013	*delete-statement-positioned*	"DYNAMIC DELETE CURSOR"	SQL_DIAG_DYNAMIC_DELETE_CURSOR
13014	*delete-statement-searched*	"DELETE WHERE"	SQL_DIAG_DELETE_WHERE
13015	*domain-definition*	"CREATE DOMAIN"	SQL_DIAG_CREATE_DOMAIN
13016	*drop-assertion-statement*	"DROP ASSERTION"	SQL_DIAG_DROP_ASSERTION
13017	*drop-character-set-stmt*	"DROP CHARACTER SET"	SQL_DIAG_DROP_CHARACTER_SET
13018	*drop-collation-statement*	"DROP COLLATION"	SQL_DIAG_DROP_COLLATION
13019	*drop-domain-statement*	"DROP DOMAIN"	SQL_DIAG_DROP_DOMAIN
13020	*drop-index-statement*	"DROP INDEX"	SQL_DIAG_DROP_INDEX
13021	*drop-schema-statement*	"DROP SCHEMA"	SQL_DIAG_DROP_SCHEMA
13022	*drop-table-statement*	"DROP TABLE"	SQL_DIAG_DROP_TABLE
13023	*drop-translation-statement*	"DROP TRANSLATION"	SQL_DIAG_DROP_TRANSLATION
13024	*drop-view-statement*	"DROP VIEW"	SQL_DIAG_DROP_VIEW
13025	*grant-statement*	"GRANT"	SQL_DIAG_GRANT
13026	*insert-statement*	"INSERT"	SQL_DIAG_INSERT
13027	*XDBC-procedure-extension*	"CALL"	SQL_DIAG_PROCEDURE_CALL
13028	*revoke-statement*	"REVOKE"	SQL_DIAG_REVOKE
13029	*schema-definition*	"CREATE SCHEMA"	SQL_DIAG_CREATE_SCHEMA
13030	*translation-definition*	"CREATE TRANSLATION"	SQL_DIAG_CREATE_TRANSLATION
13031	*update-statement-positioned*	"DYNAMIC UPDATE CURSOR"	SQL_DIAG_DYNAMIC_UPDATE_CURSOR
13032	*update-statement-searched*	"UPDATE WHERE"	SQL_DIAG_UPDATE_WHERE
13033	Unknown	*empty string*	SQL_DIAG_UNKNOWN_STATEMENT

13034 **Sequence of Status Records**

13035 Within a row, records are ranked according to rules stated in **Sequence of Status Records** on
13036 page 196. These rules provide that records are sorted according to the row number to which
13037 they pertain. Within a row, errors outrank warnings and that standard diagnostics outrank
13038 implementation-defined diagnostics.

13039 **SEE ALSO**

13040 For information about	See
13041 Obtaining multiple fields of a diagnostic data structure	*SQLGetDiagRec*()

13042 **CHANGE HISTORY**

13043 **Version 2**

13044 Revised generally. See **Alignment with Popular Implementations** on page 2.

13045 **NAME**

13046 SQLGetDiagRec — Return the current values of multiple fields of a diagnostic record.

13047 **SYNOPSIS**

```
13048 SQLRETURN SQLGetDiagRec(
13049   SQLSMALLINT HandleType,
13050   SQLHANDLE Handle,
13051   SQLSMALLINT RecNumber,
13052   SQLCHAR * Sqlstate,
13053   SQLINTEGER * NativeErrorPtr,
13054   SQLCHAR * MessageText,
13055   SQLSMALLINT BufferLength,
13056   SQLSMALLINT * TextLengthPtr);
```

13057 **ARGUMENTS**

13058 *HandleType* [Input]

13059 A handle type identifier that describes the type of handle for which diagnostics are required.
13060 Must be one of the following:

13061 SQL_HANDLE_ENV
13062 SQL_HANDLE_DBC
13063 SQL_HANDLE_STMT
13064 SQL_HANDLE_DESC

13065 *Handle* [Input]

13066 A handle for the diagnostic data structure, of the type indicated by *HandleType*.

13067 *RecNumber* [Input]

13068 Indicates the status record from which the application seeks information. Status records are
13069 numbered from 1.

13070 *SQLState* [Output]

13071 Pointer to a buffer in which to return a five-character SQLSTATE code pertaining to the
13072 diagnostic record *RecNumber*. The first two characters indicate the class; the next three
13073 indicate the subclass.

13074 *NativeErrorPtr* [Output]

13075 Pointer to a buffer in which to return the native error code, specific to the data source.

13076 *MessageText* [Output]

13077 Pointer to a buffer in which to return the error message text. The fields returned by
13078 *SQLGetDiagRec*() are contained in a text string. For the format of the string, see Section
13079 15.3.0 on page 198.

13080 *BufferLength* [Input]

13081 Length (in octets) of the *MessageText buffer.

13082 *TextLengthPtr* [Output]

13083 Pointer to a buffer in which to return the total number of octets (excluding the number of
13084 octets required for the null terminator) available to return in *MessageText. If the number of
13085 octets available to return is greater than *BufferLength*, then the error message text in
13086 *MessageText is truncated to *BufferLength* minus the length of a null terminator.

13087 **RETURN VALUE**

13088 SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.

13089 **DIAGNOSTICS**

13090 *SQLGetDiagRec*() does not post status records for itself. It uses the following return values to
13091 report the outcome of its own execution:

SQL_SUCCESS
The function successfully returned diagnostic information.

SQL_SUCCESS_WITH_INFO
The *MessageText buffer was too small to hold the requested error message. No diagnostic records were generated. To determine that a truncation occurred, the application compares *BufferLength* to the actual number of octets available, which is written to **StringLengthPtr*.

SQL_INVALID_HANDLE
The handle indicated by *HandleType* and *Handle* was not a valid handle.

SQL_ERROR
One of the following occurred:

- *RecNumber* was negative or 0.
- The value requested was a character string and *BufferLength* was less than zero.

SQL_NO_DATA
RecNumber was greater than the number of diagnostic records that existed for *Handle*. The function also returns SQL_NO_DATA for any positive *RecNumber* if there are no diagnostic records for *Handle*.

COMMENTS

Unlike *SQLGetDiagField*(), which returns one diagnostic field per call, *SQLGetDiagRec*() returns several commonly-used fields of a diagnostic record, including the SQLSTATE, the native error code, and the error message text.

An application typically calls *SQLGetDiagRec*() when a previous call to an XDBC function has returned SQL_SUCCESS or SQL_SUCCESS_WITH_INFO. However, because any XDBC function can post zero or more errors each time it is called, an application can call *SQLGetDiagRec*() after any XDBC function call. An application can call *SQLGetDiagRec*() multiple times to return some or all of the records in the diagnostic data structure.

SQLGetDiagRec() returns a character string containing multiple fields of the diagnostic data structure record. The form of the error message string is described in Chapter 15.

SQLGetDiagRec() cannot be used to return fields from the header of the diagnostic data structure (*RecNumber* must be greater than 0). The application should call *SQLGetDiagField*() for this purpose.

SQLGetDiagRec() retrieves only the diagnostic information most recently associated with *Handle*. If the application calls another XDBC function, except *SQLGetDiagRec*() or *SQLGetDiagField*(), any diagnostic information from the previous calls on the same handle is lost.

An application can scan all diagnostic records by looping, incrementing *RecNumber*, as long as *SQLGetDiagRec*() returns SQL_SUCCESS. Calls to *SQLGetDiagRec*() are non-destructive to the header and record fields. The application can call *SQLGetDiagRec*() again at a later time to retrieve a field from a record, as long as no other function, except *SQLGetDiagRec*() or *SQLGetDiagField*() has been called in the interim. The application can also retrieve a count of the total number of diagnostic records available by calling *SQLGetDiagField*() to retrieve the value of the SQL_DIAG_NUMBER field, then call *SQLGetDiagRec*() that many times.

For a description of the fields of the diagnostic data structure, see *SQLGetDiagField*().

HandleType Argument

Each handle type can have diagnostic information associated with it. *HandleType* denotes the handle type of *Handle*.

13136 Some header and record fields cannot be returned for all types of handles: environment,
13137 connection, statement, and descriptor. Those handles for which a field is not applicable are
13138 indicated in the list of fields in *SQLGetDiagField*().

13139 Descriptor handles can also have diagnostic information associated with them. These diagnostic
13140 data structure contain information on errors or warnings that occur when a function is called
13141 with a descriptor handle.

13142 **SEE ALSO**

13143	**For information about**	**See**
13144 13145	Obtaining field of a diagnostic record or a field of the diagnostic header	*SQLGetDiagField*()

13146 **CHANGE HISTORY**

13147 **Version 2**

13148 Revised generally. See **Alignment with Popular Implementations** on page 2.

13149 **NAME**
13150 SQLGetEnvAttr — Return the current setting of an environment attribute.

13151 **SYNOPSIS**

```
13152 SQLRETURN SQLGetEnvAttr(
13153   SQLHENV EnvironmentHandle,
13154   SQLINTEGER Attribute,
13155   SQLPOINTER ValuePtr,
13156   SQLINTEGER BufferLength,
13157   SQLINTEGER * StringLengthPtr);
```

13158 **ARGUMENTS**

13159 *EnvironmentHandle* [Input]
13160 Environment handle.

13161 *Attribute* [Input]
13162 Attribute to retrieve.

13163 *ValuePtr* [Output]
13164 Pointer to a buffer in which to return the current value of the attribute specified by *Attribute*.

13165 *BufferLength* [Input]
13166 If *ValuePtr* points to data of variable length, this argument should be the length of **ValuePtr*.

13167 *StringLengthPtr* [Output]
13168 A pointer to a buffer in which to return the total number of octets (excluding the null
13169 terminator) available to return in **ValuePtr*. If *ValuePtr* is a null pointer, no length is
13170 returned. If the attribute value is a character string, and the number of octets available to
13171 return is greater than or equal to *BufferLength*, the data in **ValuePtr* is truncated to
13172 *BufferLength* minus the length of a null terminator and is null-terminated.

13173 **RETURN VALUE**
13174 SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NO_DATA, SQL_ERROR, or
13175 SQL_INVALID_HANDLE.

13176 **DIAGNOSTICS**
13177 When *SQLGetEnvAttr*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated
13178 SQLSTATE value can be obtained by calling *SQLGetDiagRec*() with a *HandleType* of
13179 SQL_HANDLE_ENV and a *Handle* of *EnvironmentHandle*. The following SQLSTATE values are
13180 commonly returned by *SQLGetEnvAttr*(). The return code associated with each SQLSTATE
13181 value is SQL_ERROR, except that for SQLSTATE values in class 01, the return code is
13182 SQL_SUCCESS_WITH_INFO.

13183 01000 — General warning
13184 Implementation-defined informational message.

13185 01004 — String data, right truncation
13186 The data returned in **ValuePtr* was truncated to be *BufferLength* minus the null terminator.
13187 The length of the untruncated string value is returned in **StringLengthPtr*.

13188 HY000 — General error
13189 An error occurred for which there was no specific SQLSTATE and for which no
13190 implementation-specific SQLSTATE was defined. The error message returned by
13191 *SQLGetDiagRec*() in the **MessageText* buffer describes the error and its cause.

13192 HY001 — Memory allocation error
13193 The implementation failed to allocate memory required to support execution or completion
13194 of the function.

13195 HY092 — Invalid attribute identifier
13196 *Attribute* was not valid for this connection to this data source.

13197 HYC00 — Optional feature not implemented
13198 *Attribute* was a valid environment attribute but the data source does not support it.

13199 HYT01 — Connection timeout expired
13200 The connection timeout period expired before the data source responded to the request. The
13201 connection timeout period is set through *SQLSetConnectAttr*(),
13202 SQL_ATTR_CONNECTION_TIMEOUT.

13203 IM001 — Function not supported
13204 The function is not supported on the current connection to the data source.

13205 **COMMENTS**

13206 For a list of options, see *SQLSetEnvAttr*(). If *Attribute* specifies an attribute that returns a string,
13207 *ValuePtr* points to a buffer in which to return the string. The maximum length of the string,
13208 including the null terminator, is *BufferLength* octets.

13209 *SQLGetEnvAttr*() can be called at any time between the allocation and the freeing of an
13210 environment handle. All environment attributes successfully set by the application for the
13211 environment persist until *SQLFreeHandle*() is called on the *EnvironmentHandle* with a *HandleType*
13212 of SQL_HANDLE_ENV. More than one environment handle can be allocated simultaneously.
13213 An environment attribute on one environment is not affected when another environment is
13214 allocated.

13215 **SEE ALSO**

	For information about	**See**
13216		
13217	Returning the setting of a connection attribute	*SQLGetConnectAttr*()
13218	Returning the setting of a statement attribute	*SQLGetStmtAttr*()
13219	Setting a connection attribute	*SQLSetConnectAttr*()
13220	Setting an environment attribute	*SQLSetEnvAttr*()
13221	Setting a statement attribute	*SQLSetStmtAttr*()

13222 **CHANGE HISTORY**

13223 **Version 2**

13224 Revised generally. See **Alignment with Popular Implementations** on page 2.

13225 **NAME**

13226 SQLGetFunctions — Indicate the level of support for a specified XDBC function.

13227 **SYNOPSIS**

```
13228 SQLRETURN SQLGetFunctions(
13229   SQLHDBC ConnectionHandle,
13230   SQLUSMALLINT FunctionId,
13231   SQLUSMALLINT * SupportedPtr);
```

13232 **ARGUMENTS**

13233 *ConnectionHandle* [Input]

13234 Connection handle.

13235 *FunctionId* [Input]

13236 A **#define** value that identifies the XDBC function of interest, or
13237 SQL_API_XDBC_ALL_FUNCTIONS. For a list of **#define** values that identify XDBC
13238 functions, see the tables in ‘‘Comments.’’

13239 *SupportedPtr* [Output]

13240 If *FunctionId* identifies a single XDBC function, *SupportedPtr* points to a single
13241 SQLUSMALLINT value that is SQL_TRUE if the specified function is supported by the data
13242 source, and SQL_FALSE if it is not supported.

13243 If *FunctionId* is SQL_API_XDBC_ALL_FUNCTIONS, the application must point
13244 *SupportedPtr* to a SQLSMALLINT array with a number of elements equal to
13245 SQL_API_XDBC_ALL_FUNCTIONS_SIZE. This array is a bitmap that indicates whether an
13246 XDBC function is supported. The application can call the SQL_FUNC_EXISTS() macro to
13247 determine if a specific function is supported (see ‘‘Comments’’).

13248 The arrays returned in **SupportedPtr* use zero-based indexing.

13249 **RETURN VALUE**

13250 SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.

13251 **DIAGNOSTICS**

13252 When *SQLGetFunctions*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated
13253 SQLSTATE value can be obtained by calling *SQLGetDiagRec*() with a *HandleType* of
13254 SQL_HANDLE_DBC and a *Handle* of *ConnectionHandle*. The following SQLSTATE values are
13255 commonly returned by *SQLGetFunctions*(). The return code associated with each SQLSTATE
13256 value is SQL_ERROR, except that for SQLSTATE values in class 01, the return code is
13257 SQL_SUCCESS_WITH_INFO.

13258 01000 — General warning

13259 Implementation-defined informational message.

13260 08S01 — Communication link failure

13261 The communication link to the data source failed before the function completed processing.

13262 HY000 — General error

13263 An error occurred for which there was no specific SQLSTATE and for which no
13264 implementation-specific SQLSTATE was defined. The error message returned by
13265 *SQLGetDiagRec*() in the **MessageText* buffer describes the error and its cause.

13266 HY001 — Memory allocation error

13267 The implementation failed to allocate memory required to support execution or completion
13268 of the function.

13269 HY010 — Function sequence error

13270 The function was called before *SQLConnect*(), *SQLBrowseConnect*(), or *SQLDriverConnect.*()

13271 *SQLBrowseConnect*() was called for *ConnectionHandle* and returned SQL_NEED_DATA. This
13272 function was called before *SQLBrowseConnect*() returned SQL_SUCCESS_WITH_INFO or
13273 SQL_SUCCESS.

13274 HY095 — Function type out of range
13275 *FunctionId* was invalid.

13276 HYT01 — Connection timeout expired
13277 The connection timeout period expired before the data source responded to the request. The
13278 connection timeout period is set through *SQLSetConnectAttr*(),
13279 SQL_ATTR_CONNECTION_TIMEOUT.

13280 **COMMENTS**

13281 *SQLGetFunctions*() determines whether an XDBC function specified by *FunctionId* is supported
13282 for use on *ConnectionHandle*. If so, it sets *SupportedPtr* to SQL_TRUE; if not, it sets *SupportedPtr*
13283 to SQL_FALSE.

13284 The manifest constant for use as *FunctionId* is the name of the function (including its SQL prefix),
13285 with the additional prefix ‘‘SQL_API_’’.

13286 *SQLGetFunctions*() always reports that the following functions are supported:

13287 SQL_API_SQLDATASOURCES SQL_API_SQLGETFUNCTIONS
13288 SQL_API_SQLDRIVERS OP

13289 In implementations that comply with the ISO CLI International Standard, *SQLGetFunctions*()
13290 reports that at least the following additional functions are supported:

13291 SQL_API_SQLALLOCHANDLE SQL_API_SQLGETDESCREC
13292 SQL_API_SQLBINDCOL SQL_API_SQLGETDIAGFIELD
13293 SQL_API_SQLCANCEL SQL_API_SQLGETDIAGREC
13294 SQL_API_SQLCLOSECURSOR SQL_API_SQLGETENVATTR
13295 SQL_API_SQLCOLATTRIBUTE SQL_API_SQLGETINFO
13296 SQL_API_SQLCONNECT SQL_API_SQLGETSTMTATTR
13297 SQL_API_SQLCOPYDESC SQL_API_SQLGETTYPEINFO
13298 SQL_API_SQLDESCRIBECOL SQL_API_SQLNUMRESULTCOLS
13299 SQL_API_SQLDISCONNECT SQL_API_SQLPARAMDATA
13300 SQL_API_SQLENDTRAN SQL_API_SQLPREPARE
13301 SQL_API_SQLEXECDIRECT SQL_API_SQLPUTDATA
13302 SQL_API_SQLEXECUTE SQL_API_SQLROWCOUNT
13303 SQL_API_SQLFETCH SQL_API_SQLSETCONNECTATTR
13304 SQL_API_SQLFETCHSCROLL SQL_API_SQLSETCURSORNAME
13305 SQL_API_SQLFREEHANDLE SQL_API_SQLSETDESCFIELD
13306 SQL_API_SQLGETCONNECTATTR SQL_API_SQLSETDESCREC
13307 SQL_API_SQLGETCURSORNAME SQL_API_SQLSETENVATTR
13308 SQL_API_SQLGETDATA SQL_API_SQLSETSTMTATTR
13309 SQL_API_SQLGETDESCFIELD

13310 In implementations that comply with the X/Open **CLI** specification (1995), *SQLGetFunctions*()
13311 reports that at least the following additional functions are supported:

13312 SQL_API_SQLCOLUMNS SQL_API_SQLSTATISTICS

13313 SQL_API_SQLSPECIALCOLUMNS SQL_API_SQLTABLES

13314 In implementations that fully comply with the present X/Open specification, *SQLGetFunctions*()
13315 reports that at least the following additional functions are supported:

SQL_API_SQLBINDPARAMETER	SQL_API_SQLNATIVESQL
SQL_API_SQLBROWSECONNECT	SQL_API_SQLNUMPARAMS
SQL_API_SQLBULKOPERATIONS	SQL_API_SQLPRIMARYKEYS
SQL_API_SQLCOLUMNPRIVILEGES	SQL_API_SQLPROCEDURECOLUMNS
SQL_API_SQLDESCRIBEPARAM	SQL_API_SQLPROCEDURES
SQL_API_SQLDRIVERCONNECT	SQL_API_SQLSETPOS
SQL_API_SQLFOREIGNKEYS	SQL_API_SQLTABLEPRIVILEGES
SQL_API_SQLMORERESULTS	

13324 **SQL_FUNC_EXISTS() Macro**

13325 The SQL_FUNC_EXISTS(*lpbFuncExists*, *nwIndex*) macro is used to determine support for
13326 functions after *SQLGetFunctions*() has been called with an *FunctionId* argument of
13327 SQL_API_XDBC_ALL_FUNCTIONS. The application calls SQL_FUNC_EXISTS() with the
13328 *lpbFuncExists* argument set to the bitmap pointed to by the value returned in **SupportedPtr*, and
13329 with the *nwIndex* argument set to the **#define** for the function. SQL_FUNC_EXISTS() returns
13330 SQL_TRUE if the function is supported, and SQL_FALSE otherwise.

13331 SEE ALSO

For information about	See
Returning the setting of a connection attribute	*SQLGetConnectAttr*()
Returning information about an implementation	*SQLGetInfo*()
Returning the setting of a statement attribute	*SQLGetStmtAttr*()

13336 CHANGE HISTORY

13337 Version 2

13338 Revised generally. See **Alignment with Popular Implementations** on page 2.

13339 **NAME**

13340 SQLGetInfo — Return general information about the data source and the connection to it.

13341 **SYNOPSIS**

```
13342 SQLRETURN SQLGetInfo(
13343   SQLHDBC ConnectionHandle,
13344   SQLSMALLINT InfoType,
13345   SQLPOINTER InfoValuePtr,
13346   SQLSMALLINT BufferLength,
13347   SQLSMALLINT * StringLengthPtr);
```

13348 **ARGUMENTS**

13349 *ConnectionHandle* [Input]

13350 Connection handle.

13351 *InfoType* [Input]

13352 Type of information.

13353 *InfoValuePtr* [Output]

13354 Pointer to a buffer in which to return the information. Depending on *InfoType*, the
13355 information returned is either a null-terminated character string, a SQLSMALLINT value, a
13356 SQLINTEGER bitmask, a SQLINTEGER flag, or a 32-bit binary value.

13357 *BufferLength* [Input]

13358 Length of the **InfoValuePtr* buffer. If the value in **InfoType* is not a character string, or if
13359 *InfoType* is a null pointer, *BufferLength* is ignored. **InfoValuePtr* is assumed to be 16 bits or 32
13360 bits, based on *InfoType*.

13361 *StringLengthPtr* [Output]

13362 Pointer to a buffer in which to return the total number of octets (excluding the null
13363 terminator for character data) available to return in **InfoValuePtr*.

13364 For character data, if the number of octets available to return is greater than or equal to
13365 *BufferLength*, the information in **InfoValuePtr* is truncated to *BufferLength* octets minus the
13366 length of a null terminator and is null-terminated.

13367 For all other types of data, the value of *BufferLength* is ignored and **InfoValuePtr* is assumed
13368 to be 16 bits or 32 bits, depending on *InfoType*.

13369 **RETURN VALUE**

13370 SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.

13371 **DIAGNOSTICS**

13372 When *SQLGetInfo*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated
13373 SQLSTATE value can be obtained by calling *SQLGetDiagRec*() with a *HandleType* of
13374 SQL_HANDLE_DBC and a *Handle* of *ConnectionHandle*. The following SQLSTATE values are
13375 commonly returned by *SQLGetInfo*(). The return code associated with each SQLSTATE value is
13376 SQL_ERROR, except that for SQLSTATE values in class 01, the return code is
13377 SQL_SUCCESS_WITH_INFO.

13378 01000 — General warning

13379 Implementation-defined informational message.

13380 01004 — String data, right truncation

13381 The buffer **InfoValuePtr* was not large enough to return all of the requested information, so
13382 the information was truncated. The length of the requested information in its untruncated
13383 form is returned in **StringLengthPtr*.

13384 08003 — Connection does not exist
13385 *InfoType* specifies an option that requires an open connection. Of the XDBC-defined options,
13386 only SQL_XDBC_VER can be returned without an open connection.

13387 08S01 — Communication link failure
13388 The communication link to the data source failed before the function completed processing.

13389 HY000 — General error
13390 An error occurred for which there was no specific SQLSTATE and for which no
13391 implementation-specific SQLSTATE was defined. The error message returned by
13392 *SQLGetDiagRec*() in the *MessageText* buffer describes the error and its cause.

13393 HY001 — Memory allocation error
13394 The implementation failed to allocate memory required to support execution or completion
13395 of the function.

13396 HY090 — Invalid string or buffer length
13397 *BufferLength* was less than 0. •

13398 HY096 — Information type out of range
13399 *InfoType* is in the range of numbers defined by XDBC and the implementation does not
13400 support it.

13401 HYC00 — Optional feature not implemented
13402 *InfoType* is a valid value but is not supported by the data source. (This can occur when an
13403 XDBC application makes certain requests of a data source that complies with the ISO CLI
13404 International Standard or the March 1995 issue of this specification.)

13405 HYT01 — Connection timeout expired
13406 The connection timeout period expired before the data source responded to the request. The
13407 connection timeout period is set through *SQLSetConnectAttr*(),
13408 SQL_ATTR_CONNECTION_TIMEOUT.

13409 IM001 — Function not supported
13410 The function is not supported on the current connection to the data source.

13411 **COMMENTS**

13412 The currently-defined options are listed in **Information Type Descriptions** on page 377.
13413 Additional types may be defined in the future for both standard and implementation-defined
13414 information requests. Vendors must reserve values for proprietary requests from X/Open (see
13415 Section 1.8 on page 21).

13416 The format of the information returned in **InfoValuePtr* depends on the *InfoType* requested.
13417 *SQLGetInfo*() returns information in one of five different formats:

13418 • A null-terminated character string
13419 • An SQLUSMALLINT value
13420 • An SQLUINTEGER bitmask
13421 • An SQLUINTEGER value
13422 • A 32-bit binary value.

13423 The format of each of the following options is noted in the type's description. The application
13424 must cast the value returned in **InfoValuePtr* accordingly.

13425 The implementation must return a value for each of the options defined in the following tables.
13426 If *InfoType* is not applicable, then the implementation returns the following:

Format of *InfoValuePtr	Returned Value
Character string (''Y'' or ''N'')	''N''
Character string (not ''Y'' or ''N'')	Empty string
SQLUSMALLINT	0
SQLUINTEGER bitmask or 32-bit binary value	0L

For example, if a data source does not support procedures, *SQLGetInfo*() returns the following values for the values of *InfoType* that relate to procedures:

InfoType	Returned value
SQL_PROCEDURES	''N''
SQL_ACCESSIBLE_PROCEDURES	''N''
SQL_MAX_PROCEDURE_NAME_LEN	0
SQL_PROCEDURE_TERM	Empty string

The SQLSTATE values HY096 (Invalid argument value) and HYC00 (Optional feature not implemented) both report that the implementation does not support the specified *InfoType*. HY096 is used for XDBC-defined values and HYC00 is used for values in the implementation-defined range.

GetInfo() Options

This section lists the options XDBC defines for use with *SQLGetInfo*(). Information types are grouped by category. Following these tables, every XDBC-defined type is listed alphabetically.

Implementation Information

The following values of *InfoType* return information about the implementation, such as the number of active statements, the data source name, and compliance with X/Open specifications and standards:

SQL_ACTIVE_ENVIRONMENTS
SQL_ASYNC_MODE
SQL_BATCH_ROW_COUNT
SQL_BATCH_SUPPORT
SQL_DATA_SOURCE_NAME
SQL_DYNAMIC_CURSOR_ATTRIBUTES1
SQL_DYNAMIC_CURSOR_ATTRIBUTES2
SQL_FILE_USAGE
SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES1
SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES2
SQL_GETDATA_EXTENSIONS
SQL_INFO_SCHEMA_VIEWS
SQL_KEYSET_CURSOR_ATTRIBUTES1
SQL_KEYSET_CURSOR_ATTRIBUTES2
SQL_MAX_ASYNC_CONCURRENT_STATEMENTS
SQL_MAX_CONCURRENT_ACTIVITIES
SQL_MAX_DRIVER_CONNECTIONS
SQL_PARAM_ARRAY_ROW_COUNTS
SQL_PARAM_ARRAY_SELECTS
SQL_ROW_UPDATES
SQL_SEARCH_PATTERN_ESCAPE
SQL_SERVER_NAME
SQL_STATIC_CURSOR_CAPABILITIES1
SQL_STATIC_CURSOR_CAPABILITIES2
SQL_XDBC_INTERFACE_CONFORMANCE
SQL_XDBC_STANDARD_CLI_CONFORMANCE
SQL_XDBC_VER

13464 **Data Source Product Information**

13465 The following values of *InfoType* return product information about the data source, such as the
13466 vendor's product name and version:

13467 SQL_DATABASE_NAME SQL_DBMS_NAME SQL_DBMS_VER

13468 **Data Source Information**

13469 The following values of *InfoType* return information about the data source, such as cursor
13470 characteristics and transaction capabilities:

13471 SQL_ACCESSIBLE_PROCEDURES
13472 SQL_ACCESSIBLE_TABLES
13473 SQL_BOOKMARK_PERSISTENCE
13474 SQL_CATALOG_TERM
13475 SQL_COLLATION_SEQ
13476 SQL_CONCAT_NULL_BEHAVIOR
13477 SQL_CURSOR_COMMIT_BEHAVIOR
13478 SQL_CURSOR_ROLLBACK_BEHAVIOR
13479 SQL_CURSOR_SENSITIVITY
13480 SQL_DATA_SOURCE_READ_ONLY
13481 SQL_DEFAULT_TXN_ISOLATION
13482 SQL_DESCRIBE_PARAMETER
13483 SQL_DYNAMIC_CURSOR_ATTRIBUTES2
13484 SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES2
SQL_KEYSET_CURSOR_ATTRIBUTES2
SQL_MULTIPLE_ACTIVE_TXN
SQL_MULT_RESULT_SETS
SQL_NEED_LONG_DATA_LEN
SQL_NULL_COLLATION
SQL_PROCEDURE_TERM
SQL_SCHEMA_TERM
SQL_SCROLL_OPTIONS
SQL_STATIC_CURSOR_CAPABILITIES2
SQL_TABLE_TERM
SQL_TXN_CAPABLE
SQL_TXN_ISOLATION_OPTION
SQL_USER_NAME

13485 **Supported SQL**

13486 The following values of *InfoType* return information about the dialect of SQL that can be used on
13487 the connection. This information does not specify the entire SQL grammar; it simply describes
13488 aspects of SQL for which implementations support differently.

13489 SQL_ALTER_DOMAIN
13490 SQL_ALTER_SCHEMA
13491 SQL_ALTER_TABLE
13492 SQL_ANSI_SQL_CONFORMANCE
13493 SQL_ANSI_SQL_DATETIME_LITERALS
13494 SQL_CATALOG_LOCATION
13495 SQL_CATALOG_NAME
13496 SQL_CATALOG_NAME_SEPARATOR
13497 SQL_CATALOG_USAGE
13498 SQL_COLUMN_ALIAS
13499 SQL_CORRELATION_NAME
13500 SQL_CREATE_ASSERTION
13501 SQL_CREATE_CHARACTER_SET
13502 SQL_CREATE_COLLATION
13503 SQL_CREATE_DOMAIN
13504 SQL_CREATE_SCHEMA
13505 SQL_CREATE_TABLE
13506 SQL_CREATE_TRANSLATION
13507 SQL_DROP_ASSERTION
13508 SQL_DROP_CHARACTER_SET
13509 SQL_DROP_COLLATION
13510 SQL_DROP_DOMAIN
13511 SQL_DROP_SCHEMA
13512 SQL_DROP_TABLE
13513 SQL_DROP_TRANSLATION
SQL_DROP_VIEW
SQL_DYNAMIC_CURSOR_ATTRIBUTES1
SQL_EXPRESSIONS_IN_ORDERBY
SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES1
SQL_GROUP_BY
SQL_IDENTIFIER_CASE
SQL_IDENTIFIER_QUOTE_CHAR
SQL_INDEX_KEYWORDS
SQL_KEYSET_CURSOR_ATTRIBUTES1
SQL_KEYWORDS
SQL_LIKE_ESCAPE_CLAUSE
SQL_NON_NULLABLE_COLUMNS
SQL_OJ_CAPABILITIES
SQL_ORDER_BY_COLUMNS_IN_SELECT
SQL_OUTER_JOINS
SQL_PROCEDURES
SQL_QUOTED_IDENTIFIER_CASE
SQL_REVOKE
SQL_SCHEMA_USAGE
SQL_SPECIAL_CHARACTERS
SQL_STATIC_CURSOR_CAPABILITIES1
SQL_SUBQUERIES
SQL_UNION
SQL_XDBC_SQL_OPT_IEF

13514 **SQL Limits**

13515 The following values of *InfoType* return information about the limits applied to identifiers and
13516 clauses in SQL statements, such as the maximum lengths of identifiers and the maximum
13517 number of columns in a select list. Limitations may be imposed by either the data source or
13518 software that implements the connection to it.

13519	SQL_MAX_BINARY_LITERAL_LEN	SQL_MAX_IDENTIFIER_LEN
13520	SQL_MAX_CATALOG_NAME_LEN	SQL_MAX_INDEX_SIZE
13521	SQL_MAX_CHAR_LITERAL_LEN	SQL_MAX_PROCEDURE_NAME_LEN
13522	SQL_MAX_COLUMN_NAME_LEN	SQL_MAX_ROW_SIZE
13523	SQL_MAX_COLUMNS_IN_GROUP_BY	SQL_MAX_ROW_SIZE_INCLUDES_LONG
13524	SQL_MAX_COLUMNS_IN_INDEX	SQL_MAX_SCHEMA_NAME_LEN
13525	SQL_MAX_COLUMNS_IN_ORDER_BY	SQL_MAX_STATEMENT_LEN
13526	SQL_MAX_COLUMNS_IN_SELECT	SQL_MAX_TABLE_NAME_LEN
13527	SQL_MAX_COLUMNS_IN_TABLE	SQL_MAX_TABLES_IN_SELECT
13528	SQL_MAX_CURSOR_NAME_LEN	SQL_MAX_USER_NAME_LEN

13529 **Scalar Function Information**

13530 The following values of *InfoType* return information about the scalar functions the
13531 implementation supports (see Appendix F).

13532	SQL_CONVERT_FUNCTIONS	SQL_TIMEDATE_ADD_INTERVALS
13533	SQL_NUMERIC_FUNCTIONS	SQL_TIMEDATE_DIFF_INTERVALS
13534	SQL_STRING_FUNCTIONS	SQL_TIMEDATE_FUNCTIONS
13535	SQL_SYSTEM_FUNCTIONS	

13536 **Conversion Information**

13537 The following values of *InfoType* return a list of the SQL data types to which the data source can
13538 convert the specified SQL data type with the CONVERT scalar function:

13539	SQL_CONVERT_BIGINT	SQL_CONVERT_LONGVARBINARY
13540	SQL_CONVERT_BINARY	SQL_CONVERT_LONGVARCHAR
13541	SQL_CONVERT_BIT	SQL_CONVERT_NUMERIC
13542	SQL_CONVERT_CHAR	SQL_CONVERT_REAL
13543	SQL_CONVERT_DATE	SQL_CONVERT_SMALLINT
13544	SQL_CONVERT_DECIMAL	SQL_CONVERT_TIME
13545	SQL_CONVERT_DOUBLE	SQL_CONVERT_TIMESTAMP
13546	SQL_CONVERT_FLOAT	SQL_CONVERT_TINYINT
13547	SQL_CONVERT_INTEGER	SQL_CONVERT_VARBINARY
13548	SQL_CONVERT_INTERVAL_DAY_TIME	SQL_CONVERT_VARCHAR
13549	SQL_CONVERT_INTERVAL_YEAR_MONTH	

13550 **Information Type Descriptions**

13551 Here is an alphabetical list of each valid value of *InfoType* and a description of the information
13552 obtained.

13553 SQL_ACCESSIBLE_PROCEDURES
13554 A character string: ''Y'' if the user can execute all procedures returned by *SQLProcedures*(),
13555 ''N'' if there may be procedures returned that the user cannot execute.

13556 SQL_ACCESSIBLE_TABLES
13557 A character string: ''Y'' if the user is guaranteed SELECT privileges to all tables returned by
13558 *SQLTables*(), ''N'' if there may be tables returned that the user cannot access.

13559 SQL_ACTIVE_ENVIRONMENTS
13560 An SQLSMALLINT value specifying the maximum number of active environments that the
13561 implementation supports. If there is no specified limit or the limit is unknown, this value is
13562 set to zero.

13563 SQL_ALTER_DOMAIN
13564 An SQLINTEGER bitmask enumerating the clauses in the ALTER DOMAIN statement, as
13565 defined in the ISO SQL standard, that the implementation supports.

13566 The following bitmasks are used to determine which clauses are supported:

13567 SQL_AD_ALTER_DOMAIN
13568 SQL_AD_ADD_DOMAIN_CONSTRAINT
13569 SQL_AD_DROP_DOMAIN_CONSTRAINT
13570 SQL_AD_ADD_DOMAIN_DEFAULT
13571 SQL_AD_DROP_DOMAIN_DEFAULT

13572 SQL_ALTER_TABLE
13573 An SQLINTEGER bitmask enumerating the clauses in the ALTER TABLE statement that the
13574 implementation supports. The following bitmasks are available. If the bit is set, it indicates
13575 support for the corresponding feature of the ALTER TABLE statement. The standards
13576 compliance level corresponding to each clause is shown in parentheses.

13577 SQL_AT_ADD_COLUMN_SINGLE
13578 <add column> clause (FIPS Transitional)

13579 SQL_AT_ADD_COLUMN_CONSTRAINT
13580 <add column> clause with column constraints (FIPS Transitional)

13581 SQL_AT_ADD_COLUMN_DEFAULT
13582 <add column> clause with column defaults (FIPS Transitional)

13583 SQL_AT_ADD_COLUMN_COLLATION
13584 <add column> clause with column collation (Full)

13585 SQL_AT_ALTER_COLUMN_SET_DEFAULT
13586 <alter column> <set column default clause> (Intermediate)

13587 SQL_AT_ALTER_COLUMN_DROP_DEFAULT
13588 <alter column> <drop column default clause> (Intermediate)

13589 SQL_AT_DROP_COLUMN_CASCADE
13590 <drop column> CASCADE (FIPS Transitional)

13591 SQL_AT_DROP_COLUMN_RESTRICT
13592 <drop column> RESTRICT (FIPS Transitional)

13593 SQL_AT_ADD_TABLE_CONSTRAINT
13594 <add table constraint> clause (FIPS Transitional)

13595 SQL_AT_DROP_TABLE_CONSTRAINT_CASCADE
13596 <drop table constraint> CASCADE clause (FIPS Transitional)

13597 SQL_AT_DROP_TABLE_CONSTRAINT_RESTRICT
13598 <drop table constraint> RESTRICT clause (FIPS Transitional)

13599 SQL_AT_CONSTRAINT_NAME_DEFINITION
13600 <constraint name definition> for naming column and table constraints. (Intermediate)

13601 The following 4 bits specify the supported <constraint attributes> if specifying column or
13602 table constraints is supported (Full):

13603 SQL_AT_CONSTRAINT_INITIALLY_DEFERRED
13604 SQL_AT_CONSTRAINT_INITIALLY_IMMEDIATE
13605 SQL_AT_CONSTRAINT_DEFERRABLE
13606 SQL_AT_CONSTRAINT_NON_DEFERRABLE

13607 SQL_ANSI_SQL_DATETIME_LITERALS
13608 An SQLINTEGER bitmask enumerating the date/time literals that the implementation
13609 supports. These are the date/time literals listed in the ISO SQL standard, not the date/time
13610 literal escape clauses defined by XDBC in Section 8.3.1 on page 84.

13611 The following bitmasks are used to determine which literals are supported:

13612 SQL_ASDL_DATE
13613 SQL_ASDL_TIME
13614 SQL_ASDL_TIMESTAMP
13615 SQL_ASDL_INTERVAL_YEAR
13616 SQL_ASDL_INTERVAL_MONTH
13617 SQL_ASDL_INTERVAL_DAY
13618 SQL_ASDL_INTERVAL_HOUR
13619 SQL_ASDL_INTERVAL_MINUTE
13620 SQL_ASDL_INTERVAL_SECOND
13621 SQL_ASDL_INTERVAL_YEAR_TO_MONTH
13622 SQL_ASDL_INTERVAL_DAY_TO_HOUR
13623 SQL_ASDL_INTERVAL_DAY_TO_MINUTE
13624 SQL_ASDL_INTERVAL_DAY_TO_SECOND
13625 SQL_ASDL_INTERVAL_HOUR_TO_MINUTE
13626 SQL_ASDL_INTERVAL_HOUR_TO_SECOND
13627 SQL_ASDL_INTERVAL_MINUTE_TO_SECOND

13628 SQL_ANSI_SQL_CONFORMANCE
13629 An SQLINTEGER value indicating the level of SQL grammar the implementation supports:

13630 SQL_ASC_ANSI92_ENTRY_LEVEL
13631 Entry-level grammar of the ISO SQL standard

13632 SQL_ASC_FIPS_TRANSITIONAL
13633 The transitional level of FIPS 127-2

13634 SQL_ASC_ANSI_92_FULL
13635 Full grammar of the ISO SQL standard

13636 SQL_ASC_ANSI_92_INTERMEDIATE
13637 Intermediate-level grammar of the ISO SQL standard

13638 SQL_ASYNC_MODE
13639 An SQLINTEGER value indicating the extent to which the implementation supports
13640 asynchrony:

13641 SQL_AM_CONNECTION
13642 Connection level asynchronous execution is supported. Either all statement handles
13643 associated with a given connection handle are in asynchronous mode, or all are in
13644 synchronous mode. A statement handle on a connection cannot be in asynchronous
13645 mode while another statement handle on the same connection is in synchronous mode,
13646 and vice versa.

13647 SQL_AM_STATEMENT
13648 Statement level asynchronous execution is supported. Some statement handles
13649 associated with a connection handle can be in asynchronous mode, while other
13650 statement handles on the same connection are in synchronous mode.

13651 SQL_AM_NONE
13652 Asynchronous mode is not supported.

13653 SQL_BATCH_ROW_COUNT
13654 An SQLINTEGER bitmask enumerating the behavior of the data source with respect to the
13655 availability of row counts. The following bit masks are used in conjunction with the option: |

13656 SQL_BRC_ROLLED_UP
13657 Row counts for consecutive INSERT, DELETE, or UPDATE statements are rolled up
13658 into one. If this bit is not set, then row counts are available for each individual
13659 statement.

13660 SQL_BRC_PROCEDURES
13661 Row counts, if any, are available when a batch is executed in a stored procedure. If row
13662 counts are available, they may be rolled up or individually available, depending on the
13663 SQL_BRC_ROLLED_UP bit.

13664 SQL_BRC_EXPLICIT
13665 Row counts, if any, are available when a batch is executed directly by calling
13666 *SQLExecute*() or *SQLExecDirect*(). If row counts are available, they may be rolled up or
13667 individually available, depending on the SQL_BRC_ROLLED_UP bit.

13668 SQL_BATCH_SUPPORT
13669 An SQLINTEGER bitmask specifying whether the implementation supports batches. The
13670 following bitmasks are used to determine which level is supported:

13671 SQL_BS_SELECT_EXPLICIT
13672 The implementation supports explicit batches that can have result-set generating
13673 statements.

13674 SQL_BS_ROW_COUNT_EXPLICIT
13675 The implementation supports explicit batches that can have row-count generating
13676 statements.

13677 SQL_BS_SELECT_PROC
13678 The implementation supports explicit procedures that can have result-set generating
13679 statements.

13680 SQL_BS_ROW_COUNT_PROC
13681 The implementation supports explicit procedures that can have row-count generating
13682 statements.

13683 SQL_BOOKMARK_PERSISTENCE
13684 An SQLINTEGER bitmask enumerating the operations through which bookmarks persist.

13685 The following bitmasks are used in conjunction with the flag to determine through which
13686 options bookmarks persist:

13687 SQL_BP_CLOSE
13688 Bookmarks are valid after an application calls *SQLFreeStmt*() with the SQL_CLOSE
13689 option, or *SQLCloseCursor*() to close the cursor associated with a statement.

13690 SQL_BP_DELETE
13691 The bookmark for a row is valid after that row has been deleted.

13692 SQL_BP_DROP
13693 Bookmarks are valid after an application calls *SQLFreeHandle*() with *HandleType* of
13694 SQL_HANDLE_STMT to drop a statement.

13695 SQL_BP_TRANSACTION
13696 Bookmarks are valid after an application commits or rolls back a transaction.

13697 SQL_BP_UPDATE
13698 The bookmark for a row is valid after any column in that row has been updated,
13699 including key columns.

13700 SQL_BP_OTHER_HSTMT
13701 A bookmark associated with one statement can be used with another statement. Unless
13702 SQL_BP_CLOSE or SQL_BP_DROP is specified, the cursor on the first statement must
13703 be open.

13704 SQL_BP_DISCONNECT
13705 Bookmarks are valid after an application disconnects from the data source.

13706 SQL_CATALOG_LOCATION
13707 An SQLSMALLINT value indicating the position of the catalog in a qualified table name.
13708 This is one of the following:

13709 SQL_CL_START The catalog appears at the start of the table name.
13710 SQL_CL_END The catalog appears at the end of the table name.

13711 SQL_CATALOG_NAME
13712 A character string: ‘‘Y’’ if the *CatalogName* argument of the catalog functions can be used to
13713 specify a catalog, or ‘‘N’’ if it cannot. (The SQL_CATALOG_USAGE option provides more
13714 information on the valid contexts in which catalogs can be specified.)

13715 SQL_CATALOG_NAME_SEPARATOR
13716 A character string: the character or characters that the data source defines as the separator
13717 between a catalog name and the qualified name element that follows or precedes it.

13718 SQL_CATALOG_TERM
13719 A character string with the data source vendor’s name for a catalog; for example,
13720 ‘‘database’’ or ‘‘directory.’’

13721 SQL_CATALOG_USAGE
13722 An SQLINTEGER bitmask enumerating the statements in which catalogs can be used.

13723 The following bitmasks are used to determine where catalogs can be used:

13724 SQL_CU_DML_STATEMENTS
13725 Catalogs are supported in all Data Manipulation Language statements: SELECT,
13726 INSERT, UPDATE, DELETE, and, if supported, SELECT FOR UPDATE and positioned
13727 UPDATE and DELETE statements.

13728 SQL_CU_PROCEDURE_INVOCATION
13729 Catalogs are supported in the XDBC escape clause to call a procedure.

13730 SQL_CU_TABLE_DEFINITION
13731 Catalogs are supported in all table definition statements: CREATE TABLE, CREATE

13732 VIEW, ALTER TABLE, DROP TABLE, and DROP VIEW.

13733 SQL_CU_INDEX_DEFINITION
13734 Catalogs are supported in all index definition statements: CREATE INDEX and DROP
13735 INDEX.

13736 SQL_CU_PRIVILEGE_DEFINITION Catalogs are supported in all privilege definition
13737 statements: GRANT and REVOKE.

13738 SQL_COLLATION_SEQ
13739 The name of the collation sequence. This is a character string that indicates the default
13740 ordering of the character set for this data source.

13741 SQL_COLUMN_ALIAS
13742 A character string: ''Y'' if the data source supports column aliases; ''N'' otherwise.

13743 SQL_CONCAT_NULL_BEHAVIOR
13744 An SQLSMALLINT value indicating how the data source handles the concatenation of
13745 NULL valued character data type columns with non-NULL valued character data type
13746 columns:

13747 SQL_CB_NULL Result is NULL valued.

13748 SQL_CB_NON_NULL Result is concatenation of non-NULL valued column or
13749 columns.

13750 SQL_CONVERT_*
13751 A series of SQLINTEGER bitmasks. Any of the following may be specified as *InfoType*:

13752 SQL_CONVERT_BIGINT SQL_CONVERT_LONGVARBINARY
13753 SQL_CONVERT_BINARY SQL_CONVERT_LONGVARCHAR
13754 SQL_CONVERT_BIT SQL_CONVERT_NUMERIC
13755 SQL_CONVERT_CHAR SQL_CONVERT_REAL
13756 SQL_CONVERT_DATE SQL_CONVERT_SMALLINT
13757 SQL_CONVERT_DECIMAL SQL_CONVERT_TIME
13758 SQL_CONVERT_DOUBLE SQL_CONVERT_TIMESTAMP
13759 SQL_CONVERT_FLOAT SQL_CONVERT_TINYINT
13760 SQL_CONVERT_INTEGER SQL_CONVERT_VARBINARY
13761 SQL_CONVERT_INTERVAL_DAY_TIME SQL_CONVERT_VARCHAR
13762 SQL_CONVERT_INTERVAL_YEAR_MONTH

13763 The bitmask indicates the conversions supported by the data source with the CONVERT
13764 scalar function for data of the type named in *InfoType*. If the bitmask equals zero, the data
13765 source does not support any conversions for data of the named type, including conversion
13766 to the same data type.

13767 For example, to find out if a data source supports the conversion of SQL_INTEGER data to
13768 the SQL_BIGINT data type, an application calls *SQLGetInfo*() with an *InfoType* of
13769 SQL_CONVERT_INTEGER. The application combines the returned bitmask with
13770 SQL_CVT_BIGINT with the AND operation. If the resulting value is nonzero, the
13771 conversion is supported.

13772 The following bitmasks are used to determine which conversions are supported:

13773 SQL_CVT_BIGINT SQL_CVT_LONGVARBINARY
13774 SQL_CVT_BINARY SQL_CVT_LONGVARCHAR
13775 SQL_CVT_BIT SQL_CVT_NUMERIC
13776 SQL_CVT_CHAR SQL_CVT_REAL
13777 SQL_CVT_DATE SQL_CVT_SMALLINT

13778 SQL_CVT_DECIMAL SQL_CVT_TIME
13779 SQL_CVT_DOUBLE SQL_CVT_TIMESTAMP
13780 SQL_CVT_FLOAT SQL_CVT_TINYINT
13781 SQL_CVT_INTEGER SQL_CVT_VARBINARY
13782 SQL_CVT_INTERVAL_DAY_TIME SQL_CVT_VARCHAR
13783 SQL_CVT_INTERVAL_YEAR_MONTH

13784 SQL_CONVERT_FUNCTIONS
13785 An SQLINTEGER bitmask enumerating the scalar conversion functions the implementation
13786 supports. These functions are defined in Section F.5 on page 609.

13787 The following bitmasks are used to determine which conversion functions are supported:

13788 SQL_FN_CVT_CAST The **CAST** function is supported.

13789 SQL_FN_CVT_CONVERT The **CONVERT** function is supported.

13790 SQL_CORRELATION_NAME
13791 An SQLSMALLINT value indicating if table correlation names are supported:

13792 SQL_CN_NONE Correlation names are not supported.

13793 SQL_CN_DIFFERENT Correlation names are supported, but must differ from the
13794 names of the tables they represent.

13795 SQL_CN_ANY Correlation names are supported and can be any valid
13796 user-defined name.

13797 SQL_CREATE_ASSERTION
13798 An SQLINTEGER bitmask enumerating the clauses in the CREATE ASSERTION statement,
13799 as defined in the ISO SQL standard, supported by the data source. The following bitmasks
13800 are used to determine which clauses are supported:

13801 SQL_CAS_CREATE_ASSERTION
13802 SQL_CAS_INITIALLY_DEFERRED
13803 SQL_CAS_INITIALLY_IMMEDIATE
13804 SQL_CAS_DEFERRABLE
13805 SQL_CAS_NOT_DEFERRABLE

13806 SQL_CREATE_CHARACTER_SET
13807 An SQLINTEGER bitmask enumerating the clauses in the CREATE CHARACTER SET
13808 statement, as defined in the ISO SQL standard, supported by the data source. The following
13809 bitmasks are used to determine which clauses are supported:

13810 SQL_CCS_CREATE_CHARACTER_SET
13811 SQL_CCS_COLLATE_CLAUSE
13812 SQL_CCS_LIMITED_COLLATION

13813 SQL_CREATE_COLLATION
13814 An SQLINTEGER bitmask enumerating the clauses in the CREATE COLLATION statement,
13815 as defined in the ISO SQL standard, supported by the data source.

13816 The following bitmask is used to determine which clauses are supported:

13817 SQL_CCOL_CREATE_COLLATION

13818 SQL_CREATE_DOMAIN
13819 An SQLINTEGER bitmask enumerating the clauses in the CREATE DOMAIN statement, as
13820 defined in the ISO SQL standard, supported by the data source.

13821 The following bitmasks are used to determine which clauses are supported:

13822 SQL_CDO_CREATE_DOMAIN
13823 SQL_CDO_DEFAULT
13824 SQL_CDO_CONSTRAINT
13825 SQL_CDO_COLLATION

13826 SQL_CREATE_SCHEMA
13827 An SQLINTEGER bitmask enumerating the clauses in the CREATE SCHEMA statement, as
13828 defined in the ISO SQL standard, supported by the data source.

13829 The following bitmasks are used to determine which clauses are supported:

13830 SQL_CS_CREATE_SCHEMA
13831 SQL_CS_AUTHORIZATION
13832 SQL_CS_DEFAULT_CHARACTER_SET

13833 SQL_CREATE_TABLE
13834 An SQLINTEGER bitmask enumerating the clauses in the CREATE TABLE statement, as
13835 defined in the ISO SQL standard, supported by the data source.

13836 The following bitmasks are used to determine which clauses are supported:

13837 SQL_CT_COMMIT_PRESERVE
13838 SQL_CT_COMMIT_DELETE
13839 SQL_CT_GLOBAL_TEMPORARY
13840 SQL_CT_LOCAL_TEMPORARY
13841 SQL_CT_CONSTRAINT_INITIALLY_DEFERRED
13842 SQL_CT_CONSTRAINT_INITIALLY_IMMEDIATE
13843 SQL_CT_CONSTRAINT_DEFERRABLE
13844 SQL_CT_CONSTRAINT_NON_DEFERRABLE

13845 SQL_CREATE_TRANSLATION
13846 An SQLINTEGER bitmask enumerating the clauses in the CREATE TRANSLATION
13847 statement, as defined in the ISO SQL standard, supported by the data source.

13848 The following bitmask is used to determine which clauses are supported:

13849 SQL_CTR_CREATE_TRANSLATION

13850 SQL_CREATE_VIEW
13851 An SQLINTEGER bitmask enumerating the clauses in the CREATE VIEW statement, as
13852 defined in the ISO SQL standard, supported by the data source.

13853 The following bitmasks are used to determine which clauses are supported:

13854 SQL_CV_CREATE_VIEW
13855 SQL_CV_CHECK_OPTION
13856 SQL_CV_CASCADED
13857 SQL_CV_LOCAL

13858 SQL_CURSOR_COMMIT_BEHAVIOR
13859 An SQLSMALLINT value indicating how a COMMIT operation affects cursors and
13860 prepared statements in the data source:

13861 SQL_CB_DELETE
13862 Close cursors and delete prepared statements. To use the cursor again, the application
13863 must reprepare and reexecute the statement.

13864 SQL_CB_CLOSE
13865 Close cursors. For prepared statements, the application can call *SQLExecute*() on the
13866 statement without calling *SQLPrepare*() again.

13867 SQL_CB_PRESERVE
13868 Preserve cursors in the same position as before the COMMIT operation. The
13869 application can continue to fetch data or it can close the cursor and reexecute the
13870 statement without repreparing it.

13871 SQL_CURSOR_ROLLBACK_BEHAVIOR
13872 An SQLSMALLINT value indicating how a ROLLBACK operation affects cursors and
13873 prepared statements in the data source. The values are the same as for
13874 SQL_CURSOR_COMMIT_BEHAVIOR.

13875 SQL_CURSOR_SENSITIVITY
13876 An SQLINTEGER value indicating the support for cursor sensitivity:

13877 SQL_INSENSITIVE
13878 All cursors on the statement handle show the result set without reflecting any changes |
13879 made to it by any other cursor within the same transaction.

13880 SQL_UNSPECIFIED
13881 Support for cursor sensitivity is unspecified. It is unspecified whether cursors on the
13882 statement handle make visible the changes made to a result set by another cursor |
13883 within the same transaction. Cursors on the statement handle may make visible none, |
13884 some, or all such changes.

13885 SQL_SENSITIVE
13886 Cursors are sensitive to changes made by other cursors in the same transaction. |

13887 SQL_DATA_SOURCE_NAME
13888 A character string with the data source name used during connection. If the application
13889 called *SQLConnect*(), this is the value of the *DSN* argument. If the application called |
13890 *SQLDriverConnect*() or *SQLBrowseConnect*(), this is the value of the DSN keyword in the
13891 connection string. If the connection string did not contain the DSN keyword (for example,
13892 when implementation-defined keywords specified the connection), this is an empty string.

13893 SQL_DATA_SOURCE_READ_ONLY
13894 A character string. ''Y'' if the data source is set to READ ONLY mode, ''N'' if it is otherwise.

13895 SQL_DATABASE_NAME
13896 A character string with the name of the current database in use, if the data source defines a
13897 named object called ''database.'' (This information is also available by calling
13898 *SQLGetConnectAttr*() with an *Attribute* of SQL_ATTR_CURRENT_CATALOG.)

13899 SQL_DBMS_NAME
13900 A character string with the product name of the data source.

13901 SQL_DBMS_VER
13902 A character string indicating the product version of the data source. The string must begin |
13903 with the product version, in the format ##.##.####, where the first two digits are the major |
13904 version, the next two digits are the minor version, and the last four digits are the release |
13905 version. This may be followed by a product-specific version identification.

13906 SQL_DEFAULT_TXN_ISOLATION
13907 An SQLINTEGER value that indicates the implementation's default transaction isolation |
13908 level. This is the initial value of the SQL_ATTR_TXN_ISOLATION connection attribute. Its |
13909 value is one of the following:

13910 0 Transactions are not supported.

13911 SQL_TXN_READ_UNCOMMITTED
13912 Isolation is at the Read Uncommitted level: Dirty reads, nonrepeatable reads, and
13913 phantoms are possible.

13914 SQL_TXN_READ_COMMITTED
13915 Isolation is at the Read Committed level: Dirty reads are not possible. Nonrepeatable
13916 reads and phantoms are possible.

13917 SQL_TXN_REPEATABLE_READ
13918 Isolation is at the Repeatable Read level: Dirty reads and nonrepeatable reads are not
13919 possible. Phantoms are possible.

13920 SQL_TXN_SERIALIZABLE
13921 Transactions are serializable. Serializable transactions do not allow dirty reads,
13922 nonrepeatable reads, or phantoms.

13923 The terms used above for both the isolation levels and the isolation failure phenomena are
13924 defined in Section 14.2.2 on page 186.

13925 SQL_DESCRIBE_PARAMETER
13926 A character string: ''Y'' if parameters can be described; ''N'' if not.

13927 SQL_DROP_ASSERTION
13928 An SQLINTEGER bitmask enumerating the clauses in the DROP ASSERTION statement, as |
13929 defined in the ISO SQL standard, supported by the data source.

13930 The following bitmask is used to determine which clauses are supported:

13931 SQL_DA_DROP_ASSERTION

13932 SQL_DROP_CHARACTER_SET
13933 An SQLINTEGER bitmask enumerating the clauses in the DROP CHARACTER SET |
13934 statement, as defined in the ISO SQL standard, supported by the data source.

13935 The following bitmask is used to determine which clauses are supported:

13936 SQL_DCS_DROP_CHARACTER_SET

13937 SQL_DROP_COLLATION
13938 An SQLINTEGER bitmask enumerating the clauses in the DROP COLLATION statement, |
13939 as defined in the ISO SQL standard, supported by the data source.

13940 The following bitmask is used to determine which clauses are supported:

13941 SQL_DC_DROP_COLLATION

13942 SQL_DROP_DOMAIN
13943 An SQLINTEGER bitmask enumerating the clauses in the DROP DOMAIN statement, as |
13944 defined in the ISO SQL standard, supported by the data source.

13945 The following bitmasks are used to determine which clauses are supported:

13946 SQL_DD_DROP_DOMAIN
13947 SQL_DD_CASCADE
13948 SQL_DD_RESTRICT

13949 SQL_DROP_SCHEMA
13950 An SQLINTEGER bitmask enumerating the clauses in the DROP SCHEMA statement, as |
13951 defined in the ISO SQL standard, supported by the data source.

13952 The following bitmasks are used to determine which clauses are supported:

13953 SQL_DS_DROP_SCHEMA
13954 SQL_DS_CASCADE
13955 SQL_DS_RESTRICT

13956 SQL_DROP_TABLE
13957 An SQLINTEGER bitmask enumerating the clauses in the DROP TABLE statement, as
13958 defined in the ISO SQL standard, supported by the data source.

13959 The following bitmasks are used to determine which clauses are supported:

13960 SQL_DT_DROP_TABLE
13961 SQL_DT_CASCADE
13962 SQL_DT_RESTRICT

13963 SQL_DROP_TRANSLATION
13964 An SQLINTEGER bitmask enumerating the clauses in the DROP TRANSLATION
13965 statement, as defined in the ISO SQL standard, supported by the data source.

13966 The following bitmask is used to determine which clauses are supported:

13967 SQL_DTR_DROP_TRANSLATION

13968 SQL_DROP_VIEW
13969 An SQLINTEGER bitmask enumerating the clauses in the DROP VIEW statement, as
13970 defined in the ISO SQL standard, supported by the data source.

13971 The following bitmasks are used to determine which clauses are supported:

13972 SQL_DV_DROP_VIEW
13973 SQL_DV_CASCADE
13974 SQL_DV_RESTRICT

13975 SQL_DYNAMIC_CURSOR_ATTRIBUTES1

13976 SQL_DYNAMIC_CURSOR_ATTRIBUTES2
13977 A pair of 32-bit bitmasks that indicate supported operations for dynamic cursors and
13978 describe other attributes of dynamic cursors. See **Detecting Cursor Capabilities with**
13979 **SQLGetInfo()** on page 402.

13980 SQL_EXPRESSIONS_IN_ORDERBY
13981 A character string: ''Y'' if the data source supports expressions in the ORDER BY list; ''N'' if
13982 it does not.

13983 DE SQL_FETCH_DIRECTION (type: INTEGER)
13984 This indicates the type of cursor movement the implementation supports. The value is a
13985 32-bit bitmask with the low-order bits identified as follows:

13986 SQL_FD_ABSOLUTE
13987 SQL_FD_FIRST
13988 SQL_FD_LAST
13989 SQL_FD_NEXT
13990 SQL_FD_PRIOR
13991 SQL_FD_RELATIVE

13992 SQL_FILE_USAGE
13993 An SQLSMALLINT value indicating how a single-tier implementation directly treats files in
13994 a data source:

13995 SQL_FILE_NOT_SUPPORTED
13996 The driver is not a single-tier driver.

13997 SQL_FILE_TABLE
13998 A single-tier driver treats files in a data source as tables.

13999 SQL_FILE_CATALOG
14000 A single-tier driver treats files in a data source as a catalog.

14001 An application might use this to determine how users will select data. For example, the
14002 procedure by which the user selects data may vary depending on whether the user is
14003 opening a file or a table.

14004 SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES1

14005 SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES2
14006 A pair of 32-bit bitmasks that indicate supported operations for forward-only cursors and
14007 describe other attributes of forward-only cursors. See **Detecting Cursor Capabilities with**
14008 **SQLGetInfo()** on page 402.

14009 SQL_GETDATA_EXTENSIONS
14010 An SQLINTEGER bitmask enumerating extensions to *SQLGetData*().

14011 The following bitmasks are used in conjunction with the flag to determine what common
14012 extensions the implementation supports for *SQLGetData*():

14013 SQL_GD_ANY_COLUMN
14014 *SQLGetData*() can be called for any unbound column, including those before the last
14015 bound column. Note that the columns must be called in order of ascending column
14016 number unless SQL_GD_ANY_ORDER is also returned.

14017 SQL_GD_ANY_ORDER
14018 *SQLGetData*() can be called for unbound columns in any order. Note that *SQLGetData*()
14019 can only be called for columns after the last bound column unless
14020 SQL_GD_ANY_COLUMN is also returned.

14021 SQL_GD_BLOCK
14022 *SQLGetData*() can be called for an unbound column in any row in a block (where the |
14023 row-set size is greater than 1) of data after positioning to that row with *SQLSetPos*().

14024 SQL_GD_BOUND
14025 *SQLGetData*() can be called for bound columns as well as unbound columns. An
14026 implementation cannot return this value unless it also returns
14027 SQL_GD_ANY_COLUMN.

14028 *SQLGetData*() is only required to return data from unbound columns that occur after
14029 the last bound column, are called in order of increasing column number, and are not in
14030 a row in a block of rows.

14031 If the implementation supports bookmarks, it must support calling *SQLGetData*() on
14032 column 0, regardless of which of the above values it returns.

14033 SQL_GROUP_BY
14034 An SQLSMALLINT value specifying the relationship between the columns in the GROUP
14035 BY clause and the non-aggregated columns in the select list:

14036 SQL_GB_NOT_SUPPORTED
14037 GROUP BY clauses are not supported.

14038 SQL_GB_GROUP_BY_EQUALS_SELECT
14039 The GROUP BY clause must contain all non-aggregated columns in the select list. It
14040 cannot contain any other columns. For example, SELECT DEPT, MAX(SALARY)
14041 FROM EMPLOYEE GROUP BY DEPT.

14042 SQL_GB_GROUP_BY_CONTAINS_SELECT
14043 The GROUP BY clause must contain all non-aggregated columns in the select list. It can
14044 contain columns that are not in the select list. For example, SELECT DEPT,
14045 MAX(SALARY) FROM EMPLOYEE GROUP BY DEPT, AGE.

14046 SQL_GB_NO_RELATION The columns in the GROUP BY clause and the select list are not
14047 related. The meaning of non-grouped, non-aggregated columns in the select list is data
14048 source-dependent. For example, SELECT DEPT, SALARY FROM EMPLOYEE GROUP
14049 BY DEPT, AGE.

14050 SQL_IDENTIFIER_CASE
14051 An SQLSMALLINT value as follows:

14052 SQL_IC_UPPER
14053 Identifiers in SQL are not case-sensitive and are stored in upper case in the system
14054 catalog.

14055 SQL_IC_LOWER
14056 Identifiers in SQL are not case-sensitive and are stored in lower case in the system
14057 catalog.

14058 SQL_IC_SENSITIVE
14059 Identifiers in SQL are case-sensitive and are stored in mixed case in the system catalog.

14060 SQL_IC_MIXED
14061 Identifiers in SQL are not case-sensitive and are stored in mixed case in the system
14062 catalog.

14063 SQL_IDENTIFIER_QUOTE_CHAR
14064 The character string used as the starting and ending delimiter of a quoted (delimited)
14065 identifiers in SQL statements. (Identifiers passed as arguments to XDBC functions do not
14066 need to be quoted.) If the data source does not support quoted identifiers, a blank is
14067 returned.

14068 SQL_INDEX_KEYWORDS
14069 A 32-bit bitmask that enumerates keywords in the CREATE INDEX statement that the
14070 implementation supports. |

14071 SQL_IK_NONE None of the keywords are supported.
14072 SQL_IK_ASC ASC keyword is supported.
14073 SQL_IK_DESC DESC keyword is supported.
14074 SQL_IK_ALL Both keywords are supported.

14075 SQL_INFO_SCHEMA_VIEWS
14076 An SQLINTEGER bitmask enumerating the views in the INFORMATION_SCHEMA that
14077 the implementation supports. The views in, and the contents of, INFORMATION_SCHEMA |
14078 are as defined in the ISO SQL standard.

14079 The following bitmasks are used to determine which views are supported:

14080 SQL_ISV_ASSERTIONS
14081 Identifies the catalog's assertions that are owned by a given user.

14082 SQL_ISV_CHARACTER_SETS
14083 Identifies the catalog's character sets that are accessible to a given user.

14084 SQL_ISV_CHECK_CONSTRAINTS
14085 Identifies the CHECK constraints that are owned by a given user.

14086 SQL_ISV_COLLATIONS
14087 Identifies the character collations for the catalog that are accessible to a given user.

14088 SQL_ISV_COLUMN_DOMAIN_USAGE
14089 Identifies columns for the catalog that are dependent on domains defined in the catalog
14090 and are owned by a given user.

14091 SQL_ISV_COLUMN_PRIVILEGES
14092 Identifies the privileges on columns of persistent tables that are available to or granted
14093 by a given user.

14094 SQL_ISV_COLUMNS
14095 Identifies the columns of persistent tables that are accessible to a given user.

14096 SQL_ISV_CONSTRAINT_COLUMN_USAGE
14097 Similar to CONSTRAINT_TABLE_USAGE view, columns are identified for the various
14098 constraints that are owned by a given user.

14099 SQL_ISV_CONSTRAINT_TABLE_USAGE
14100 Identifies the tables that are used by constraints (referential, unique, and assertions),
14101 and are owned by a given user.

14102 SQL_ISV_DOMAIN_CONSTRAINTS
14103 Identifies the domain constraints (of the domains in the catalog) that are accessible to a
14104 given user.

14105 SQL_ISV_DOMAINS
14106 Identifies the domains defined in a catalog that are accessible to the user.

14107 SQL_ISV_KEY_COLUMN_USAGE
14108 Identifies columns defined in the catalog that are constrained as keys by a given user.

14109 SQL_ISV_REFERENTIAL_CONSTRAINTS
14110 Identifies the referential constraints that are owned by a given user.

14111 SQL_ISV_SCHEMATA
14112 Identifies the schemas that are owned by a given user.

14113 SQL_ISV_SQL_LANGUAGES
14114 Identifies the SQL conformance levels, options and dialects supported by the SQL
14115 implementation.

14116 SQL_ISV_TABLE_CONSTRAINTS
14117 Identifies the the table constraints that are owned by a given user.

14118 SQL_ISV_TABLE_PRIVILEGES
14119 Identifies the privileges on persistent tables that are available to or granted by a given
14120 user.

14121 SQL_ISV_TABLES
14122 Identifies the persistent tables defined in a catalog that are accessible to a given user.

14123 SQL_ISV_TRANSLATIONS
14124 Identifies character translations for the catalog that are accessible to a given user.

14125 SQL_ISV_USAGE_PRIVILEGES
14126 Identifies the USAGE privileges on catalog objects that are available to or owned by a
14127 given user.

14128 SQL_ISV_VIEW_COLUMN_USAGE
14129 Identifies the columns on which the catalog's views that are owned by a given user are
14130 dependent.

14131 SQL_ISV_VIEW_TABLE_USAGE
14132 Identifies the tables on which the catalog's views that are owned by a given user are
14133 dependent.

14134 SQL_INTEGRITY
14135 A character string: ''Y'' if the data source supports the Integrity Enhancement Facility; ''N''

14136 if it does not.

14137 SQL_KEYSET_CURSOR_ATTRIBUTES1

14138 SQL_KEYSET_CURSOR_ATTRIBUTES2
14139 A pair of 32-bit bitmasks that indicate supported operations for keyset-driven cursors and
14140 describe other attributes of keyset-driven cursors. See **Detecting Cursor Capabilities with**
14141 **SQLGetInfo()** on page 402.

14142 SQL_KEYWORDS
14143 A character string containing a comma-separated list of all data source-specific keywords.
14144 This list does not contain keywords specific to XDBC or keywords used by both the data
14145 source and XDBC.

14146 The **#define** value SQL_XDBC_KEYWORDS contains a comma-separated list of XDBC
14147 keywords.

14148 SQL_LIKE_ESCAPE_CLAUSE
14149 A character string: ''Y'' if the data source supports an escape character for the percent
14150 character (%) and underscore character (_) in a LIKE predicate and the implementation
14151 supports the XDBC syntax for defining a LIKE predicate escape character; ''N'' otherwise.

14152 SQL_MAX_ASYNC_CONCURRENT_STATEMENTS
14153 An SQLINTEGER value specifying the maximum number of active concurrent statements
14154 in asynchronous mode that the implementation can support on a given connection. If there
14155 is no specific limit or the limit is unknown, this value is zero.

14156 SQL_MAX_BINARY_LITERAL_LEN
14157 An SQLINTEGER value specifying the maximum length (number of hexadecimal
14158 characters, excluding the literal prefix and suffix returned by *SQLGetTypeInfo*()) of a binary
14159 literal in an SQL statement. For example, the binary literal 0xFFAA has a length of 4. If there
14160 is no maximum length or the length is unknown, this value is set to zero.

14161 SQL_MAX_CATALOG_NAME_LEN
14162 An SQLSMALLINT value specifying the maximum length of a catalog name in the data
14163 source. If there is no maximum length or the length is unknown, this value is set to zero.

14164 SQL_MAX_CHAR_LITERAL_LEN
14165 An SQLINTEGER value specifying the maximum length (number of characters, excluding
14166 the literal prefix and suffix returned by *SQLGetTypeInfo*()) of a character literal in an SQL
14167 statement. If there is no maximum length or the length is unknown, this value is set to zero.

14168 SQL_MAX_COLUMN_NAME_LEN
14169 An SQLSMALLINT value specifying the maximum length of a column name in the data
14170 source. If there is no maximum length or the length is unknown, this value is set to zero.

14171 SQL_MAX_COLUMNS_IN_GROUP_BY
14172 An SQLSMALLINT value specifying the maximum number of columns allowed in a
14173 GROUP BY clause. If there is no specified limit or the limit is unknown, this value is set to
14174 zero.

14175 SQL_MAX_COLUMNS_IN_INDEX
14176 An SQLSMALLINT value specifying the maximum number of columns allowed in an
14177 index. If there is no specified limit or the limit is unknown, this value is set to zero.

14178 SQL_MAX_COLUMNS_IN_ORDER_BY
14179 An SQLSMALLINT value specifying the maximum number of columns allowed in an
14180 ORDER BY clause. If there is no specified limit or the limit is unknown, this value is set to
14181 zero.

14182 SQL_MAX_COLUMNS_IN_SELECT
14183 An SQLSMALLINT value specifying the maximum number of columns allowed in a select
14184 list. If there is no specified limit or the limit is unknown, this value is set to zero.

14185 SQL_MAX_COLUMNS_IN_TABLE
14186 An SQLSMALLINT value specifying the maximum number of columns allowed in a table.
14187 If there is no specified limit or the limit is unknown, this value is set to zero.

14188 SQL_MAX_CONCURRENT_ACTIVITIES
14189 An SQLSMALLINT value specifying the maximum number of active statements that the
14190 implementation can support for a connection. A statement is defined as active if it has
14191 results pending, with the term ''results'' meaning rows from a SELECT operation or rows
14192 affected by an INSERT, UPDATE, or DELETE operation (such as a row count), or if it is in a
14193 NEED_DATA state. This value can reflect a limitation imposed by either the data source or
14194 the software that implements the connection to it. If there is no specified limit or the limit is
14195 unknown, this value is set to zero.

14196 SQL_MAX_CURSOR_NAME_LEN
14197 An SQLSMALLINT value specifying the maximum length of a cursor name in the data
14198 source. If there is no maximum length or the length is unknown, this value is set to zero.

14199 SQL_MAX_DRIVER_CONNECTIONS
14200 An SQLSMALLINT value specifying the maximum number of active connections that the
14201 implementation can support for an environment. This value can reflect a limitation imposed
14202 by either the data source or the software that implements the connection to it. If there is no
14203 specified limit or the limit is unknown, this value is set to zero.

14204 SQL_MAX_IDENTIFIER_LEN
14205 An SQLSMALLINT that indicates the maximum size in characters that the data source
14206 supports for user-defined names.

14207 SQL_MAX_INDEX_SIZE
14208 An SQLINTEGER value specifying the maximum number of octets allowed in the |
14209 combined fields of an index. If there is no specified limit or the limit is unknown, this value
14210 is set to zero.

14211 SQL_MAX_PROCEDURE_NAME_LEN
14212 An SQLSMALLINT value specifying the maximum length of a procedure name in the data
14213 source. If there is no maximum length or the length is unknown, this value is set to zero.

14214 SQL_MAX_ROW_SIZE
14215 An SQLINTEGER value specifying the maximum length of a single row in a table. If there
14216 is no specified limit or the limit is unknown, this value is set to zero.

14217 SQL_MAX_ROW_SIZE_INCLUDES_LONG
14218 A character string: ''Y'' if the maximum row size returned for the SQL_MAX_ROW_SIZE |
14219 option includes the length of all SQL_LONGVARCHAR and SQL_LONGVARBINARY
14220 columns in the row; ''N'' otherwise.

14221 SQL_MAX_SCHEMA_NAME_LEN
14222 An SQLSMALLINT value specifying the maximum length of a schema name in the data
14223 source. If there is no maximum length or the length is unknown, this value is set to zero.

14224 SQL_MAX_STATEMENT_LEN
14225 An SQLINTEGER value specifying the maximum length (number of characters, including
14226 white space) of an SQL statement. If there is no maximum length or the length is unknown,
14227 this value is set to zero.

14228 SQL_MAX_TABLE_NAME_LEN
14229 An SQLSMALLINT value specifying the maximum length of a table name in the data
14230 source. If there is no maximum length or the length is unknown, this value is set to zero.

14231 SQL_MAX_TABLES_IN_SELECT
14232 An SQLSMALLINT value specifying the maximum number of tables allowed in the FROM
14233 clause of a SELECT statement. If there is no specified limit or the limit is unknown, this
14234 value is set to zero.

14235 SQL_MAX_USER_NAME_LEN
14236 An SQLSMALLINT value specifying the maximum length of a user name in the data
14237 source. If there is no maximum length or the length is unknown, this value is set to zero.

14238 SQL_MULT_RESULT_SETS
14239 A character string indicating in general the implementation's support for multiple result |
14240 sets. It is "Y" if the data source supports multiple result sets, "N" if it does not. |

14241 It is related to other *SQLGetInfo*() options in that it is "Y" if the SQL_BS_SELECT_EXPLICIT |
14242 or SQL_BS_SELECT_PROC bits are returned for SQL_BATCH_SUPPORT or if |
14243 SQL_PAS_BATCH is returned for SQL_PARAM_ARRAY_SELECTS.

14244 SQL_MULTIPLE_ACTIVE_TXN
14245 A character string: ''Y'' if multiple active transactions on a single connection are allowed,
14246 ''N'' if only one active transaction at a time is supported on a connection.

14247 SQL_NEED_LONG_DATA_LEN
14248 A character string: ''Y'' if the data source needs the length of a long data value (the data
14249 type is SQL_LONGVARCHAR, SQL_LONGVARBINARY, or a long, data source-specific
14250 data type) before that value is sent to the data source, ''N'' if it does not. |

14251 SQL_NON_NULLABLE_COLUMNS
14252 An SQLSMALLINT specifying whether the data source supports NOT NULL in columns:

14253 SQL_NNC_NULL
14254 All columns must be nullable.

14255 SQL_NNC_NON_NULL
14256 Columns might not be nullable (the data source supports the NOT NULL column
14257 constraint in CREATE TABLE statements).

14258 SQL_NULL_COLLATION
14259 An SQLSMALLINT value specifying where NULLs are sorted in a result set:

14260 SQL_NC_END
14261 NULLs are sorted at the end of the result set, regardless of the ASC or DESC keywords.

14262 SQL_NC_HIGH
14263 NULLs are sorted at the high end of the result set, depending on the ASC or DESC
14264 keywords.

14265 SQL_NC_LOW
14266 NULLs are sorted at the low end of the result set, depending on the ASC or DESC
14267 keywords.

14268 SQL_NC_START
14269 NULLs are sorted at the start of the result set, regardless of the ASC or DESC
14270 keywords.

14271 SQL_NUMERIC_FUNCTIONS
14272 An SQLINTEGER bitmask enumerating the scalar numeric functions the implementation |
14273 supports. Numeric functions are defined in Section F.2 on page 603. |

14274 The following bitmasks are used to determine which numeric functions are supported:

SQL_FN_NUM_ABS	SQL_FN_NUM_LOG10
SQL_FN_NUM_ACOS	SQL_FN_NUM_MOD
SQL_FN_NUM_ASIN	SQL_FN_NUM_PI
SQL_FN_NUM_ATAN	SQL_FN_NUM_POWER
SQL_FN_NUM_ATAN2	SQL_FN_NUM_RADIANS
SQL_FN_NUM_CEILING	SQL_FN_NUM_RAND
SQL_FN_NUM_COS	SQL_FN_NUM_ROUND
SQL_FN_NUM_COT	SQL_FN_NUM_SIGN
SQL_FN_NUM_DEGREES	SQL_FN_NUM_SIN
SQL_FN_NUM_EXP	SQL_FN_NUM_SQRT
SQL_FN_NUM_FLOOR	SQL_FN_NUM_TAN
SQL_FN_NUM_LOG	SQL_FN_NUM_TRUNCATE

14287 SQL_OJ_CAPABILITIES
14288 An SQLINTEGER bitmask enumerating the types of outer joins the implementation
14289 supports. The following bitmasks are used to determine which types are supported:

14290 SQL_OJ_LEFT
14291 Left outer joins are supported.

14292 SQL_OJ_RIGHT
14293 Right outer joins are supported.

14294 SQL_OJ_FULL
14295 Full outer joins are supported.

14296 SQL_OJ_NESTED
14297 Nested outer joins are supported.

14298 SQL_OJ_NOT_ORDERED
14299 The column names in the ON clause of the outer join do not have to be in the same
14300 order as their respective table names in the OUTER JOIN clause.

14301 SQL_OJ_INNER
14302 The inner table (the right table in a left outer join or the left table in a right outer join)
14303 can also be used in an inner join. This does not apply to full outer joins, which do not
14304 have an inner table.

14305 SQL_OJ_ALL_COMPARISON_OPS
14306 The comparison operator in the ON clause can be any of the XDBC comparison
14307 operators. If this bit is not set, only the equals (=) comparison operator can be used in
14308 outer joins.

14309 SQL_ORDER_BY_COLUMNS_IN_SELECT
14310 A character string: ‘‘Y’’ if the columns in the ORDER BY clause must be in the select list;
14311 ‘‘N’’ otherwise.

14312 SQL_OUTER_JOINS
14313 A character string; ‘‘Y’’ if the data source supports outer joins and the implementation
14314 supports the XDBC outer join escape clause. ‘‘N’’ otherwise.

14315 SQL_PARAM_ARRAY_ROW_COUNTS
14316 An SQLINTEGER enumerating the implementation’s properties regarding the availability |
14317 of row counts in a parameterized execution. (See Section 11.3 on page 156.) Has the |
14318 following values:

14319 SQL_PARC_BATCH
14320 Individual row counts are available for each set of parameters. This is conceptually
14321 equivalent to the implementation generating a batch of SQL statements, one for each
14322 parameter set in the array. Extended error information can be retrieved by using the

14323 SQL_PARAM_STATUS_PTR descriptor field.

14324 SQL_PARC_NO_BATCH
14325 There is only one row count available, which is the cumulative row count resulting
14326 from the execution of the statement for the entire array of parameters. This is
14327 conceptually equivalent treating the statement along with the entire parameter array as
14328 one atomic unit. Errors are handled the same as if one statement were executed.

14329 SQL_PARAM_ARRAY_SELECTS
14330 An SQLINTEGER enumerating the implementation's properties regarding the availability |
14331 of result sets in a parameterized execution. (See Section 11.3 on page 156.) Has the |
14332 following values:

14333 SQL_PAS_BATCH
14334 There is one result set available per set of parameters. This is conceptually equivalent
14335 to the implementation generating a batch of SQL statements, one for each parameter set
14336 in the array.

14337 SQL_PAS_NO_BATCH
14338 There is only one result set available, which represents the cumulative result set
14339 resulting from the execution of the statement for the entire array of parameters. This is
14340 conceptually equivalent to treating the statement along with the entire parameter array
14341 as one atomic unit.

14342 SQL_PAS_NO_SELECT
14343 The implementation cannot execute a result-set generating statement with an array of
14344 parameters.

14345 SQL_PROCEDURE_TERM
14346 A character string with the data source vendor's name for a procedure; for example,
14347 ''database procedure'', ''stored procedure'', ''procedure'', ''package'', or ''stored query''.

14348 SQL_PROCEDURES
14349 A character string: ''Y'' if the data source supports procedures and the XDBC procedure
14350 invocation syntax; ''N'' otherwise.

14351 SQL_QUOTED_IDENTIFIER_CASE
14352 An SQLINTEGER value as follows:

14353 SQL_IC_UPPER
14354 Quoted identifiers in SQL are not case-sensitive and are stored in upper case in the
14355 system catalog.

14356 SQL_IC_LOWER
14357 Quoted identifiers in SQL are not case-sensitive and are stored in lower case in the
14358 system catalog.

14359 SQL_IC_SENSITIVE
14360 Quoted identifiers in SQL are case-sensitive and are stored in mixed case in the system |
14361 catalog. (Note that in a data source that complies with the ISO SQL standard, quoted |
14362 identifiers are always case-sensitive.)

14363 SQL_IC_MIXED
14364 Quoted identifiers in SQL are not case-sensitive and are stored in mixed case in the
14365 system catalog.

14366 SQL_REVOKE
14367 The following bitmasks are used to determine which clauses are supported:

14368 SQL_R_CASCADE
14369 SQL_R_RESTRICT

14370 SQL_ROW_UPDATES
14371 A character string: ''Y'' if a keyset-driven or mixed cursor maintains row versions or values
14372 for all fetched rows and therefore can detect any updates made to a row by any user since
14373 the row was last fetched. (This only applies to updates, not to deletions or insertions.) The
14374 implementation can return the SQL_ROW_UPDATED flag to the row status array when
14375 *SQLFetchScroll*() is called. Otherwise, ''N''.

14376 SQL_SCHEMA_TERM
14377 A character string with the data source vendor's name for an schema; for example, ''owner'',
14378 ''Authorization ID'', or ''Schema''.

14379 SQL_SCHEMA_USAGE
14380 An SQLINTEGER bitmask enumerating the statements in which schemas can be used:

14381 SQL_SU_DML_STATEMENTS
14382 Schemas are supported in all Data Manipulation Language statements: SELECT,
14383 INSERT, UPDATE, DELETE, and, if supported, SELECT FOR UPDATE and positioned
14384 UPDATE and DELETE statements.

14385 SQL_SU_PROCEDURE_INVOCATION
14386 Schemas are supported in the XDBC procedure invocation statement.

14387 SQL_SU_TABLE_DEFINITION
14388 Schemas are supported in all table definition statements: CREATE TABLE, CREATE
14389 VIEW, ALTER TABLE, DROP TABLE, and DROP VIEW.

14390 SQL_SU_INDEX_DEFINITION
14391 Schemas are supported in all index definition statements: CREATE INDEX and DROP
14392 INDEX.

14393 SQL_SU_PRIVILEGE_DEFINITION
14394 Schemas are supported in all privilege definition statements: GRANT and REVOKE.

14395 DE SQL_SCROLL_CONCURRENCY (type: INTEGER)
14396 This indicates the concurrency control capabilities that the implementation supports for
14397 scrollable cursors. The value is a 32-bit bitmask with the low-order bits identified as
14398 follows:

14399 SQL_SCCO_READ_ONLY
14400 The cursor can be read, but no updates are allowed.

14401 SQL_SCCO_LOCK
14402 The cursor can use the lowest level of locking that ensures that the row can be updated.

14403 SQL_SCCO_OPT_ROWVER
14404 The cursor can use optimistic concurrency with row identifiers or timestamps.

14405 SQL_SCCO_OPT_VALUES
14406 The cursor can use optimistic concurrency comparing values.

14407 SQL_SCROLL_OPTIONS
14408 An SQLINTEGER bitmask enumerating the scroll options supported for scrollable cursors.

14409 The following bitmasks are used to determine which options are supported:

14410 SQL_SO_FORWARD_ONLY
14411 The cursor only scrolls forward.

14412 SQL_SO_STATIC
14413 The data in the result set is static.

14414 SQL_SO_KEYSET_DRIVEN
14415 The implementation saves and uses the keys for every row in the result set.

14416 SQL_SO_DYNAMIC
14417 The implementation keeps the keys for every row in the row-set (the keyset size is the
14418 same as the row-set size).

14419 SQL_SO_MIXED
14420 The implementation keeps the keys for every row in the keyset, and the keyset size is
14421 greater than the row-set size. The cursor is keyset-driven inside the keyset and dynamic
14422 outside the keyset.

14423 For information about scrollable cursors, see Section 11.2 on page 147.

14424 SQL_SEARCH_PATTERN_ESCAPE
14425 A character string specifying what the implementation supports as an escape character that
14426 permits the use of the pattern match metacharacters underscore (_) and percent (%) as valid
14427 characters in search patterns. This escape character applies only for those catalog function
14428 arguments that support search strings. If this string is empty, the implementation does not
14429 support a search-pattern escape character.

14430 This option is limited to catalog functions. Search patterns are defined in **Pattern Value (PV)
14431 Arguments** on page 71.

14432 SQL_SERVER_NAME
14433 A character string with the actual data source-specific server name; useful when a data
14434 source name is used during *SQLConnect*(), *SQLDriverConnect*(), and *SQLBrowseConnect*().

14435 SQL_SPECIAL_CHARACTERS
14436 A character string containing all special characters (that is, all characters except a through z,
14437 A through Z, 0 through 9, and underscore) that can be used in an identifier name, such as a
14438 table, column, or index name, on the data source. For example, ''#$^''. This string contains
14439 characters taken from a single, implementation-defined character set.

14440 Portable applications should use the delimited identifier syntax to code identifiers that
14441 contain one or more of these special characters, and should not create an identifier whose
14442 name begins or ends with a special character.

14443 SQL_SQL92_DATETIME_FUNCTIONS
14444 An SQLINTEGER bitmask enumerating the date/time scalar functions that the
14445 implementation supports, as defined in the ISO SQL standard.

14446 The following bitmasks are used to determine which date/time functions are supported:

14447 SQL_SDF_CURRENT_DATE
14448 SQL_SDF_CURRENT_TIME
14449 SQL_SDF_CURRENT_TIMESTAMP

14450 SQL_SQL92_FOREIGN_KEY_DELETE_RULE
14451 An SQLINTEGER bitmask enumerating the rules supported for a foreign key in a DELETE
14452 statement, as defined in the ISO SQL standard.

14453 The following bitmasks are used to determine which clauses are supported by the data
14454 source:

14455 SQL_SFKDR_DELETE_CASCADE
14456 SQL_SFKDR_DELETE_NO_ACTION
14457 SQL_SFKDR_DELETE_SET_DEFAULT
14458 SQL_SFKDR_DELETE_SET_NULL

14459 SQL_SQL92_FOREIGN_KEY_UPDATE_RULE
14460 An SQLINTEGER bitmask enumerating the rules supported for a foreign key in an UPDATE |
14461 statement, as defined in the ISO SQL standard.

14462 The following bitmasks are used to determine which clauses are supported by the data
14463 source:

14464 SQL_SFKUR_UPDATE_CASCADE
14465 SQL_SFKUR_UPDATE_NO_ACTION
14466 SQL_SFKUR_UPDATE_SET_DEFAULT
14467 SQL_SFKUR_UPDATE_SET_NULL

14468 SQL_SQL92_GRANT
14469 An SQLINTEGER bitmask enumerating the clauses supported in the GRANT statement, as |
14470 defined in the ISO SQL standard.

14471 The following bitmasks are used to determine which clauses are supported by the data
14472 source:

14473 SQL_SG_USAGE_ON_DOMAIN
14474 SQL_SG_USAGE_ON_CHARACTER_SET
14475 SQL_SG_USAGE_ON_COLLATION
14476 SQL_SG_USAGE_ON_TRANSLATION
14477 SQL_SG_WITH_GRANT_OPTION

14478 SQL_SQL92_NUMERIC_VALUE_FUNCTIONS
14479 An SQLINTEGER bitmask enumerating the numeric value scalar functions that the |
14480 implementation supports, as defined in the ISO SQL standard. |

14481 The following bitmasks are used to determine which numeric value scalar functions are |
14482 supported:

14483 SQL_NVF_BIT_LENGTH
14484 SQL_NVF_CHAR_LENGTH
14485 SQL_NVF_CHARACTER_LENGTH
14486 SQL_NVF_EXTRACT
14487 SQL_NVF_OCTET_LENGTH
14488 SQL_NVF_POSITION

14489 SQL_SQL92_PREDICATES
14490 An SQLINTEGER bitmask enumerating the predicates supported in a SELECT statement, as |
14491 defined in the ISO SQL standard.

14492 The following bitmasks are used to determine which options are supported by the data
14493 source:

14494 SQL_SP_EXISTS
14495 SQL_SP_ISNOTNULL
14496 SQL_SP_ISNULL
14497 SQL_SP_MATCH_FULL
14498 SQL_SP_MATCH_PARTIAL
14499 SQL_SP_MATCH_UNIQUE_FULL
14500 SQL_SP_MATCH_UNIQUE_PARTIAL
14501 SQL_SP_OVERLAPS
14502 SQL_SP_UNIQUE

14503 SQL_SQL92_RELATIONAL_JOIN_OPERATORS
14504 An SQLINTEGER bitmask enumerating the relational join operators supported in a SELECT |
14505 statement, as defined in the ISO SQL standard.

14506 The following bitmasks are used to determine which options are supported by the data
14507 source:

14508 SQL_SRJO_CORRESPONDING_CLAUSE
14509 SQL_SRJO_CROSS_JOIN
14510 SQL_SRJO_EXCEPT_JOIN
14511 SQL_SRJO_FULL_OUTER_JOIN
14512 SQL_SRJO_INNER_JOIN
14513 SQL_SRJO_INTERSECT_JOIN
14514 SQL_SRJO_LEFT_OUTER_JOIN
14515 SQL_SRJO_NATURAL_JOIN
14516 SQL_SRJO_RIGHT_OUTER_JOIN
14517 SQL_SRJO_UNION_JOIN

14518 SQL_SQL92_REVOKE
14519 An SQLINTEGER bitmask enumerating the clauses supported in the REVOKE statement, as |
14520 defined in the ISO SQL standard, supported by the data source.

14521 The following bitmasks are used to determine which clauses are supported by the data
14522 source:

14523 SQL_SR_USAGE_ON_DOMAIN
14524 SQL_SR_USAGE_ON_CHARACTER_SET
14525 SQL_SR_USAGE_ON_COLLATION
14526 SQL_SR_USAGE_ON_TRANSLATION
14527 SQL_SR_GRANT_OPTION_FOR

14528 SQL_SQL92_ROW_VALUE_CONSTRUCTOR
14529 An SQLINTEGER bitmask enumerating the row value constructor expressions supported in |
14530 a SELECT statement, as defined in the ISO SQL standard.

14531 The following bitmasks are used to determine which options are supported by the data
14532 source:

14533 SQL_RVC_VALUE_EXPRESSION
14534 SQL_RVC_NULL
14535 SQL_RVC_DEFAULT
14536 SQL_RVC_ROW_SUBQUERY

14537 SQL_SQL92_STRING_FUNCTIONS
14538 An SQLINTEGER bitmask enumerating the string scalar functions that the implementation |
14539 supports, as defined in the ISO SQL standard. |

14540 The following bitmasks are used to determine which string scalar functions are supported:

14541 SQL_SSF_CONVERT
14542 SQL_SSF_LOWER
14543 SQL_SSF_UPPER
14544 SQL_SSF_SUBSTRING
14545 SQL_SSF_TRANSLATE
14546 SQL_SSF_TRIM_BOTH
14547 SQL_SSF_TRIM_LEADING
14548 SQL_SSF_TRIM_TRAILING

14549 SQL_SQL92_VALUE_EXPRESSIONS
14550 An SQLINTEGER bitmask enumerating the value expressions supported in a SELECT |
14551 statement, as defined in the ISO SQL standard.

14552 The following bitmasks are used to determine which options are supported by the data
14553 source:

14554 SQL_SVE_CASE
14555 SQL_SVE_CAST
14556 SQL_SVE_COALESCE
14557 SQL_SVE_NULLIF

14558 SQL_STANDARD_CLI_CONFORMANCE
14559 An SQLINTEGER bitmask enumerating the CLI standard(s) with which the implementation |
14560 complies. The following bitmasks are used to determine which levels the implementation |
14561 complies with:

14562 SQL_SCC_XOPEN_CLI_VERSION1
14563 The implementation complies with the X/Open CLI version 1. |

14564 SQL_SCC_ISO92_CLI
14565 The implementation complies with the ISO CLI International Standard. |

14566 SQL_STATIC_CURSOR_ATTRIBUTES1

14567 SQL_STATIC_CURSOR_ATTRIBUTES2
14568 A pair of 32-bit bitmasks that indicate supported operations for static cursors and describe
14569 other attributes of static cursors. See **Detecting Cursor Capabilities with SQLGetInfo()** on
14570 page 402.

14571 SQL_STRING_FUNCTIONS
14572 An SQLINTEGER bitmask enumerating the scalar string functions the implementation |
14573 supports. String functions are defined in Section F.1 on page 601.

14574 The following bitmasks are used to determine which string functions are supported:

14575 SQL_FN_STR_ASCII SQL_FN_STR_LTRIM
14576 SQL_FN_STR_BIT_LENGTH SQL_FN_STR_OCTET_LENGTH
14577 SQL_FN_STR_CHAR SQL_FN_STR_POSITION
14578 SQL_FN_STR_CHAR_LENGTH SQL_FN_STR_REPEAT
14579 SQL_FN_STR_CHARACTER_LENGTH SQL_FN_STR_REPLACE
14580 SQL_FN_STR_CONCAT SQL_FN_STR_RIGHT
14581 SQL_FN_STR_DIFFERENCE SQL_FN_STR_RTRIM
14582 SQL_FN_STR_INSERT SQL_FN_STR_SOUNDEX
14583 SQL_FN_STR_LCASE SQL_FN_STR_SPACE
14584 SQL_FN_STR_LEFT SQL_FN_STR_SUBSTRING
14585 SQL_FN_STR_LENGTH SQL_FN_STR_UCASE
14586 SQL_FN_STR_LOCATE

14587 If an application can call the LOCATE scalar function with the string_exp1, string_exp2, and
14588 start arguments, the implementation returns the SQL_FN_STR_LOCATE bitmask. If an
14589 application can call the LOCATE scalar function with only the string_exp1 and string_exp2

14590 arguments, the implementation returns the SQL_FN_STR_LOCATE_2 bitmask.
14591 Implementations that fully support the LOCATE scalar function return both bitmasks.

14592 SQL_SUBQUERIES
14593 An SQLINTEGER bitmask enumerating the predicates that support subqueries:

14594 SQL_SQ_CORRELATED_SUBQUERIES
14595 SQL_SQ_COMPARISON
14596 SQL_SQ_EXISTS
14597 SQL_SQ_IN
14598 SQL_SQ_QUANTIFIED

14599 The SQL_SQ_CORRELATED_SUBQUERIES bitmask indicates that all predicates that
14600 support subqueries support correlated subqueries.

14601 SQL_SYSTEM_FUNCTIONS
14602 An SQLINTEGER bitmask enumerating the scalar system functions the implementation |
14603 supports. These functions are defined in Section F.4 on page 608.

14604 The following bitmasks are used to determine which system functions are supported:

14605 SQL_FN_SYS_DBNAME
14606 SQL_FN_SYS_IFNULL
14607 SQL_FN_SYS_USERNAME

14608 SQL_TABLE_TERM
14609 A character string with the data source vendor's name for a table; for example, ''table'' or
14610 ''file''.

14611 SQL_TIMEDATE_ADD_INTERVALS
14612 An SQLINTEGER bitmask enumerating the timestamp intervals the implementation
14613 supports for the TIMESTAMPADD scalar function.

14614 The following bitmasks are used to determine which intervals are supported:

14615 SQL_FN_TSI_FRAC_SECOND
14616 SQL_FN_TSI_SECOND
14617 SQL_FN_TSI_MINUTE
14618 SQL_FN_TSI_HOUR
14619 SQL_FN_TSI_DAY
14620 SQL_FN_TSI_WEEK
14621 SQL_FN_TSI_MONTH
14622 SQL_FN_TSI_QUARTER
14623 SQL_FN_TSI_YEAR

14624 SQL_TIMEDATE_DIFF_INTERVALS
14625 An SQLINTEGER bitmask enumerating the timestamp intervals the implementation
14626 supports for the TIMESTAMPDIFF scalar function.

14627 The following bitmasks are used to determine which intervals are supported:

14628 SQL_FN_TSI_FRAC_SECOND
14629 SQL_FN_TSI_SECOND
14630 SQL_FN_TSI_MINUTE
14631 SQL_FN_TSI_HOUR
14632 SQL_FN_TSI_DAY
14633 SQL_FN_TSI_WEEK
14634 SQL_FN_TSI_MONTH
14635 SQL_FN_TSI_QUARTER
14636 SQL_FN_TSI_YEAR

14637 SQL_TIMEDATE_FUNCTIONS
14638 An SQLINTEGER bitmask enumerating the scalar date and time functions the |
14639 implementation supports. These functions are defined in Section F.3 on page 605.

14640 The following bitmasks are used to determine which date and time functions are supported:

14641 SQL_FN_TD_CURDATE SQL_FN_TD_MINUTE
14642 SQL_FN_TD_CURRENT_DATE SQL_FN_TD_MONTH
14643 SQL_FN_TD_CURRENT_TIME SQL_FN_TD_MONTHNAME
14644 SQL_FN_TD_CURRENT_TIMESTAMP SQL_FN_TD_NOW
14645 SQL_FN_TD_CURTIME SQL_FN_TD_QUARTER
14646 SQL_FN_TD_DAYNAME SQL_FN_TD_SECOND
14647 SQL_FN_TD_DAYOFMONTH SQL_FN_TD_TIMESTAMPADD
14648 SQL_FN_TD_DAYOFWEEK SQL_FN_TD_TIMESTAMPDIFF
14649 SQL_FN_TD_DAYOFYEAR SQL_FN_TD_WEEK
14650 SQL_FN_TD_EXTRACT SQL_FN_TD_YEAR
14651 SQL_FN_TD_HOUR

14652 SQL_TXN_CAPABLE
14653 An SQLSMALLINT value describing the extent to which the data source supports
14654 transactions:

14655 SQL_TC_NONE
14656 Transactions not supported.

14657 SQL_TC_DML
14658 Transactions can only contain Data Manipulation Language (DML) statements
14659 (SELECT, INSERT, UPDATE, DELETE). Data Definition Language (DDL) statements
14660 encountered in a transaction cause an error.

14661 SQL_TC_DDL_COMMIT
14662 Transactions can only contain DML statements. DDL statements (CREATE TABLE,
14663 DROP INDEX, an so on) encountered in a transaction cause the transaction to be
14664 committed.

14665 SQL_TC_DDL_IGNORE
14666 Transactions can only contain DML statements. DDL statements encountered in a
14667 transaction are ignored.

14668 SQL_TC_ALL
14669 Transactions can contain DDL statements and DML statements in any order.

14670 SQL_TXN_ISOLATION_OPTION
14671 An SQLINTEGER bitmask enumerating the transaction isolation levels the implementation |
14672 supports. This indicates the valid values to which the application can set the |
14673 SQL_ATTR_TXN_ISOLATION connection attribute.

14674 The following bitmasks are used in conjunction with the flag to determine which options
14675 are supported:

SQL_TXN_READ_UNCOMMITTED
SQL_TXN_READ_COMMITTED
SQL_TXN_REPEATABLE_READ
SQL_TXN_SERIALIZABLE

The above values correspond to the values for the SQL_DEFAULT_TXN_ISOLATION option described above. The terms used above for both the isolation levels and the isolation failure phenomena are defined in Section 14.2.2 on page 186.

SQL_UNION
An SQLINTEGER bitmask enumerating the support for the UNION clause:

SQL_U_UNION	The data source supports the UNION clause.
SQL_U_UNION_ALL	The data source supports the ALL keyword in the UNION clause. (*SQLGetInfo*() returns both SQL_U_UNION and SQL_U_UNION_ALL in this case.)

SQL_USER_NAME
A character string with the name used in a particular database, which can be different from login name.

SQL_XDBC_INTERFACE_CONFORMANCE
An SQLINTEGER value indicating the level of the XDBC interface to which the implementation complies:

SQL_OIC_CORE	claims XDBC Core-level compliance
SQL_OIC_LEVEL1	claims XDBC Level 1 compliance
SQL_OIC_LEVEL2	claims XDBC Level 2 compliance

These terms are defined in Section 1.7 on page 13.

SQL_XDBC_VER
A character string with the version of XDBC to which the implementation complies. The version is of the form ##.##.0000, where the first two digits are the major version and the next two digits are the minor version.

SQL_XOPEN_CLI_YEAR
A character string that indicates the year of publication of the X/Open specification with which the XDBC implementation fully complies.

Detecting Cursor Capabilities with SQLGetInfo()

Several values of *InfoType* select bitmasks that report the capabilities the implementation supports for various types of cursor.

These bitmasks come in pairs. The first of each pair is selected by a manifest constant that ends in 1 and bits within it can be identified using constants containing _CA1_. The second of each pair is selected by a manifest constant that ends in 2 and bits within it can be identified using constants containing _CA2_.

There are the following bitmasks:

SQL_DYNAMIC_CURSOR_ATTRIBUTES1	Indicates supported operations for dynamic cursors.
SQL_DYNAMIC_CURSOR_ATTRIBUTES2	Indicates other attributes of dynamic cursors.
SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES1	Indicates supported operations for forward-only cursors.
SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES2	Indicates other attributes of forward-only cursors.

14719 SQL_KEYSET_CURSOR_ATTRIBUTES1 — Indicates supported operations for keyset-driven
14720 cursors.

14721 SQL_KEYSET_CURSOR_ATTRIBUTES2 — Indicates other attributes of keyset-driven cursors.

14722 SQL_STATIC_CURSOR_ATTRIBUTES1 — Indicates supported operations for static cursors.

14723 SQL_STATIC_CURSOR_ATTRIBUTES2 — Indicates other attributes of static cursors.

14724 If the following bits are nonzero in an information item ending in 1, it means the implementation
14725 supports the operation listed below for the cursor type specified by the information item:

14726 SQL_CA1_NEXT
14727 *SQLFetchScroll*() with *FetchOrientation* = SQL_FETCH_NEXT

14728 SQL_CA1_ABSOLUTE
14729 *SQLFetchScroll*() with *FetchOrientation* = SQL_FETCH_FIRST, SQL_FETCH_LAST, or
14730 SQL_FETCH_ABSOLUTE. (These are fetches independent of the current cursor position.)

14731 SQL_CA1_RELATIVE
14732 *SQLFetchScroll*() with *FetchOrientation* = SQL_FETCH_PRIOR or SQL_FETCH_RELATIVE.
14733 (These are fetches that depend on the current cursor position. SQL_FETCH_NEXT is also a
14734 relative fetch but is not included in this category because forward-only cursors support
14735 SQL_FETCH_NEXT but do not support SQL_FETCH_PRIOR nor
14736 SQL_FETCH_RELATIVE.)

14737 SQL_CA1_BOOKMARK
14738 *SQLFetchScroll*() with *FetchOrientation* = SQL_FETCH_BOOKMARK

14739 SQL_CA1_LOCK_NO_CHANGE
14740 *SQLSetPos*() with *LockType* = SQL_LOCK_NO_CHANGE

14741 SQL_CA1_LOCK_UNLOCK
14742 *SQLSetPos*() with *LockType* = SQL_LOCK_UNLOCK

14743 SQL_CA1_POS_POSITION
14744 *SQLSetPos*() with *Operation* = SQL_POSITION

14745 SQL_CA1_POS_UPDATE
14746 *SQLSetPos*() with *Operation* = SQL_UPDATE

14747 SQL_CA1_POS_DELETE
14748 *SQLSetPos*() with *Operation* = SQL_DELETE

14749 SQL_CA1_POS_REFRESH
14750 *SQLSetPos*() with *Operation* = SQL_REFRESH

14751 SQL_CA1_POSITIONED_UPDATE
14752 The SQL statement UPDATE WHERE CURRENT OF

14753 SQL_CA1_POSITIONED_DELETE
14754 The SQL statement DELETE WHERE CURRENT OF

14755 SQL_CA1_SELECT_FOR_UPDATE
14756 The SQL statement SELECT FOR UPDATE

14757 SQL_CA1_BULK_ADD
14758 *SQLBulkOperations*() with *Operation* = SQL_ADD

14759 SQL_CA1_BULK_UPDATE_BY_BOOKMARK
14760 *SQLBulkOperations*() with *Operation* = SQL_UPDATE_BY_BOOKMARK

14761 SQL_CA1_BULK_DELETE_BY_BOOKMARK
14762 *SQLBulkOperations*() with *Operation* = SQL_DELETE_BY_BOOKMARK

14763 SQL_CA1_BULK_FETCH_BY_BOOKMARK
14764 *SQLBulkOperations*() with *Operation* = SQL_REFRESH_BY_BOOKMARK

14765 If the following bits are nonzero in an information item ending in 2, it makes the assertion listed
14766 below for the cursor type specified by the information item:

14767 SQL_CA2_READ_ONLY_CONCUR
14768 The implementation supports read-only cursors, in which no updates are allowed. (The
14769 SQL_ATTR_CONCURRENCY statement attribute can be SQL_CONCUR_READ_ONLY.)

14770 SQL_CA2_LOCK_CONCURRENCY
14771 The implementation supports a cursor that uses the lowest level of locking sufficient to
14772 ensure that the row can be updated is supported. (The SQL_ATTR_CONCURRENCY
14773 statement attribute can be SQL_CONCUR_LOCK.)

14774 SQL_CA2_OPT_ROWVER_CONCURRENCY
14775 The implementation supports a cursor that uses the optimistic concurrency control
14776 comparing row versions. (The SQL_ATTR_CONCURRENCY statement attribute can be
14777 SQL_CONCUR_ROWVER.)

14778 SQL_CA2_OPT_VALUES_CONCURRENCY
14779 The implementation supports cursors that use the optimistic concurrency control
14780 comparing values. (The SQL_ATTR_CONCURRENCY statement attribute can be
14781 SQL_CONCUR_VALUES.)

14782 SQL_CA2_SENSITIVITY_ADDITIONS
14783 Added rows are visible to the cursor; the cursor can scroll to those rows. (Where these rows
14784 are added to the cursor is implementation-defined.)

14785 SQL_CA2_SENSITIVITY_DELETIONS
14786 Deleted rows are no longer available through the cursor, and do not leave a ‘‘hole’’ in the
14787 result set; after the cursor scrolls from a deleted row, it cannot return there.

14788 SQL_CA2_SENSITIVITY_UPDATES
14789 Updates to rows are visible through the cursor; if the cursor scrolls from and returns to an
14790 updated row, the data returned by the cursor is the updated data, not the original data.

14791 SQL_CA2_MAX_ROWS_SELECT
14792 The SQL_ATTR_MAX_ROWS statement attribute affects SELECT statements.

14793 SQL_CA2_MAX_ROWS_INSERT
14794 The SQL_ATTR_MAX_ROWS statement attribute affects INSERT statements.

14795 SQL_CA2_MAX_ROWS_DELETE
14796 The SQL_ATTR_MAX_ROWS statement attribute affects DELETE statements.

14797 SQL_CA2_MAX_ROWS_UPDATE
14798 The SQL_ATTR_MAX_ROWS statement attribute affects UPDATE statements.

14799 SQL_CA2_MAX_ROWS_CATALOG
14800 The SQL_ATTR_MAX_ROWS statement attribute affects CATALOG result sets.

14801 SQL_CA2_MAX_ROWS_AFFECTS_ALL
14802 The SQL_ATTR_MAX_ROWS statement attribute affects SELECT, INSERT, DELETE, and
14803 UPDATE statements, and CATALOG result sets.

14804 SQL_CA2_CRC_EXACT
14805 The exact row count is available in the SQL_DIAG_CURSOR_ROW_COUNT diagnostic
14806 field.

14807 SQL_CA2_CRC_APPROXIMATE
14808 An approximate row count is available in the SQL_DIAG_CURSOR_ROW_COUNT
14809 diagnostic field.

14810 The following three bitmasks indicate the implementation's ability to simulate positioned
14811 UPDATE and DELETE statements, and therefore indicate the valid values to which the
14812 application can set the SQL_ATTR_SIMULATE_CURSORstatement attribute:

14813 SQL_CA2_SIMULATE_NON_UNIQUE
14814 The implementation does not guarantee that simulated positioned UPDATE or DELETE
14815 statements affect only one row; the application must provide for this. (If a statement affects
14816 more than one row, *SQLExecute*() and *SQLExecDirect*() return SQLSTATE 01001 (Cursor
14817 operation conflict).)

14818 SQL_CA2_SIMULATE_TRY_UNIQUE
14819 The implementation tries to guarantee that simulated positioned UPDATE or DELETE
14820 statements affect only one row. The implementation always executes such statements, even
14821 if they might affect more than one row, such as when there is no unique key. (If a statement
14822 affects more than one row, *SQLExecute*() and *SQLExecDirect*() return SQLSTATE 01001
14823 (Cursor operation conflict).

14824 SQL_CA2_SIMULATE_UNIQUE
14825 The implementation either has true support for positioned UPDATE and DELETE
14826 statements, or guarantees that its simulation of those statements affects only one row. If the
14827 implementation cannot guarantee this for a given statement, *SQLExecDirect*() and
14828 *SQLPrepare*() return SQLSTATE01001 (Cursor operation conflict).

14829 **SEE ALSO**

14830 **For information about**	**See**
14831 Returning the setting of a connection attribute	*SQLGetConnectAttr*()
14832 Determining if a function is implemented	*SQLGetFunctions*()
14833 Returning the setting of a statement attribute	*SQLGetStmtAttr*()
14834 Returning information about a data source's data types	*SQLGetTypeInfo*()

14835 **CHANGE HISTORY**

14836 **Version 2**

14837 Revised generally. See **Alignment with Popular Implementations** on page 2.

14838 **Changes to Information Items in SQLGetInfo()**

14839 The following options are new in this issue:

14840 SQL_ACCESSIBLE_PROCEDURES	SQL_KEYWORDS
14841 SQL_ACTIVE_ENVIRONMENTS	SQL_LIKE_ESCAPE_CLAUSE
14842 SQL_ALTER_DOMAIN	SQL_MAX_ASYNC_CONCURRENT_STATEMENTS
14843 SQL_ANSI_SQL_CONFORMANCE	SQL_MAX_BINARY_LITERAL_LEN
14844 SQL_ANSI_SQL_DATETIME_LITERALS	SQL_MAX_CHAR_LITERAL_LEN
14845 SQL_ASYNC_MODE	SQL_MAX_PROCEDURE_NAME_LEN
14846 SQL_BATCH_ROW_COUNT	SQL_MAX_ROW_SIZE_INCLUDES_LONG
14847 SQL_BATCH_SUPPORT	SQL_MULTIPLE_ACTIVE_TXN
14848 SQL_BOOKMARK_PERSISTENCE	SQL_MULT_RESULT_SETS

SQL_CATALOG_LOCATION
SQL_CATALOG_NAME_SEPARATOR
SQL_CATALOG_TERM
SQL_CATALOG_USAGE
SQL_COLUMN_ALIAS
SQL_CONCAT_NULL_BEHAVIOR
SQL_CONVERT_*
SQL_CONVERT_FUNCTIONS
SQL_CORRELATION_NAME
SQL_CREATE_ASSERTION
SQL_CREATE_CHARACTER_SET
SQL_CREATE_COLLATION
SQL_CREATE_DOMAIN
SQL_CREATE_SCHEMA
SQL_CREATE_TABLE
SQL_CREATE_TRANSLATION
SQL_CREATE_VIEW
SQL_CURSOR_ROLLBACK_BEHAVIOR
SQL_DATABASE_NAME
SQL_DROP_ASSERTION
SQL_DROP_CHARACTER_SET
SQL_DROP_COLLATION
SQL_DROP_DOMAIN
SQL_DROP_SCHEMA
SQL_DROP_TABLE
SQL_DROP_TRANSLATION
SQL_DROP_VIEW
SQL_DYNAMIC_CURSOR_ATTRIBUTES1
SQL_DYNAMIC_CURSOR_ATTRIBUTES2
SQL_EXPRESSIONS_IN_ORDERBY
SQL_FILE_USAGE
SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES1
SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES2
SQL_GROUP_BY
SQL_INDEX_KEYWORDS
SQL_INFO_SCHEMA_VIEWS
SQL_KEYSET_CURSOR_ATTRIBUTES1
SQL_KEYSET_CURSOR_ATTRIBUTES2
SQL_NEED_LONG_DATA_LEN
SQL_NON_NULLABLE_COLUMNS
SQL_NUMERIC_FUNCTIONS
SQL_OUTER_JOINS
SQL_PARAM_ARRAY_ROW_COUNTS
SQL_PARAM_ARRAY_SELECTS
SQL_PROCEDURES
SQL_PROCEDURE_TERM
SQL_QUOTED_IDENTIFIER_CASE
SQL_REVOKE
SQL_ROW_UPDATES
SQL_SCHEMA_TERM
SQL_SCHEMA_USAGE
SQL_SCROLL_OPTIONS
SQL_SQL92_DATETIME_FUNCTIONS
SQL_SQL92_FOREIGN_KEY_DELETE_RULE
SQL_SQL92_FOREIGN_KEY_UPDATE_RULE
SQL_SQL92_GRANT
SQL_SQL92_NUMERIC_VALUE_FUNCTIONS
SQL_SQL92_PREDICATES
SQL_SQL92_RELATIONAL_JOIN_OPERATORS
SQL_SQL92_REVOKE
SQL_SQL92_ROW_VALUE_CONSTRUCTOR
SQL_SQL92_STRING_FUNCTIONS
SQL_SQL92_VALUE_EXPRESSIONS
SQL_STANDARD_CLI_CONFORMANCE
SQL_STATIC_CURSOR_ATTRIBUTES1
SQL_STATIC_CURSOR_ATTRIBUTES2
SQL_STRING_FUNCTIONS
SQL_SUBQUERIES
SQL_SYSTEM_FUNCTIONS
SQL_TABLE_TERM
SQL_TIMEDATE_ADD_INTERVALS
SQL_TIMEDATE_DIFF_INTERVALS
SQL_TIMEDATE_FUNCTIONS
SQL_UNION
SQL_XDBC_INTERFACE_CONFORMANCE
SQL_XDBC_VER

DE The following options are deprecated: SQL_FETCH_DIRECTION and SQL_SCROLL_CONCURRENCY. See **Detecting Cursor Capabilities with SQLGetInfo()** on page 402 for the preferred technique.

14890 **NAME**

14891 SQLGetStmtAttr — Return the current setting of a statement attribute.

14892 **SYNOPSIS**

```
14893 SQLRETURN SQLGetStmtAttr(
14894   SQLHSTMT StatementHandle,
14895   SQLINTEGER Attribute,
14896   SQLPOINTER ValuePtr,
14897   SQLINTEGER BufferLength,
14898   SQLINTEGER * StringLengthPtr);
```

14899 **ARGUMENTS**

14900 *StatementHandle* [Input]
14901 Statement handle.

14902 *Attribute* [Input]
14903 Attribute to retrieve.

14904 *ValuePtr* [Output]
14905 Pointer to a buffer in which to return the value of the attribute specified in *Attribute*.

14906 *BufferLength* [Input]
14907 If *ValuePtr* points to data of variable length, this argument should be the length of **ValuePtr*.
14908 If what is contained in *ValuePtr* is itself a pointer, but not to data of variable length, then
14909 *BufferLength* should have the value SQL_IS_POINTER. If what is contained in *ValuePtr* is
14910 actual data of fixed length, then *BufferLength* should have the value
14911 SQL_IS_NOT_POINTER.

14912 *StringLengthPtr* [Output]
14913 A pointer to a buffer in which to return the total number of octets (excluding the null
14914 terminator) available to return in **ValuePtr*. If *ValuePtr* is a null pointer, no length is
14915 returned. If the attribute value is a character string, and the number of octets available to
14916 return is greater than or equal to *BufferLength*, the data in **ValuePtr* is truncated to
14917 *BufferLength* minus the length of a null terminator and is null-terminated,

14918 **RETURN VALUE**

14919 SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.

14920 **DIAGNOSTICS**

14921 When *SQLGetStmtAttr*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated
14922 SQLSTATE value can be obtained by calling *SQLGetDiagRec*() with a *HandleType* of
14923 SQL_HANDLE_STMT and a *Handle* of *StatementHandle*. The following SQLSTATE values are
14924 commonly returned by *SQLGetStmtAttr*(). The return code associated with each SQLSTATE
14925 value is SQL_ERROR, except that for SQLSTATE values in class 01, the return code is
14926 SQL_SUCCESS_WITH_INFO.

14927 01000 — General warning
14928 Implementation-defined informational message.

14929 01004 — String data, right truncation
14930 The data returned in **ValuePtr* was truncated to be *BufferLength* minus the length of a null
14931 terminator. The length of the untruncated string value is returned in **StringLengthPtr*.

14932 24000 — Invalid cursor state
14933 *Attribute* was SQL_ATTR_ROW_NUMBER and the cursor was not open, or the cursor was
14934 positioned before the start of the result set or after the end of the result set.

14935 HY000 — General error
14936 An error occurred for which there was no specific SQLSTATE and for which no

14937 implementation-specific SQLSTATE was defined. The error message returned by
14938 *SQLGetDiagRec*() in *MessageText* describes the error and its cause.

14939 HY001 — Memory allocation error
14940 The implementation failed to allocate memory required to support execution or completion
14941 of the function.

14942 HY010 — Function sequence error
14943 An asynchronously executing function was called for *StatementHandle* and was still
14944 executing when this function was called.

14945 *SQLBulkOperations*(), *SQLExecDirect*(), *SQLExecute*(), or *SQLSetPos*() was called for
14946 *StatementHandle* and returned SQL_NEED_DATA. This function was called before data was
14947 sent for all data-at-execution parameters or columns.

14948 HY092 — Invalid attribute identifier
14949 *Attribute* was not valid for this connection to this data source.

14950 HY109 — Invalid cursor position
14951 *Attribute* was SQL_ATTR_ROW_NUMBER and the the row had been deleted or could not
14952 be fetched.

14953 HYC00 — Optional feature not implemented
14954 *Attribute* was a valid statement attribute but is not supported by the data source.

14955 HYT01 — Connection timeout expired
14956 The connection timeout period expired before the data source responded to the request. The
14957 connection timeout period is set through *SQLSetConnectAttr*(),
14958 SQL_ATTR_CONNECTION_TIMEOUT.

14959 IM001 — Function not supported
14960 The function is not supported on the current connection to the data source.

14961 **COMMENTS**

14962 A call to *SQLGetStmtAttr*() returns in **ValuePtr* the value of the statement attribute specified in
14963 *Attribute*. That value can either be a 32-bit value or a null-terminated character string. If the
14964 value is a null-terminated string, the application specifies the maximum length of that string in
14965 *BufferLength*, and the implementation returns the length of that string in the **StringLengthPtr*
14966 buffer. If the value is a 32-bit value, *BufferLength* and *StringLengthPtr* are not used.

14967 The following statement attributes retrieve descriptor header fields:

14968 SQL_ATTR_BIND_OFFSET SQL_ATTR_PREDICATE_OCTET_LENGTH_PTR
14969 SQL_ATTR_BIND_TYPE SQL_ATTR_PREDICATE_PTR
14970 SQL_ATTR_FETCH_BOOKMARK_PTR SQL_ATTR_ROWS_FETCHED_PTR
14971 SQL_ATTR_PARAMETER_BIND_TYPE SQL_ATTR_ROWS_PROCESSED_PTR
14972 SQL_ATTR_PARAMSET_SIZE SQL_ATTR_ROW_ARRAY_SIZE
14973 SQL_ATTR_PARAM_STATUS_PTR SQL_ATTR_ROW_STATUS_PTR

14974 The following statement attributes are read-only: They can be retrieved by *SQLGetStmtAttr*(),
14975 but not set by *SQLSetStmtAttr*(). For a list of attributes that can be set and retrieved, see
14976 *SQLSetStmtAttr*().

14977 SQL_ATTR_IMP_PARAM_DESC SQL_ATTR_ROW_NUMBER

14978 SQL_ATTR_IMP_ROW_DESC

14979 **SEE ALSO**

For information about	See
Returning the setting of a connection attribute	*SQLGetConnectAttr*()
Setting a connection attribute	*SQLSetConnectAttr*()
Setting a statement attribute	*SQLSetStmtAttr*()

14984 **CHANGE HISTORY**

14985 **Version 2**

14986 Revised generally. See **Alignment with Popular Implementations** on page 2. See also the list in
14987 **New Statement Attributes in Version 2** on page 515.

14988 **NAME**
14989 SQLGetTypeInfo — Return information about data types supported by the data source.

14990 **SYNOPSIS**
```
14991 SQLRETURN SQLGetTypeInfo(
14992   SQLHSTMT StatementHandle,
14993   SQLSMALLINT DataType);
```

14994 **ARGUMENTS**

14995 *StatementHandle* [Input]
14996 Statement handle for the result set.

14997 *DataType* [Input]
14998 The SQL data type. **Applications must use the type names returned in the TYPE_NAME**
14999 **column of the result set returned by this function in any ALTER TABLE and CREATE**
15000 **TABLE statements.** Valid values are listed in Section D.1 on page 556. A value of
15001 SQL_ALL_TYPES requests information about all data types.

15002 **RETURN VALUE**
15003 SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or
15004 SQL_INVALID_HANDLE.

15005 **DIAGNOSTICS**
15006 When *SQLGetTypeInfo*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated
15007 SQLSTATE value may be obtained by calling *SQLGetDiagRec*() with a *HandleType* of
15008 SQL_HANDLE_STMT and a *Handle* of *StatementHandle*. The following table lists the SQLSTATE
15009 values commonly returned by *SQLGetTypeInfo*(). The return code associated with each
15010 SQLSTATEvalue is SQL_ERROR, except that for SQLSTATEvalues in class 01, the return code is
15011 SQL_SUCCESS_WITH_INFO.

15012 01000 — General warning
15013 Implementation-defined informational message.

15014 01S02 — Attribute value changed
15015 A specified statement attribute was invalid and a similar value was temporarily substituted.
15016 See Section 9.2.1 on page 93.

15017 08S01 — Communication link failure
15018 The communication link to the data source failed before the function completed processing.

15019 24000 — Invalid cursor state
15020 A cursor was open on *StatementHandle*.

15021 40001 — Serialization failure
15022 The transaction in which the fetch was executed was terminated to prevent deadlock.

15023 HY000 — General error
15024 An error occurred for which there was no specific SQLSTATE and for which no
15025 implementation-specific SQLSTATE was defined. The error message returned by
15026 *SQLGetDiagRec*() in the **MessageText*buffer describes the error and its cause.

15027 HY001 — Memory allocation error
15028 The implementation failed to allocate memory required to support execution or completion
15029 of the function.

15030 HY004 — Invalid SQL data type
15031 *DataType*was neither a valid XDBC data type identifier nor an implementation-defined SQL
15032 data type identifier that the data source supports.

HY008 — Operation canceled

Asynchronous processing was enabled for *StatementHandle*. The function was called and before it completed execution, *SQLCancel*() was called on *StatementHandle*. The function was then called again on *StatementHandle*.

The function was called and, before it completed execution, *SQLCancel*() was called on *StatementHandle* from a different thread in a multithread application.

HY010 — Function sequence error

An asynchronously executing function (not this one) was called for *StatementHandle* and was still executing when this function was called.

SQLBulkOperations(), *SQLExecDirect*(), *SQLExecute*(), or *SQLSetPos*() was called for *StatementHandle* and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.

HYC00 — Optional feature not implemented

The value specified for *DataType* is a valid XDBC SQL data type identifier but is not supported by the implementation.

The data source does not support the combination of the current settings of the SQL_ATTR_CONCURRENCY and SQL_ATTR_CURSOR_TYPE statement attributes.

The SQL_ATTR_USE_BOOKMARKS statement attribute was set to SQL_UB_VARIABLE, and the SQL_ATTR_CURSOR_TYPE statement attribute was set to a cursor type for which the data source does not support bookmarks.

HYT00 — Timeout expired

The query timeout period expired before the data source returned the result set. The timeout period is set through *SQLSetStmtAttr*(), SQL_ATTR_QUERY_TIMEOUT.

HYT01 — Connection timeout expired

The connection timeout period expired before the data source responded to the request. The connection timeout period is set through *SQLSetConnectAttr*(), SQL_ATTR_CONNECTION_TIMEOUT.

IM001 — Function not supported

The function is not supported on the current connection to the data source.

COMMENTS

SQLGetTypeInfo() returns information on a specified data type in the form of an SQL result set. The data types are intended for use in Data Definition Language (DDL) statements.

SQLGetTypeInfo() may return more than one row with the same value in the DATA_TYPE column.

SQLGetTypeInfo() returns the results as a standard result set, ordered by DATA_TYPE and then by how closely the data type maps to the corresponding XDBC SQL data type. Data types defined by the data source take precedence over user-defined data types. For example, suppose that a data source defined INTEGER and COUNTER data types, where COUNTER is auto-incrementing, and that a user-defined data type WHOLENUM has also been defined. These would be returned in the order INTEGER, WHOLENUM, and COUNTER, because WHOLENUM maps closely to the XDBC SQL data type SQL_INTEGER, while the auto-incrementing data type, even though supported by the data source, does not map closely to an XDBC SQL data type.

The following table lists the columns in the result set. Additional columns beyond column 18 (NUM_PREC_RADIX) can be defined by the implementation. An application should gain access to implementation-defined columns by counting down from the end of the result set rather than by specifying an explicit ordinal position; see Section 7.3 on page 68.

15080 The data types returned by *SQLGetTypeInfo*() are those supported by the data source. They are
15081 intended for use in Data Definition Language (DDL) statements. Implementations can return
15082 result set data using data types other than the types returned by *SQLGetTypeInfo*(). In creating
15083 the result set for a catalog function, the implementation can use a data type that is not supported
15084 by the data source.

Column Name	Col. No.	Data Type	Comments
TYPE_NAME	1	Varchar not NULL	Data source-dependent data type name; for example, ‘‘CHAR()’’, ‘‘VARCHAR()’’, ‘‘MONEY’’, ‘‘LONG VARBINARY’’, or ‘‘CHAR ( ) FOR BIT DATA’’. Applications must use this name in CREATE TABLE and ALTER TABLE statements.
DATA_TYPE	2	Smallint not NULL	SQL data type. This can be an XDBC SQL data type or an implementation-defined SQL data type. For date/time or interval data types, this column returns the concise data type (for example, SQL_TYPE_TIME or SQL_INTERVAL_YEAR_TO_MONTH). For a list of valid XDBC SQL data types, see Section D.1 on page 556.
COLUMN_SIZE	3	Integer	The maximum column size that the server supports for this data type. For numeric data, this is the maximum precision. For string data, this is the length in characters. For date/time data types, this is the length in characters of the string representation (assuming the maximum allowed precision of the fractional seconds component.) NULL is returned for data types where column size is not applicable. For interval data types, this is the number of characters in the character representation of the interval literal (as defined by the interval leading precision, see **Interval Data Type Length** on page 571). For more information on column size, see Section D.3 on page 562.
LITERAL_PREFIX	4	Varchar	Character or characters used to prefix a literal; for example, a single quotation mark for character data types or 0x for binary data types; NULL is returned for data types where a literal prefix is not applicable.

LITERAL_SUFFIX	5	Varchar	Character or characters used to terminate a literal; for example, a single quotation mark for character data types; NULL is returned for data types where a literal suffix is not applicable.
CREATE_PARAMS	6	Varchar	A list of keywords, separated by commas, corresponding to each parameter that the application may specify in parentheses when using the name that is returned in the TYPE_NAME field. The keywords in the list can be any of the following: length, precision, scale. They appear in the order that the syntax requires that they be used. For example, CREATE_PARAMS for DECIMAL would be ‘‘precision,scale’’; CREATE_PARAMS for VARCHAR would equal ‘‘length’’. NULL is returned if there are no parameters for the data type definition, for example INTEGER. The language used for the CREATE_PARAMS text is locale-dependent.
NULLABLE	7	Smallint not NULL	Whether the data type accepts a NULL value: SQL_NO_NULLS if the data type does not accept NULL values. SQL_NULLABLE if the data type accepts NULL values. SQL_NULLABLE_UNKNOWN if it is not known if the column accepts NULL values.
CASE_SENSITIVE	8	Smallint not NULL	Whether a character data type is case-sensitive in collations and comparisons: SQL_TRUE if the data type is a character data type and is case-sensitive. SQL_FALSE if the data type is not a character data type or is not case-sensitive.

15157 15158	SEARCHABLE	9	Smallint not NULL	How the data type is used in a WHERE clause:
15159 15160				SQL_PRED_NONE if the column cannot be used in a WHERE clause.
15161 15162 15163				SQL_PRED_CHAR if the column can be used in a WHERE clause, but only with the LIKE predicate.
15164 15165 15166 15167 15168 15169				SQL_PRED_BASIC if the column can be used in a WHERE clause with all the comparison operators except LIKE (comparison, quantified comparison, BETWEEN, DISTINCT, IN, MATCH, and UNIQUE).
15170 15171 15172				SQL_SEARCHABLE if the column can be used in a WHERE clause with any comparison operator.
15173 15174 15175 15176 15177	UNSIGNED_ATTRIBUTE	10	Smallint	SQL_TRUE if the data type is unsigned; SQL_FALSE if the data type is signed. NULL is returned if the attribute is not applicable to the data type or the data type is not numeric.
15178 15179 15180 15181 15182	FIXED_PREC_SCALE	11	Smallint not NULL	SQL_TRUE if the data type has predefined fixed precision and scale (which are data-source specific), like a money data type. SQL_FALSE if it does not have predefined fixed precision and scale.
15183 15184 15185 15186 15187	AUTO_UNIQUE_VALUE	12	Smallint	SQL_TRUE if the data type is autoincrementing. SQL_FALSE if the data type is not autoincrementing. NULL is returned if the attribute is not applicable to the data type or the data type is not numeric.
15188 15189 15190				An application can insert values into a column having this attribute, but typically cannot update the values in the column.
15191 15192 15193 15194 15195 15196 15197				When an insert is made into an auto-increment column, a unique value is inserted into the column at insert time. The increment is data-source-specific. An application should not assume that an auto-increment column starts at any particular point or increments by any particular value.

LOCAL_TYPE_NAME	13	Varchar	Localized version of the data source-dependent name of the data type. NULL is returned if a localized name is not supported by the data source. This name is intended for display only, such as in dialog boxes.
MINIMUM_SCALE	14	Smallint	The minimum scale of the data type on the data source. If a data type has a fixed scale, the MINIMUM_SCALE and MAXIMUM_SCALE columns both contain this value. For example, an SQL_TYPE_TIMESTAMP column might have a fixed scale for fractional seconds. NULL is returned where scale is not applicable. For more information, see Section D.3 on page 562.
MAXIMUM_SCALE	15	Smallint	The maximum scale of the data type on the data source. NULL is returned where scale is not applicable. If the maximum scale is not defined separately on the data source, but is instead defined to be the same as the maximum precision, this column contains the same value as the COLUMN_SIZE column. For more information, see Section D.3 on page 562.
SQL_DATA_TYPE	16	Smallint not NULL	The value of the SQL data type as it appears in the SQL_DESC_TYPE field of the descriptor. This column is the same as the DATA_TYPE column, except for interval and date/time data types. For interval and date/time data types, the SQL_DATA_TYPE field in the result set returns SQL_INTERVAL or SQL_DATETIME, and the SQL_DATETIME_SUB field returns the subcode for the specific interval or date/time data type (see **Data Type Identification in Descriptors** on page 574).

SQL_DATETIME_SUB	17	Smallint	When the value of SQL_DATA_TYPE is SQL_DATETIME or SQL_INTERVAL, this column contains the subcode. For data types other than date/time and interval, this field is NULL. For interval or date/time data types, the SQL_DATA_TYPE field in the result set returns SQL_INTERVAL or SQL_DATETIME, and the SQL_DATETIME_SUB field returns the subcode for the specific interval or date/time data type (see Appendix D).
NUM_PREC_RADIX	18	Smallint	If the data type is an approximate numeric type, this column contains the value 2 to indicate that COLUMN_SIZE specifies a number of bits. For exact numeric types, this column contains the value 10 to indicate that COLUMN_SIZE specifies a number of decimal digits. Otherwise, this column is NULL.
INTERVAL_PRECISION	19	Smallint	If the data type is an interval data type, then this column contains the value of the interval leading precision (see **Interval Precision** on page 571). Otherwise, this column is NULL.

Attribute information can apply to data types or to specific columns in a result set. *SQLGetTypeInfo*() returns information about attributes associated with data types; *SQLColAttribute*() returns information about attributes associated with columns in a result set.

SEE ALSO

For information about	See
Overview of catalog functions	Chapter 7
Binding a buffer to a column in a result set	*SQLBindCol*()
Canceling statement processing	*SQLCancel*()
Returning information about a column in a result set	*SQLColAttribute*()
Fetching a block of data or scrolling through a result set	*SQLFetchScroll*()
Fetching a single row or a block of data in a forward-only direction	*SQLFetch*()
Returning information about an implementation	*SQLGetInfo*()

CHANGE HISTORY

Version 2

Revised generally. See **Alignment with Popular Implementations** on page 2.

15277 **NAME**

15278 SQLMoreResults — Determine whether there are more results available on a statement
15279 containing SELECT, UPDATE, INSERT, or DELETE statements and, if so, initialize processing for
15280 those results.

15281 **SYNOPSIS**

```
15282 SQLRETURN SQLMoreResults(
15283   SQLHSTMT StatementHandle);
```

15284 **ARGUMENTS**

15285 *StatementHandle* [Input]
15286 Statement handle.

15287 **RETURN VALUE**

15288 SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_NO_DATA,
15289 SQL_ERROR, or SQL_INVALID_HANDLE.

15290 **DIAGNOSTICS**

15291 When *SQLMoreResults*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated
15292 SQLSTATE value can be obtained by calling *SQLGetDiagRec*() with a *HandleType* of
15293 SQL_HANDLE_STMT and a *Handle* of *StatementHandle*. The following table lists the SQLSTATE
15294 values commonly returned by *SQLMoreResults*(). The return code associated with each
15295 SQLSTATEvalue is SQL_ERROR, except that for SQLSTATEvalues in class 01, the return code is
15296 SQL_SUCCESS_WITH_INFO.

15297 01000 — General warning
15298 Implementation-defined informational message.

15299 01S02 — Attribute value changed
15300 A specified statement attribute was invalid and a similar value was temporarily substituted.
15301 See Section 9.2.1 on page 93.

15302 08S01 — Communication link failure
15303 The communication link to the data source failed before the function completed processing.

15304 HY000 — General error
15305 An error occurred for which there was no specific SQLSTATE and for which no
15306 implementation-specific SQLSTATE was defined. The error message returned by
15307 *SQLGetDiagRec*() in the **MessageText* buffer describes the error and its cause.

15308 HY001 — Memory allocation error
15309 The implementation failed to allocate memory required to support execution or completion
15310 of the function.

15311 HY008 — Operation canceled
15312 Asynchronous processing was enabled for *StatementHandle*. The function was called and
15313 before it completed execution, *SQLCancel*() was called on *StatementHandle*. The function
15314 was then called again on *StatementHandle*.

15315 The function was called and, before it completed execution, *SQLCancel*() was called on
15316 *StatementHandle* from a different thread in a multithread application.

15317 HY010 — Function sequence error
15318 An asynchronously executing function (not this one) was called for *StatementHandle* and
15319 was still executing when this function was called.

15320 *SQLBulkOperations*(), *SQLExecDirect*(), *SQLExecute*(), or *SQLSetPos*() was called for
15321 *StatementHandle* and returned SQL_NEED_DATA. This function was called before data was
15322 sent for all data-at-execution parameters or columns.

15323 HYT01 — Connection timeout expired
15324 The connection timeout period expired before the data source responded to the request. The
15325 connection timeout period is set through *SQLSetConnectAttr*(),
15326 SQL_ATTR_CONNECTION_TIMEOUT.

15327 IM001 — Function not supported
15328 The function is not supported on the current connection to the data source.

15329 **COMMENTS**

15330 SELECT statements return result sets. UPDATE, INSERT, and DELETE statements return a count
15331 of affected rows. If any of these statements are batched, submitted with arrays of parameters
15332 (numbered from left to right, in the order that they appear in the batch), or in procedures, they
15333 can return multiple result sets or counts.

15334 After executing the batch, the application is positioned on the first result set. The application can
15335 call *SQLBindCol*(), *SQLBulkOperations*(), *SQLFetch*(), *SQLFetchScroll*(), *SQLGetData*(), |
15336 *SQLSetPos*(), and all the catalog functions, on the first or any subsequent result sets, just as it
15337 would if there were just a single result set. Once it is done with the first result set, the application
15338 calls *SQLMoreResults*() to move to the next result set. If another result set or count is available,
15339 *SQLMoreResults*() returns SQL_SUCCESS and initializes the result set or count for additional |
15340 processing. If there are any row-count-generating statements in between result-set-generating |
15341 statements, they can be stepped over by calling *SQLMoreResults*(). After calling
15342 *SQLMoreResults*() for UPDATE, INSERT, or DELETE statements, an application can call
15343 *SQLRowCount*().

15344 If all results have been processed, *SQLMoreResults*() returns SQL_NO_DATA. If there was a
15345 current result set with unfetched rows, *SQLMoreResults*() discards that result set and makes the
15346 next result set or count available.

15347 Any bindings that were established for the previous result set still remain valid. If the column
15348 structures are different for this result set, then calling *SQLFetch*() or *SQLFetchScroll*() may result
15349 in an error or truncation. To prevent this, the application has to call *SQLBindCol*() to explicitly
15350 rebind as appropriate (or do so by setting descriptor fields). Alternatively, the application can
15351 call *SQLFreeStmt*() with an *Option* of SQL_UNBIND to unbind all the column buffers.

15352 The values of statement attributes such as cursor type, cursor concurrency, keyset size, or
15353 maximum length, may change as the application navigates through the batch by calls to
15354 *SQLMoreResults*(). If this happens, *SQLMoreResults*() returns SQL_SUCCESS_WITH_INFO and
15355 SQLSTATE 01S02 (Attribute value changed).

15356 Calling *SQLCloseCursor*(), or *SQLFreeStmt*() with an *Option* of SQL_CLOSE, discards all the
15357 result sets and row counts that were available as a result of the execution of the batch. The
15358 statement handle returns to either the allocated or prepared state. Calling *SQLCancel*() to cancel
15359 an asynchronously executing function when a batch has been executed and the statement handle
15360 is in the executed, cursor-positioned, or asynchronous state results in all the results sets and row
15361 counts generated by the batch being discarded if the cancel call was successful. The statement
15362 then returns to the prepared or allocated state.

15363 If a batch of statements or a procedure mixes other SQL statements with SELECT, UPDATE,
15364 INSERT, and DELETE statements, these other statements do not affect *SQLMoreResults*(). |

15365 If a searched UPDATE or DELETE statement in a batch of statements does not affect any rows at |
15366 the data source, *SQLMoreResults*() returns SQL_SUCCESS and any call to *SQLRowCount*() |
15367 returns SQL_NO_DATA. This is different from the case of a searched UPDATE or DELETE |
15368 statement that is executed through *SQLExecDirect*(), *SQLExec*(), or *SQLParamData*(), which |
15369 returns SQL_NO_DATA if it does not affect any rows at the data source.

For additional information about the valid sequencing of result-processing functions, see Appendix B.

Availability of Row Counts

When a batch contains multiple consecutive row-count generating statements, it is possible that these row counts are rolled up into just one row count. For example, if a batch has five insert statements, then certain data sources are capable of returning five individual row counts. Certain other data sources return only one row count that represents the sum of the five individual row counts.

When a batch contains a combination of result-set-generating and row-count-generating statements, row counts might not be available. The application can determine their availability by calling *SQLGetInfo*() with the SQL_BATCH_ROW_COUNT option. For example, suppose that the batch contains a SELECT, followed by two INSERTs and another SELECT. Then the following cases are possible:

- The row counts corresponding to the two INSERT statements are not available at all. The first call to *SQLMoreResults*() positions the cursor on the result set of the second SELECT statement.
- The row counts corresponding to the two insert statements are available individually. (A call to *SQLGetInfo*() does not return the SQL_BRC_ROLLED_UP bit for the SQL_BATCH_ROW_COUNT option). The first call to *SQLMoreResults*() positions the cursor on the row count of the first INSERT. The second call positions the cursor on the row count of the second insert. The third call to *SQLMoreResults*() positions the cursor on the result set of the second SELECT statement.
- The row counts corresponding to the two INSERTs are rolled up into one single row count that is available. (A call to *SQLGetInfo*() returns the SQL_BRC_ROLLED_UP bit for the SQL_BATCH_ROW_COUNT option). The first call to *SQLMoreResults*() positions the cursor on the rolled-up row count, and the second call to *SQLMoreResults*() positions the cursor on the result set of the second SELECT.

Certain implementations make row counts available only for explicit batches and not for stored procedures.

SEE ALSO

For information about	See
Canceling statement processing	*SQLCancel*()
Fetching a block of data or scrolling through a result set	*SQLFetchScroll*()
Fetching a single row or a block of data in a forward-only direction	*SQLFetch*()
Fetching part or all of a column of data	*SQLGetData*()

CHANGE HISTORY

Version 2

Function added in this version.

15409 **NAME**

15410 SQLNativeSql — Return the text of a specified SQL statement as modified by the
15411 implementation, without executing the statement.

15412 **SYNOPSIS**

```
15413 SQLRETURN SQLNativeSql(
15414   SQLHDBC ConnectionHandle,
15415   SQLCHAR * InStatementText,
15416   SQLINTEGER TextLength1,
15417   SQLCHAR * OutStatementText,
15418   SQLINTEGER BufferLength,
15419   SQLINTEGER * TextLength2Ptr);
```

15420 **ARGUMENTS**

15421 *ConnectionHandle* [Input]
15422 Connection handle.

15423 *InStatementText* [Input]
15424 SQL text string to be translated.

15425 *TextLength1* [Input]
15426 Length of the text string in **InStatementText*.

15427 *OutStatementText* [Output]
15428 Pointer to a buffer in which to return the translated SQL string.

15429 *BufferLength* [Input]
15430 Length of the **OutStatementText* buffer.

15431 *TextLength2Ptr* [Output]
15432 Pointer to a buffer in which to return the total number of octets (excluding the null
15433 terminator) available to return in **OutStatementText*. If the number of octets available to
15434 return is greater than or equal to *BufferLength*, the translated SQL string in **OutStatementText*
15435 is truncated to *BufferLength* minus the length of a null terminator.

15436 **RETURN VALUE**

15437 SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.

15438 **DIAGNOSTICS**

15439 When *SQLNativeSql*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated
15440 SQLSTATE value can be obtained by calling *SQLGetDiagRec*() with a *HandleType* of
15441 SQL_HANDLE_DBC and a *Handle* of *ConnectionHandle*. The following table lists the SQLSTATE
15442 values commonly returned by *SQLNativeSql*(). The return code associated with each SQLSTATE
15443 value is SQL_ERROR, except that for SQLSTATE values in class 01, the return code is
15444 SQL_SUCCESS_WITH_INFO.

15445 01000 — General warning
15446 Implementation-defined informational message.

15447 01004 — String data, right truncation
15448 The buffer **OutStatementText* was not large enough to return the entire SQL string, so the
15449 SQL string was truncated. The length of the untruncated SQL string is returned in
15450 **TextLength2Ptr*.

15451 08003 — Connection does not exist
15452 *ConnectionHandle* was not in a connected state.

15453 08S01 — Communication link failure
15454 The communication link to the data source failed before the function completed processing.

15455 22007 — Invalid date/time format
15456 **InStatementText* contained an escape clause with an invalid date, time, or timestamp value.

15457 24000 — Invalid cursor state
15458 The cursor referred to in the statement was positioned before the start of the result set or
15459 after the end of the result set. Some implementations of *SQLNativeSQL*() do not determine
15460 the cursor position and might not report this error.

15461 40001 — Serialization failure
15462 The transaction in which the fetch was executed was terminated to prevent deadlock.

15463 HY000 — General error
15464 An error occurred for which there was no specific SQLSTATE and for which no
15465 implementation-specific SQLSTATE was defined. The error message returned by
15466 *SQLGetDiagRec*() in the **MessageText* buffer describes the error and its cause.

15467 HY001 — Memory allocation error
15468 The implementation failed to allocate memory required to support execution or completion
15469 of the function.

15470 HY009 — Invalid use of null pointer
15471 *InStatementText* was a null pointer.

15472 HY090 — Invalid string or buffer length
15473 *TextLength1* was less than 0, but not equal to SQL_NTS.

15474 *BufferLength* was less than 0 and *OutStatementText* was not a null pointer.

15475 HY109 — Invalid cursor position
15476 The current row of the cursor had been deleted or had not been fetched. Some
15477 implementations of *SQLNativeSQL*() do not determine the cursor position and might not
15478 report this error.

15479 HYT01 — Connection timeout expired
15480 The connection timeout period expired before the data source responded to the request. The
15481 connection timeout period is set through *SQLSetConnectAttr*(),
15482 SQL_ATTR_CONNECTION_TIMEOUT.

15483 IM001 — Function not supported
15484 The function is not supported on the current connection to the data source.

15485 **COMMENTS**

15486 Section 8.3 on page 84 defines XDBC escape sequences that portable applications can use in
15487 coding XSQL statements. The XDBC implementation converts these escape sequences into the
15488 SQL dialect that the target data source accepts.

15489 The application can call *SQLNativeSql*() to see the results of this conversion without executing
15490 the SQL statement. The string at **OutStatementText* is the SQL statement that the XDBC
15491 implementation would send to the data source if the string at **InStatementText* were submitted
15492 for execution, such as by calling *SQLExecDirect*().

15493 If **InStatementText* contains a distributed request, such as an SQL statement that joins tables
15494 from diverse data sources, the effect of calling *SQLNativeSql*() implementation-defined.

15495 The following examples show how *SQLNativeSql*() might translate uses of the **CONVERT**()
15496 scalar function (see Appendix F). Assume that the column empid is of type INTEGER in the
15497 data source, and that *InStatementText* contains:

```
15498 SELECT { fn CONVERT (empid, SQL_SMALLINT) } FROM employee
```

15499 For a Microsoft SQL Server data source, the implementation might translate the string to:

```
15500 SELECT convert (smallint, empid) FROM employee
```

15501 For an ORACLE data source, the translation might be:

```
15502 SELECT to_number (empid) FROM employee
```

15503 For an Ingres data source, the translation might be:

```
15504 SELECT int2 (empid) FROM employee
```

15505 **CHANGE HISTORY**

15506 **Version 2**

15507 Function added in this version.

15508 **NAME**

15509 SQLNumParams — Return the number of parameters in an SQL statement.

15510 **SYNOPSIS**

```
15511 SQLRETURN SQLNumParams(
15512   SQLHSTMT StatementHandle,
15513   SQLSMALLINT * ParameterCountPtr);
```

15514 **ARGUMENTS**

15515 *StatementHandle* [Input]

15516 Statement handle.

15517 *ParameterCountPtr* [Output]

15518 Pointer to a buffer in which to return the number of parameters in the statement.

15519 **RETURN VALUE**

15520 SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or
15521 SQL_INVALID_HANDLE.

15522 **DIAGNOSTICS**

15523 When *SQLNumParams*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated
15524 SQLSTATE value can be obtained by calling *SQLGetDiagRec*() with a *HandleType* of
15525 SQL_HANDLE_STMT and a *Handle* of *StatementHandle*. The following SQLSTATE values are
15526 commonly returned by *SQLNumParams*(). The return code associated with each SQLSTATE
15527 value is SQL_ERROR, except that for SQLSTATE values in class 01, the return code is
15528 SQL_SUCCESS_WITH_INFO.

15529 01000 — General warning

15530 Implementation-defined informational message.

15531 08S01 — Communication link failure

15532 The communication link to the data source failed before the function completed processing.

15533 40001 — Serialization failure

15534 The transaction in which the fetch was executed was terminated to prevent deadlock.

15535 HY000 — General error

15536 An error occurred for which there was no specific SQLSTATE and for which no
15537 implementation-specific SQLSTATE was defined. The error message returned by
15538 *SQLGetDiagRec*() in the **MessageText* buffer describes the error and its cause.

15539 HY001 — Memory allocation error

15540 The implementation failed to allocate memory required to support execution or completion
15541 of the function.

15542 HY008 — Operation canceled

15543 Asynchronous processing was enabled for *StatementHandle*. The function was called and
15544 before it completed execution, *SQLCancel*() was called on *StatementHandle*. The function
15545 was then called again on *StatementHandle*.

15546 The function was called and, before it completed execution, *SQLCancel*() was called on
15547 *StatementHandle* from a different thread in a multithread application.

15548 HY010 — Function sequence error

15549 The function was called prior to calling *SQLPrepare*() or *SQLExecDirect*() for
15550 *StatementHandle*.

15551 An asynchronously executing function (not this one) was called for *StatementHandle* and
15552 was still executing when this function was called.

15553 *SQLBulkOperations*(), *SQLExecDirect*(), *SQLExecute*(), or *SQLSetPos*() was called for
15554 *StatementHandle* and returned SQL_NEED_DATA. This function was called before data was
15555 sent for all data-at-execution parameters or columns.

15556 HYT01 — Connection timeout expired
15557 The connection timeout period expired before the data source responded to the request. The
15558 connection timeout period is set through *SQLSetConnectAttr*(),
15559 SQL_ATTR_CONNECTION_TIMEOUT.

15560 IM001 — Function not supported
15561 The function is not supported on the current connection to the data source.

15562 **COMMENTS**

15563 The number of parameters reported by *SQLNumParams*() is the same value as the
15564 SQL_DESC_COUNT field of the IPD, when population of IPDs is enabled.

15565 *SQLNumParams*() can be called only after *SQLPrepare*() has been called.

15566 If the statement associated with *StatementHandle* does not contain parameters, *SQLNumParams*()
15567 sets **ParameterCountPtr* to 0.

15568 **SEE ALSO**

15569	For information about	See
15570	Returning information about a parameter in a statement	*SQLDescribeParam*()
15571	Binding a buffer to a parameter	*SQLBindParameter*()

15572 **CHANGE HISTORY**

15573 **Version 2**

15574 Function added in this version.

15575 **NAME**

15576 SQLNumResultCols — Return the number of columns in a result set.

15577 **SYNOPSIS**

```
15578 SQLRETURN SQLNumResultCols(
15579   SQLHSTMT StatementHandle,
15580   SQLSMALLINT * ColumnCountPtr);
```

15581 **ARGUMENTS**

15582 *StatementHandle* [Input]

15583 Statement handle.

15584 *ColumnCountPtr* [Output]

15585 Pointer to a buffer in which to return the number of columns in the result set. This count
15586 does not include a bound bookmark column.

15587 **RETURN VALUE**

15588 SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or
15589 SQL_INVALID_HANDLE.

15590 **DIAGNOSTICS**

15591 When *SQLNumResultCols*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated
15592 SQLSTATE value can be obtained by calling *SQLGetDiagRec*() with a *HandleType* of
15593 SQL_HANDLE_STMT and a *Handle* of *StatementHandle*. The following table lists the SQLSTATE
15594 values commonly returned by *SQLNumResultCols*(). The return code associated with each
15595 SQLSTATEvalue is SQL_ERROR, except that for SQLSTATEvalues in class 01, the return code is
15596 SQL_SUCCESS_WITH_INFO.

15597 01000 — General warning

15598 Implementation-defined informational message.

15599 08S01 — Communication link failure

15600 The communication link to the data source failed before the function completed processing.

15601 40001 — Serialization failure

15602 The transaction in which the fetch was executed was terminated to prevent deadlock.

15603 HY000 — General error

15604 An error occurred for which there was no specific SQLSTATE and for which no
15605 implementation-specific SQLSTATE was defined. The error message returned by
15606 *SQLGetDiagRec*() in the **MessageText* buffer describes the error and its cause.

15607 HY001 — Memory allocation error

15608 The implementation failed to allocate memory required to support execution or completion
15609 of the function.

15610 HY008 — Operation canceled

15611 Asynchronous processing was enabled for *StatementHandle*. The function was called and
15612 before it completed execution, *SQLCancel*() was called on *StatementHandle*. The function
15613 was then called again on *StatementHandle*.

15614 The function was called and, before it completed execution, *SQLCancel*() was called on
15615 *StatementHandle* from a different thread in a multithread application.

15616 HY010 — Function sequence error

15617 The function was called prior to calling *SQLPrepare*() or *SQLExecDirect*() for
15618 *StatementHandle*.

15619 An asynchronously executing function (not this one) was called for *StatementHandle* and
15620 was still executing when this function was called.

15621 *SQLBulkOperations*(), *SQLExecDirect*(), *SQLExecute*(), or *SQLSetPos*() was called for
15622 *StatementHandle* and returned SQL_NEED_DATA. This function was called before data was
15623 sent for all data-at-execution parameters or columns.

15624 HYT01 — Connection timeout expired
15625 The connection timeout period expired before the data source responded to the request. The
15626 connection timeout period is set through *SQLSetConnectAttr*(),
15627 SQL_ATTR_CONNECTION_TIMEOUT.

15628 IM001 — Function not supported
15629 The function is not supported on the current connection to the data source.

15630 *SQLNumResultCols*() can return any SQLSTATE that can be returned by *SQLPrepare*() or
15631 *SQLExecute*() when called after *SQLPrepare*() and before *SQLExecute*() depending on when the
15632 data source evaluates the SQL statement associated with the statement.

15633 **COMMENTS**

15634 The number of result columns reported by *SQLNumResultCols*() is the same value as the
15635 SQL_DESC_COUNT field of the IRD.

15636 *SQLNumResultCols*() can be called successfully only when the statement is in the prepared,
15637 executed, or positioned state.

15638 If the statement associated with *StatementHandle* does not return a result set,
15639 *SQLNumResultCols*() sets **ColumnCountPtr* to 0.

15640 Calling *SQLNumResultCols*() between the preparation and execution of a statement can be
15641 costly; see **Performance Note** on page 279.

15642 **SEE ALSO**

15643	For information about	See
15644	Binding a buffer to a column in a result set	*SQLBindCol*()
15645	Canceling statement processing	*SQLCancel*()
15646	Returning information about a column in a result set	*SQLColAttribute*()
15647	Returning information about a column in a result set	*SQLDescribeCol*()
15648	Executing an SQL statement	*SQLExecDirect*()
15649	Executing a prepared SQL statement	*SQLExecute*()
15650	Fetching a block of data or scrolling through a result set	*SQLFetchScroll*()
15651 15652	Fetching a single row or a block of data in a forward-only direction	*SQLFetch*()
15653	Fetching part or all of a column of data	*SQLGetData*()
15654	Preparing an SQL statement for execution	*SQLPrepare*()

15655 **CHANGE HISTORY**

15656 **Version 2**

15657 Revised generally. See **Alignment with Popular Implementations** on page 2.

NAME

SQLParamData — Supply parameter data at statement execution time.

SYNOPSIS

```
SQLRETURN SQLParamData(
  SQLHSTMT StatementHandle,
  SQLPOINTER * ValuePtr);
```

ARGUMENTS

StatementHandle [Input]
Statement handle.

ValuePtr [Output]
Pointer to a buffer in which to return the address of the *ParameterValuePtr* buffer specified in *SQLBindParameter*() (for parameter data) or the address of the *TargetValuePtr* buffer specified in *SQLBindCol*() (for column data), as contained in the SQL_DESC_DATA_PTR descriptor record field.

RETURN VALUE

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NEED_DATA, SQL_NO_DATA, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE.

DIAGNOSTICS

When *SQLParamData*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling *SQLGetDiagRec*() with a *HandleType* of SQL_HANDLE_STMT and a *Handle* of *StatementHandle*. The following table lists the SQLSTATE values commonly returned by *SQLParamData*(). The return code associated with each SQLSTATE value is SQL_ERROR, except that for SQLSTATE values in class 01, the return code is SQL_SUCCESS_WITH_INFO.

01000 — General warning
Implementation-defined informational message.

07006 — Restricted data type attribute violation
The data value identified by *ValueType* in *SQLBindParameter*() for the bound parameter could not be converted to the data type identified by *ParameterType* in *SQLBindParameter*().

The data value returned for a parameter bound as SQL_PARAM_INPUT_OUTPUT or SQL_PARAM_OUTPUT could not be converted to the data type identified by *ValueType* in *SQLBindParameter*().

(If the data values for one or more rows could not be converted, but one or more rows were successfully returned, this function returns SQL_SUCCESS_WITH_INFO.)

08S01 — Communication link failure
The communication link to the data source failed before the function completed processing.

22026 — String data, length mismatch
The SQL_NEED_LONG_DATA_LEN option in *SQLGetInfo*() was ‘‘Y’’ and less data was sent for a long parameter (the data type was SQL_LONGVARCHAR, SQL_LONGVARBINARY, or a long, data source-specific data type) than was specified with *StrLen_or_IndPtr* in *SQLBindParameter*().

The SQL_NEED_LONG_DATA_LEN option in *SQLGetInfo*() was ‘‘Y’’ and less data was sent for a long column (the data type was SQL_LONGVARCHAR, SQL_LONGVARBINARY, or a long, data source-specific data type) than was specified in the length buffer corresponding to a column in a row of data that was added or updated with *SQLSetPos*() or *SQLBulkOperations*().

15704 40001 — Serialization failure
15705 The transaction in which the fetch was executed was terminated to prevent deadlock.

15706 HY000 — General error
15707 An error occurred for which there was no specific SQLSTATE and for which no
15708 implementation-specific SQLSTATE was defined. The error message returned by
15709 *SQLGetDiagRec*() in the **MessageText* buffer describes the error and its cause.

15710 HY001 — Memory allocation error
15711 The implementation failed to allocate memory required to support execution or completion
15712 of the function.

15713 HY008 — Operation canceled
15714 Asynchronous processing was enabled for *StatementHandle*. The function was called and
15715 before it completed execution, *SQLCancel*() was called on *StatementHandle*. The function
15716 was then called again on *StatementHandle*.

15717 The function was called and, before it completed execution, *SQLCancel*() was called on
15718 *StatementHandle* from a different thread in a multithread application.

15719 HY010 — Function sequence error
15720 The previous function call was not a call to *SQLBulkOperations*(), *SQLExecDirect*(),
15721 *SQLExecute*(), or *SQLSetPos*() where the return code was SQL_NEED_DATA, or the
15722 previous function call was a call to *SQLPutData*().

15723 The previous function call was a call to *SQLParamData*().

15724 An asynchronously executing function (not this one) was called for *StatementHandle* and
15725 was still executing when this function was called.

15726 *SQLExecute*(), *SQLExecDirect*(), or *SQLSetPos*() was called for *StatementHandle* and returned
15727 SQL_NEED_DATA. *SQLCancel*() was called before data was sent for all data-at-execution
15728 parameters or columns.

15729 HYT01 — Connection timeout expired
15730 The connection timeout period expired before the data source responded to the request. The
15731 connection timeout period is set through *SQLSetConnectAttr*(),
15732 SQL_ATTR_CONNECTION_TIMEOUT.

15733 IM001 — Function not supported
15734 The function is not supported on the current connection to the data source.

15735 If *SQLParamData*() is called while sending data for a parameter in an SQL statement, it can
15736 return any SQLSTATE that can be returned by the function called to execute the statement
15737 (*SQLExecute*() or *SQLExecDirect*()). If it is called while sending data for a column being updated
15738 or added with *SQLSetPos*() or *SQLBulkOperations*(), it can return any SQLSTATE that can be
15739 returned by that function.

15740 **COMMENTS**

15741 *SQLParamData*() can be called to supply data-at-execution data for two uses: parameter data to
15742 be used in a call to *SQLExecute*() or *SQLExecDirect*(), or column data to be used when a row is
15743 updated or added by a call to *SQLSetPos*() or *SQLBulkOperations*(). At execution time,
15744 *SQLParamData*() returns to the application an indicator of which data the implementation
15745 requires.

15746 *SQLBulkOperations*(), *SQLExecDirect*(), *SQLExecute*(), and *SQLSetPos*(), return
15747 SQL_NEED_DATA if they need data-at-execution data. An application then calls
15748 *SQLParamData*() to determine which data it should send. If the implementation requires
15749 parameter data, it returns in the **ValuePtr* output buffer the value the application placed in the
15750 row-set buffer. The application can use this value to determine which parameter data the

15751 implementation is requesting on this occasion. If the implementation requires column data, it
15752 returns in the **ValuePtr* buffer the address of the row where the data can be found. It also returns
15753 SQL_NEED_DATA, which is an indicator to the application that it should call *SQLPutData*() to
15754 send the data.

15755 When *SQLPutData*() returns SQL_SUCCESS, the application calls *SQLParamData*() again. If
15756 *SQLPutData*() returns SQL_NEED_DATA, then it requires data for another parameter or column,
15757 and the application again calls *SQLPutData*(). If *SQLParamData*() returns SQL_SUCCESS, then
15758 all data-at-execution data has been sent, and the SQL statement can be executed or the
15759 *SQLBulkOperations*() or *SQLSetPos*() call can be processed.

15760 If *SQLParamData*() supplies parameter data for a searched UPDATE or DELETE statement that
15761 does not affect any rows at the data source, the call to *SQLParamData*() returns SQL_NO_DATA.

15762 For more information on how data-at-execution parameter data is passed at statement execution
15763 time, see **Passing Parameter Values** on page 227. For more information on how data-at-
15764 execution column data is updated or added, see Section 12.3 on page 163 and
15765 *SQLBulkOperations*().

15766 **SEE ALSO**

15767	For information about	See
15768	Canceling statement processing	*SQLCancel*()
15769	Returning information about a parameter in a statement	*SQLDescribeParam*()
15770	Executing an SQL statement	*SQLExecDirect*()
15771	Executing a prepared SQL statement	*SQLExecute*()
15772	Sending parameter data at execution time	*SQLPutData*()
15773	Binding a buffer to a parameter	*SQLBindParameter*()

15774 **CHANGE HISTORY**

15775 **Version 2**

15776 Revised generally. See **Alignment with Popular Implementations** on page 2.

15777 **NAME**

15778 SQLPrepare — Prepare an SQL statement for execution.

15779 **SYNOPSIS**

```
15780 SQLRETURN SQLPrepare(
15781   SQLHSTMT StatementHandle,
15782   SQLCHAR * StatementText,
15783   SQLINTEGER TextLength);
```

15784 **ARGUMENTS**

15785 *StatementHandle* [Input]
15786 Statement handle.

15787 *StatementText* [Input]
15788 SQL text string.

15789 *TextLength* [Input]
15790 Length of **StatementText*.

15791 **RETURN VALUE**

15792 SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or
15793 SQL_INVALID_HANDLE.

15794 **DIAGNOSTICS**

15795 When *SQLPrepare*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated
15796 SQLSTATE value can be obtained by calling *SQLGetDiagRec*() with a *HandleType* of
15797 SQL_HANDLE_STMT and a *Handle* of *StatementHandle*. The following table lists the SQLSTATE
15798 values commonly returned by *SQLPrepare*(). The return code associated with each SQLSTATE
15799 value is SQL_ERROR, except that for SQLSTATE values in class 01, the return code is
15800 SQL_SUCCESS_WITH_INFO.

15801 01000 — General warning
15802 Implementation-defined informational message.

15803 01S02 — Attribute value changed
15804 A specified statement attribute was invalid and a similar value was temporarily substituted.
15805 See Section 9.2.1 on page 93.

15806 08S01 — Communication link failure
15807 The communication link to the data source failed before the function completed processing.

15808 22018 — Invalid character value
15809 **StatementText* contained an SQL statement that contained a literal or parameter and the
15810 value was incompatible with the data type of the associated table column.

15811 22019 — Invalid escape character
15812 **StatementText* contained a LIKE predicate with an ESCAPE in the WHERE clause, and the
15813 length of the escape character following ESCAPE was not equal to 1.

15814 22025 — Invalid escape sequence
15815 **StatementText* contained ''LIKE *pattern value* ESCAPE *escape character*'' in the WHERE
15816 clause, and the character following the escape character in the pattern value was not one of
15817 ''%'' or ''_''.

15818 24000 — Invalid cursor state
15819 A cursor was open on *StatementHandle*

15820 34000 — Invalid cursor name
15821 **StatementText* contained a positioned DELETE or a positioned UPDATE and the cursor
15822 referenced by the statement being prepared was not open.

15823 HY000 — General error
15824 An error occurred for which there was no specific SQLSTATE and for which no
15825 implementation-specific SQLSTATE was defined. The error message returned by
15826 *SQLGetDiagRec*() in the **MessageText* buffer describes the error and its cause.

15827 HY001 — Memory allocation error
15828 The implementation failed to allocate memory required to support execution or completion
15829 of the function.

15830 HY008 — Operation canceled
15831 Asynchronous processing was enabled for *StatementHandle*. The function was called and
15832 before it completed execution, *SQLCancel*() was called on *StatementHandle*. The function
15833 was then called again on *StatementHandle*.

15834 The function was called and, before it completed execution, *SQLCancel*() was called on
15835 *StatementHandle* from a different thread in a multithread application.

15836 HY009 — Invalid use of null pointer
15837 *StatementText* was a null pointer.

15838 HY010 — Function sequence error
15839 An asynchronously executing function (not this one) was called for *StatementHandle* and
15840 was still executing when this function was called.

15841 *SQLBulkOperations*(), *SQLExecDirect*(), *SQLExecute*(), or *SQLSetPos*() was called for
15842 *StatementHandle* and returned SQL_NEED_DATA. This function was called before data was
15843 sent for all data-at-execution parameters or columns.

15844 HY090 — Invalid string or buffer length
15845 *TextLength* was less than or equal to 0, but not equal to SQL_NTS.

15846 HYC00 — Optional feature not implemented
15847 The concurrency setting was invalid for the type of cursor defined.

15848 The SQL_ATTR_USE_BOOKMARKS statement attribute was set to SQL_UB_VARIABLE,
15849 and the SQL_ATTR_CURSOR_TYPE statement attribute was set to a cursor type for which
15850 the data source does not support bookmarks.

15851 HYT00 — Timeout expired
15852 The timeout period expired before the data source returned the result set. The timeout
15853 period is set through *SQLSetStmtAttr*(), SQL_ATTR_QUERY_TIMEOUT.

15854 HYT01 — Connection timeout expired
15855 The connection timeout period expired before the data source responded to the request. The
15856 connection timeout period is set through *SQLSetConnectAttr*(),
15857 SQL_ATTR_CONNECTION_TIMEOUT.

15858 IM001 — Function not supported
15859 The function is not supported on the current connection to the data source.

15860 In addition, the following diagnostics, defined in the X/Open **SQL** specification, can occur based
15861 on the SQL statement text:*

15862 **Cardinality violation**
15863 21S01 — Insert value does not match column list.
15864 21S02 — Degree of derived table does not match column list.

42000	**Syntax error or access violation**
42S01	— Base table or view already exists.
42S02	— Base table or view not found.
42S11	— Index already exists.
42S12	— Index not found.
42S21	— Column already exists.
42S22	— Column not found.

COMMENTS

The application calls *SQLPrepare*() to send an SQL statement to the data source for preparation. The application can include one or more parameter markers in the SQL statement. To include a parameter marker, the application embeds a question mark into the SQL string at the appropriate position.

Note: If an application uses *SQLPrepare*() to prepare and *SQLExecute*() to submit a COMMIT or ROLLBACK statement, it will not be interoperable between data sources. To commit or roll back a transaction, call *SQLEndTran*().

Once a statement is prepared, the application uses the statement handle to refer to the statement in later function calls. The prepared statement associated with the statement handle may be re-executed by calling *SQLExecute*() until the application frees the statement handle with a call to *SQLFreeHandle*() or until the statement handle is used in a call to *SQLPrepare*(), *SQLExecDirect*(), or a catalog function. Once the application prepares a statement, it can request information about the format of the result set. For some implementations, calling *SQLDescribeCol*() or *SQLDescribeParam*() after *SQLPrepare*() may not be as efficient as calling the function after *SQLExecute*() or *SQLExecDirect*().

Some implementations return syntax errors not when the statement is prepared but when it is executed. Some implementations do the same for access violations. Applications must be able to handle these conditions when calling subsequent related functions such as *SQLColAttribute*(), *SQLDescribeCol*(), *SQLExecute*(), and *SQLNumResultCols*().

Some implementations check parameter information (such as data types) when the statement is prepared (if all parameters have been bound), or when it is executed (if all parameters have not been bound). For maximum interoperability, an application should unbind all parameters that applied to an old SQL statement before preparing a new SQL statement on the same statement. This prevents errors that are due to old parameter information being applied to the new statement.

Transaction completion may have side-effects on cursors and on access plans of prepared statements. See Section 14.1.3 on page 184.

SEE ALSO

For information about	See
Allocating a statement handle	*SQLAllocHandle*()
Binding a buffer to a column in a result set	*SQLBindCol*()
Canceling statement processing	*SQLCancel*()
Executing a commit or rollback operation	*SQLEndTran*()
Executing an SQL statement	*SQLExecDirect*()
Executing a prepared SQL statement	*SQLExecute*()

15908 Returning the number of rows affected by a statement *SQLRowCount*()
15909 Setting a cursor name *SQLSetCursorName*()
15910 Binding a buffer to a parameter *SQLBindParameter*()

15911 **CHANGE HISTORY**

15912 **Version 2**

15913 Revised generally. See **Alignment with Popular Implementations** on page 2.

15914 **NAME**

15915 SQLPrimaryKeys — Return as a result set the column names of the primary key of a table.

15916 **SYNOPSIS**

```
15917 SQLRETURN SQLPrimaryKeys(
15918   SQLHSTMT StatementHandle,
15919   SQLCHAR * CatalogName,
15920   SQLSMALLINT NameLength1,
15921   SQLCHAR * SchemaName,
15922   SQLSMALLINT NameLength2,
15923   SQLCHAR * TableName,
15924   SQLSMALLINT NameLength3);
```

15925 **ARGUMENTS**

15926 *StatementHandle* [Input]
15927 Statement handle.

15928 *CatalogName* [Input]
15929 Catalog name. If a data source supports catalogs, an empty string denotes those tables that
15930 do not have catalogs.

15931 If the SQL_ATTR_METADATA_ID statement attribute is SQL_TRUE, this argument is
15932 interpreted as specified in **Identifier (ID) Arguments** on page 72. If it is SQL_FALSE, this
15933 argument is interpreted as specified in **Ordinary Arguments (OA)** on page 71.

15934 *NameLength1* [Input]
15935 Length in octets of **CatalogName*.

15936 *SchemaName* [Input]
15937 Schema name. If a data source supports schemas, an empty string denotes those tables that
15938 do not have schemas.

15939 If the SQL_ATTR_METADATA_ID statement attribute is SQL_TRUE, this argument is
15940 interpreted as specified in **Identifier (ID) Arguments** on page 72. If it is SQL_FALSE, this
15941 argument is interpreted as specified in **Ordinary Arguments (OA)** on page 71.

15942 *NameLength2* [Input]
15943 Length in octets of **SchemaName*.

15944 *TableName* [Input]
15945 Table name. This argument cannot be a null pointer.

15946 If the SQL_ATTR_METADATA_ID statement attribute is SQL_TRUE, this argument is
15947 interpreted as specified in **Identifier (ID) Arguments** on page 72. If it is SQL_FALSE, this
15948 argument is interpreted as specified in **Ordinary Arguments (OA)** on page 71.

15949 *NameLength3* [Input]
15950 Length in octets of **TableName*.

15951 **RETURN VALUE**

15952 SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or
15953 SQL_INVALID_HANDLE.

15954 **DIAGNOSTICS**

15955 When *SQLPrimaryKeys*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated
15956 SQLSTATE value may be obtained by calling *SQLGetDiagRec*() with a *HandleType* of
15957 SQL_HANDLE_STMT and a *Handle* of *StatementHandle*. The following table lists the SQLSTATE
15958 values commonly returned by *SQLPrimaryKeys*(). The return code associated with each
15959 SQLSTATEvalue is SQL_ERROR, except that for SQLSTATEvalues in class 01, the return code is

15960 SQL_SUCCESS_WITH_INFO.

15961 01000 — General warning
15962 Implementation-defined informational message.

15963 08S01 — Communication link failure
15964 The communication link to the data source failed before the function completed processing.

15965 24000 — Invalid cursor state
15966 A cursor was open on *StatementHandle*.

15967 HY000 — General error
15968 An error occurred for which there was no specific SQLSTATE and for which no
15969 implementation-specific SQLSTATE was defined. The error message returned by
15970 *SQLGetDiagRec*() in the **MessageText* buffer describes the error and its cause.

15971 HY001 — Memory allocation error
15972 The implementation failed to allocate memory required to support execution or completion
15973 of the function.

15974 HY008 — Operation canceled
15975 Asynchronous processing was enabled for *StatementHandle*. The function was called and
15976 before it completed execution, *SQLCancel*() was called on *StatementHandle*. The function
15977 was then called again on *StatementHandle*.

15978 The function was called and, before it completed execution, *SQLCancel*() was called on
15979 *StatementHandle* from a different thread in a multithread application.

15980 HY009 — Invalid use of null pointer
15981 *TableName* was a null pointer.

15982 The SQL_ATTR_METADATA_ID statement attribute was set to SQL_TRUE, *CatalogName*
15983 was a null pointer, and the SQL_CATALOG_NAME option of *SQLGetInfo*() returns that
15984 catalog names are supported.

15985 The SQL_ATTR_METADATA_ID statement attribute was set to SQL_TRUE, and
15986 *SchemaName* or *TableName* was a null pointer.

15987 HY010 — Function sequence error
15988 An asynchronously executing function (not this one) was called for *StatementHandle* and
15989 was still executing when this function was called.

15990 *SQLBulkOperations*(), *SQLExecDirect*(), *SQLExecute*(), or *SQLSetPos*() was called for
15991 *StatementHandle* and returned SQL_NEED_DATA. This function was called before data was
15992 sent for all data-at-execution parameters or columns.

15993 HY090 — Invalid string or buffer length
15994 The value of one of the name length arguments was less than 0, but not equal to SQL_NTS.

15995 The value of one of the name length arguments exceeded the maximum length value for the
15996 corresponding name.

15997 HYC00 — Optional feature not implemented
15998 A catalog was specified and the implementation does not support catalog.

15999 A schema was specified and the implementation does not support schemas.

16000 The data source does not support the combination of the current settings of the
16001 SQL_ATTR_CONCURRENCY and SQL_ATTR_CURSOR_TYPE statement attributes.

16002 The SQL_ATTR_USE_BOOKMARKS statement attribute was set to SQL_UB_VARIABLE,
16003 and the SQL_ATTR_CURSOR_TYPE statement attribute was set to a cursor type for which

16004 the data source does not support bookmarks.

16005 HYT00 — Timeout expired
16006 The timeout period expired before the data source returned the requested result set. The
16007 timeout period is set through *SQLSetStmtAttr*(), SQL_ATTR_QUERY_TIMEOUT.

16008 HYT01 — Connection timeout expired
16009 The connection timeout period expired before the data source responded to the request. The
16010 connection timeout period is set through *SQLSetConnectAttr*(),
16011 SQL_ATTR_CONNECTION_TIMEOUT.

16012 IM001 — Function not supported
16013 The function is not supported on the current connection to the data source.

16014 **COMMENTS**

16015 *SQLPrimaryKeys*() returns the columns that are the primary keys of *TableName* as a result set.
16016 The result set is ordered by TABLE_CAT,TABLE_SCHEM, TABLE_NAME, and KEY_SEQ. •

16017 To determine the actual lengths of the TABLE_CAT, TABLE_SCHEM, TABLE_NAME, and
16018 COLUMN_NAME columns, call *SQLGetInfo*() with the SQL_MAX_CATALOG_NAME_LEN,
16019 SQL_MAX_SCHEMA_NAME_LEN, SQL_MAX_TABLE_NAME_LEN, and
16020 SQL_MAX_COLUMN_NAME_LEN options.

16021 This function does not support returning primary keys from multiple tables in a single call.

16022 The following table lists the columns in the result set. Additional columns beyond column 6
16023 (PK_NAME) can be defined by the implementation. An application should gain access to |
16024 implementation-defined columns by counting down from the end of the result set rather than by |
16025 specifying an explicit ordinal position; see Section 7.3 on page 68.

16026–16039

Column name	Col. No.	Data type	Comments
TABLE_CAT	1	Varchar	Primary key table catalog identifier; NULL if not applicable to the data source. If a data source supports catalogs, it returns an empty string for those tables that do not have catalogs.
TABLE_SCHEM	2	Varchar	Primary key table schema identifier; NULL if not applicable to the data source. If a data source supports schemas, it returns an empty string for those tables that do not have schemas.
TABLE_NAME	3	Varchar not NULL	Primary key table identifier.

COLUMN_NAME	4	Varchar not NULL	Primary key column identifier; an empty string if the column is unnamed.
KEY_SEQ	5	Smallint not NULL	Column sequence number in key (starting with 1).
PK_NAME	6	Varchar	Primary key identifier. NULL if not applicable to the data source.

SEE ALSO

For information about	See
Overview of catalog functions	Chapter 7
Binding a buffer to a column in a result set	*SQLBindCol*()
Canceling statement processing	*SQLCancel*()
Fetching a block of data or scrolling through a result set	*SQLFetchScroll*()
Fetching a single row or a block of data in a forward-only direction	*SQLFetch*()
Returning the columns of foreign keys	*SQLForeignKeys*()
Returning table statistics and indexes	*SQLStatistics*()

CHANGE HISTORY

Version 2

Function added in this version.

NAME

SQLProcedureColumns — Return as a result set the list of input and output parameters, and the columns of the result set, for the specified procedures.

SYNOPSIS

```
SQLRETURN SQLProcedureColumns(
  SQLHSTMT StatementHandle,
  SQLCHAR * CatalogName,
  SQLSMALLINT NameLength1,
  SQLCHAR * SchemaName,
  SQLSMALLINT NameLength2,
  SQLCHAR * ProcName,
  SQLSMALLINT NameLength3,
  SQLCHAR * ColumnName,
  SQLSMALLINT NameLength4);
```

ARGUMENTS

StatementHandle [Input]
Statement handle.

CatalogName [Input]
Procedure catalog name. If a data source supports catalogs, an empty string denotes those procedures that do not have catalogs.

If the SQL_ATTR_METADATA_ID statement attribute is SQL_TRUE, this argument is interpreted as specified in **Identifier (ID) Arguments** on page 72. If it is SQL_FALSE, this argument is interpreted as specified in **Ordinary Arguments (OA)** on page 71.

NameLength1 [Input]
Length of **CatalogName*.

SchemaName [Input]
String search pattern for procedure schema names. If a data source supports schemas, an empty string denotes those procedures that do not have schemas.

If the SQL_ATTR_METADATA_ID statement attribute is SQL_TRUE, this argument is interpreted as specified in **Identifier (ID) Arguments** on page 72. If it is SQL_FALSE, this argument is interpreted as specified in **Pattern Value (PV) Arguments** on page 71 and the application may use a search pattern.

NameLength2 [Input]
Length of **SchemaName*.

ProcName [Input]
String search pattern for procedure names.

If the SQL_ATTR_METADATA_ID statement attribute is SQL_TRUE, this argument is interpreted as specified in **Identifier (ID) Arguments** on page 72. If it is SQL_FALSE, this argument is interpreted as specified in **Pattern Value (PV) Arguments** on page 71 and the application may use a search pattern.

NameLength3 [Input]
Length of **ProcName*.

ColumnName [Input]
String search pattern for column names.

If the SQL_ATTR_METADATA_ID statement attribute is SQL_TRUE, this argument is interpreted as specified in **Identifier (ID) Arguments** on page 72. If it is SQL_FALSE, this

16105 argument is interpreted as specified in **Pattern Value (PV) Arguments** on page 71 and the
16106 application may use a search pattern.

16107 *NameLength4* [Input]
16108 Length of **ColumnName*.

16109 **RETURN VALUE**

16110 SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or
16111 SQL_INVALID_HANDLE.

16112 When *SQLProcedureColumns*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an
16113 associated SQLSTATE value can be obtained by calling *SQLGetDiagRec*() with a *HandleType* of
16114 SQL_HANDLE_STMT and a *Handle* of *StatementHandle*. The following SQLSTATE values are
16115 commonly returned by *SQLProcedureColumns*(). The return code associated with each
16116 SQLSTATEvalue is SQL_ERROR, except that for SQLSTATEvalues in class 01, the return code is
16117 SQL_SUCCESS_WITH_INFO.

16118 01000 — General warning
16119 Implementation-defined informational message.

16120 08S01 — Communication link failure
16121 The communication link to the data source failed before the function completed processing.

16122 24000 — Invalid cursor state
16123 A cursor was open on *StatementHandle*.

16124 HY000 — General error
16125 An error occurred for which there was no specific SQLSTATE and for which no
16126 implementation-specific SQLSTATE was defined. The error message returned by
16127 *SQLGetDiagRec*() in the **MessageText* buffer describes the error and its cause.

16128 HY001 — Memory allocation error
16129 The implementation failed to allocate memory required to support execution or completion
16130 of the function.

16131 HY008 — Operation canceled
16132 Asynchronous processing was enabled for *StatementHandle*. The function was called and
16133 before it completed execution, *SQLCancel*() was called on *StatementHandle*. The function
16134 was then called again on *StatementHandle*.

16135 The function was called and, before it completed execution, *SQLCancel*() was called on
16136 *StatementHandle* from a different thread in a multithread application.

16137 HY009 — Invalid use of null pointer
16138 The SQL_ATTR_METADATA_IDstatement attribute was set to SQL_TRUE, *CatalogName*
16139 was a null pointer, and the SQL_CATALOG_NAME option of *SQLGetInfo*() returns that
16140 catalog names are supported.

16141 The SQL_ATTR_METADATA_ID statement attribute was set to SQL_TRUE, and
16142 *SchemaName*, *ProcName*, or *ColumnName* was a null pointer.

16143 HY010 — Function sequence error
16144 An asynchronously executing function (not this one) was called for *StatementHandle* and
16145 was still executing when this function was called.

16146 *SQLBulkOperations*(), *SQLExecDirect*(), *SQLExecute*(), or *SQLSetPos*() was called for
16147 *StatementHandle* and returned SQL_NEED_DATA. This function was called before data was
16148 sent for all data-at-execution parameters or columns.

16149 HY090 — Invalid string or buffer length
16150 The value of one of the name length arguments was less than 0, but not equal to SQL_NTS.

16151 The value of one of the name length arguments exceeded the maximum length value for the
16152 corresponding catalog, schema, procedure, or column name.

16153 HYC00 — Optional feature not implemented
16154 A catalog was specified and the implementation does not support catalogs.

16155 A schema was specified and the implementation does not support schemas.

16156 A string search pattern was specified for the procedure schema, procedure name, or column
16157 name and the data source does not support search patterns for one or more of those
16158 arguments.

16159 The data source does not support the combination of the current settings of the
16160 SQL_ATTR_CONCURRENCY and SQL_ATTR_CURSOR_TYPE statement attributes.

16161 The SQL_ATTR_USE_BOOKMARKS statement attribute was set to SQL_UB_VARIABLE,
16162 and the SQL_ATTR_CURSOR_TYPE statement attribute was set to a cursor type for which
16163 the data source does not support bookmarks.

16164 HYT00 — Timeout expired
16165 The timeout period expired before the data source returned the result set. The timeout
16166 period is set through *SQLSetStmtAttr*(), SQL_ATTR_QUERY_TIMEOUT.

16167 HYT01 — Connection timeout expired
16168 The connection timeout period expired before the data source responded to the request. The
16169 connection timeout period is set through *SQLSetConnectAttr*(),
16170 SQL_ATTR_CONNECTION_TIMEOUT.

16171 IM001 — Function not supported
16172 The function is not supported on the current connection to the data source.

16173 **COMMENTS**

16174 This function is typically used before executing a statement that invokes a procedure, to retrieve
16175 information about any parameters of the procedure and about any columns in a result set the
16176 procedure may return.

16177 **Note:** *SQLProcedureColumns*() might not return all columns used by a procedure. For example, a
16178 data source might only return information about the parameters used by a procedure and not
16179 the columns in a result set it generates.

16180 *SQLProcedureColumns*() returns the results as a standard result set, ordered by
16181 PROCEDURE_CAT, PROCEDURE_SCHEM, PROCEDURE_NAME, and ORDINAL_POSITION.
16182 Column names are returned for each procedure in the following order: the name of the return
16183 value, the names of each parameter in the procedure invocation (in call order), and then the
16184 names of each column in the result set returned by the procedure (in column order).

16185 Applications should bind implementation-defined columns relative to the end of the result set.
16186 For more information, see Section 7.3 on page 68.

16187 To determine the actual lengths of the PROCEDURE_CAT, PROCEDURE_SCHEM,
16188 PROCEDURE_NAME, and COLUMN_NAME columns, an application can call *SQLGetInfo*()
16189 with the SQL_MAX_CATALOG_NAME_LEN, SQL_MAX_SCHEMA_NAME_LEN,
16190 SQL_MAX_PROCEDURE_NAME_LEN, and SQL_MAX_COLUMN_NAME_LEN options.

16191 The following table lists the columns in the result set. Additional columns beyond column 19
16192 (IS_NULLABLE) can be defined by the implementation. An application should gain access to
16193 implementation-defined columns by counting down from the end of the result set rather than by
16194 specifying an explicit ordinal position; see Section 7.3 on page 68.

Column Name	Col. No.	Data Type	Comments
PROCEDURE_CAT	1	Varchar	Procedure catalog identifier; NULL if not applicable to the data source. If a data source supports catalogs, it returns an empty string for those procedures that do not have catalogs.
PROCEDURE_SCHEM	2	Varchar	Procedure schema identifier; NULL if not applicable to the data source. If a data source supports schemas, it returns an empty string for those procedures that do not have schemas.
PROCEDURE_NAME	3	Varchar not NULL	Procedure identifier. An empty string is returned for a procedure that does not have an identifier.
COLUMN_NAME	4	Varchar not NULL	Procedure column identifier. An empty string is returned for a procedure column that does not have an identifier.
COLUMN_TYPE	5	Smallint not NULL	Defines the procedure column as parameter or a result set column: SQL_PARAM_TYPE_UNKNOWN: The procedure column is a parameter whose type is unknown. SQL_PARAM_INPUT: The procedure column is an input parameter. SQL_PARAM_INPUT_OUTPUT: the procedure column is an input/output parameter. SQL_PARAM_OUTPUT: The procedure column is an output parameter. SQL_RETURN_VALUE: The procedure column is the return value of the procedure. SQL_RESULT_COL: The procedure column is a result set column.

DATA_TYPE	6	Smallint not NULL	SQL data type. This can be an XDBC SQL data type or an implementation-defined SQL data type. For date/time and interval data types, this column returns the concise data types (for example, SQL_TYPE_TIME or SQL_INTERVAL_YEAR_TO_MONTH). For a list of valid XDBC SQL data types, see Section D.1 on page 556.
TYPE_NAME	7	Varchar not NULL	Data source-dependent data type name; for example, ''CHAR'', ''VARCHAR'', ''MONEY'', ''LONG VARBINARY'', or ''CHAR ( ) FOR BIT DATA''.
COLUMN_SIZE	8	Integer	The column size of the procedure column on the data source. NULL is returned for data types where column size is not applicable. For more information concerning precision, see Section D.3 on page 562.
BUFFER_LENGTH	9	Integer	The length in octets of data transferred on an *SQLGetData*() or *SQLFetch*() operation if SQL_C_DEFAULT is specified. For numeric data, this size may be different from the size of the data stored on the data source. For more information, see Section D.3 on page 562.
DECIMAL_DIGITS	10	Smallint	The decimal digits of the procedure column on the data source. NULL is returned for data types where decimal digits is not applicable. For more information concerning decimal digits, see Section D.3 on page 562.
NUM_PREC_RADIX	11	Smallint	For numeric data types, either 10 or 2. If it is 10, the values in COLUMN_SIZE and DECIMAL_DIGITS give the number of decimal digits allowed for the column. For example, a DECIMAL(12,5) column would return a NUM_PREC_RADIX of 10, a COLUMN_SIZE of 12, and a DECIMAL_DIGITS of 5; a FLOAT column could return a NUM_PREC_RADIX of 10, a COLUMN_SIZE of 15 and a DECIMAL_DIGITS of NULL. If it is 2, the values in COLUMN_SIZE and DECIMAL_DIGITS give the number of bits allowed in the column. For example, a FLOAT column could return a NUM_PREC_RADIX of 2, a COLUMN_SIZE of 53, and a DECIMAL_DIGITS of NULL. NULL is returned for data types where

			NUM_PREC_RADIX is not applicable.
NULLABLE	12	Smallint not NULL	SQL_NO_NULLS if the procedure column does not accept NULL values. SQL_NULLABLE if the procedure column accepts NULL values. SQL_NULLABLE_UNKNOWN if it is not known if the procedure column accepts NULL values.
REMARKS	13	Varchar	A description of the procedure column.
COLUMN_DEF	14	Varchar	The default value of the column. See Section 7.3.1 on page 68.
SQL_DATA_TYPE	15	Smallint not NULL	The value of the SQL data type as it appears in the SQL_DESC_TYPE field of the descriptor. This column is the same as the DATA_TYPE column, except for date/time and interval data types.
			For date/time and interval data types, the SQL_DATA_TYPE field in the result set returns SQL_INTERVAL or SQL_DATETIME, and the SQL_DATETIME_SUB field returns the subcode for the specific interval or date/time data type (see Appendix D.)
SQL_DATETIME_SUB	16	Smallint	The subtype code for date/time and interval data types. For other data types, this column returns a NULL.
CHAR_OCTET_LENGTH	17	Integer	The maximum length in octets of a character data type column. For all other data types, this column returns a NULL.
ORDINAL_POSITION	18	Integer not NULL	For input parameters, the ordinal position of the parameter in the procedure definition (from left to right). The first parameter is number 1. For output parameters, this column is 0.
			For result-set columns, the ordinal position of the column in the table. The first column in the table is number 1. If there are multiple result sets, column ordinal positions are implementation-defined.

16315 IS_NULLABLE 19 Varchar ''NO'' if the column does not include NULLs. ''YES'' if the column can include NULLs. A zero-length string if nullability is unknown. ISO rules are followed to determine nullability. An ISO SQL compliant data source cannot return an empty string.

The value returned for this column is different from the value returned for the NULLABLE column. (See the description of the NULLABLE column.)

SEE ALSO

For information about	See
Overview of catalog functions	Chapter 7
Overview of procedures	Section 9.3.3 on page 97
Standard syntax (XDBC escape clause) for calling a procedure	Section 8.3.6 on page 88
Binding a buffer to a column in a result set	*SQLBindCol*()
Canceling statement processing	*SQLCancel*()
Fetching a block of data or scrolling through a result set	*SQLFetchScroll*()
Fetching a single row or a block of data in a forward-only direction	*SQLFetch*()
Returning a list of procedures in a data source	*SQLProcedures*()

CHANGE HISTORY

Version 2

Function added in this version.

16341 **NAME**

16342 SQLProcedures — Return the list of procedure names stored in a specified data source.

16343 **SYNOPSIS**

```
16344 SQLRETURN SQLProcedures(
16345   SQLHSTMT StatementHandle,
16346   SQLCHAR * CatalogName,
16347   SQLSMALLINT NameLength1,
16348   SQLCHAR * SchemaName,
16349   SQLSMALLINT NameLength2,
16350   SQLCHAR * ProcName,
16351   SQLSMALLINT NameLength3);
```

16352 **ARGUMENTS**

16353 *StatementHandle* [Input] |

16354 Statement handle. |

16355 *CatalogName* [Input]

16356 Procedure catalog. If a data source supports catalogs, an empty string denotes those tables
16357 that do not have catalogs.

16358 If the SQL_ATTR_METADATA_ID statement attribute is SQL_TRUE, this argument is |
16359 interpreted as specified in **Identifier (ID) Arguments** on page 72. If it is SQL_FALSE, this |
16360 argument is interpreted as specified in **Ordinary Arguments (OA)** on page 71. |

16361 *NameLength1* [Input] |

16362 Length in octets of **CatalogName*. |

16363 *SchemaName* [Input] |

16364 String search pattern for procedure schema names. If a data source supports schemas, an |
16365 empty string denotes those procedures that do not have schemas. |

16366 If the SQL_ATTR_METADATA_ID statement attribute is SQL_TRUE, this argument is |
16367 interpreted as specified in **Identifier (ID) Arguments** on page 72. If it is SQL_FALSE, this |
16368 argument is interpreted as specified in **Pattern Value (PV) Arguments** on page 71 and the |
16369 application may use a search pattern. |

16370 *NameLength2* [Input]

16371 Length in octets of **SchemaName*. |

16372 *ProcName* [Input]

16373 String search pattern for procedure names. |

16374 If the SQL_ATTR_METADATA_ID statement attribute is SQL_TRUE, this argument is |
16375 interpreted as specified in **Identifier (ID) Arguments** on page 72. If it is SQL_FALSE, this |
16376 argument is interpreted as specified in **Pattern Value (PV) Arguments** on page 71 and the |
16377 application may use a search pattern. |

16378 *NameLength3* [Input] |

16379 Length in octets of **ProcName*. |

16380 **RETURN VALUE** |

16381 SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or |
16382 SQL_INVALID_HANDLE. |

16383 **DIAGNOSTICS** |

16384 When *SQLProcedures*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated |
16385 SQLSTATE value can be obtained by calling *SQLGetDiagRec*() with a *HandleType* of
16386 SQL_HANDLE_STMT and a *Handle* of *StatementHandle*. The following table lists the SQLSTATE

16387 values commonly returned by *SQLProcedures*(). The return code associated with each
16388 SQLSTATEvalue is SQL_ERROR, except that for SQLSTATEvalues in class 01, the return code is
16389 SQL_SUCCESS_WITH_INFO.

16390 01000 — General warning
16391 Implementation-defined informational message.

16392 08S01 — Communication link failure
16393 The communication link to the data source failed before the function completed processing.

16394 24000 — Invalid cursor state
16395 A cursor was open on *StatementHandle*.

16396 HY000 — General error
16397 An error occurred for which there was no specific SQLSTATE and for which no
16398 implementation-specific SQLSTATE was defined. The error message returned by
16399 *SQLGetDiagRec*() in the **MessageText* buffer describes the error and its cause.

16400 HY001 — Memory allocation error
16401 The implementation failed to allocate memory required to support execution or completion
16402 of the function.

16403 HY008 — Operation canceled
16404 Asynchronous processing was enabled for *StatementHandle*. The function was called and
16405 before it completed execution, *SQLCancel*() was called on *StatementHandle*. The function
16406 was then called again on *StatementHandle*.

16407 The function was called and, before it completed execution, *SQLCancel*() was called on
16408 *StatementHandle* from a different thread in a multithread application.

16409 HY009 — Invalid use of null pointer
16410 The SQL_ATTR_METADATA_IDstatement attribute was set to SQL_TRUE, *CatalogName*
16411 was a null pointer, and the SQL_CATALOG_NAME option of *SQLGetInfo*() returns that
16412 catalog names are supported.

16413 The SQL_ATTR_METADATA_ID statement attribute was set to SQL_TRUE, and
16414 *SchemaName* or *ProcName* was a null pointer.

16415 HY010 — Function sequence error
16416 An asynchronously executing function (not this one) was called for *StatementHandle* and
16417 was still executing when this function was called.

16418 *SQLBulkOperations*(), *SQLExecDirect*(), *SQLExecute*(), or *SQLSetPos*() was called for
16419 *StatementHandle* and returned SQL_NEED_DATA. This function was called before data was
16420 sent for all data-at-execution parameters or columns.

16421 HY090 — Invalid string or buffer length
16422 The value of one of the name length arguments was less than 0, but not equal to SQL_NTS.

16423 The value of one of the name length arguments exceeded the maximum length value for the
16424 corresponding name.

16425 HYC00 — Optional feature not implemented
16426 A catalog was specified and the implementation does not support catalogs.

16427 A schema was specified and the implementation does not support schemas.

16428 A string search pattern was specified for the procedure schema or procedure name and the
16429 data source does not support search patterns for one or more of those arguments.

16430 The data source does not support the combination of the current settings of the
16431 SQL_ATTR_CONCURRENCY and SQL_ATTR_CURSOR_TYPE statement attributes.

16432 The SQL_ATTR_USE_BOOKMARKS statement attribute was set to SQL_UB_VARIABLE,
16433 and the SQL_ATTR_CURSOR_TYPE statement attribute was set to a cursor type for which
16434 the data source does not support bookmarks.

16435 HYT00 — Timeout expired
16436 The query timeout period expired before the data source returned the requested result set.
16437 The timeout period is set through *SQLSetStmtAttr*(), SQL_ATTR_QUERY_TIMEOUT.

16438 HYT01 — Connection timeout expired
16439 The connection timeout period expired before the data source responded to the request. The
16440 connection timeout period is set through *SQLSetConnectAttr*(),
16441 SQL_ATTR_CONNECTION_TIMEOUT.

16442 IM001 — Function not supported
16443 The function is not supported on the current connection to the data source.

16444 **COMMENTS**

16445 *SQLProcedures*() lists all procedures in the requested range. A user may or may not have
16446 permission to execute any of these procedures. To check accessibility, an application can call
16447 *SQLGetInfo*() and check the SQL_ACCESSIBLE_PROCEDURES information value. Otherwise,
16448 the application must be able to handle a situation where the user selects a procedure which it
16449 cannot execute.

16450 *SQLProcedures*() returns the results as a standard result set, ordered by PROCEDURE_CAT,
16451 PROCEDURE_SCHEMA, and PROCEDURE_NAME. •

16452 To determine the actual lengths of the PROCEDURE_CAT, PROCEDURE_SCHEM, and
16453 PROCEDURE_NAME columns, an application can call *SQLGetInfo*() with the
16454 SQL_MAX_CATALOG_NAME_LEN, SQL_MAX_SCHEMA_NAME_LEN, and
16455 SQL_MAX_PROCEDURE_NAME_LEN options.

16456 The following table lists the columns in the result set. Additional columns beyond column 8
16457 (PROCEDURE_TYPE) can be defined by the implementation. An application should gain access
16458 to implementation-defined columns by counting down from the end of the result set rather than
16459 by specifying an explicit ordinal position; see Section 7.3 on page 68.

16460
16461

Column Name	Col. No.	Data Type	Comments
PROCEDURE_CAT	1	Varchar	Procedure catalog identifier; NULL if not applicable to the data source. If a data source supports catalogs, it returns an empty string for those procedures that do not have catalogs.
PROCEDURE_SCHEM	2	Varchar	Procedure schema identifier; NULL if not applicable to the data source. If a data source supports schemas, it returns an empty string for those procedures that do not have schemas.

16462
16463
16464
16465
16466
16467
16468
16469
16470
16471

PROCEDURE_NAME	3	Varchar not NULL	Procedure identifier.
NUM_INPUT_PARAMS	4	N/A	Reserved for future use.
NUM_OUTPUT_PARAMS	5	N/A	Reserved for future use.
NUM_RESULT_SETS	6	N/A	Reserved for future use.
REMARKS	7	Varchar	A description of the procedure.
PROCEDURE_TYPE	8	Smallint	SQL_PT_FUNCTION if the returned object is a function; that is, it has a return value. SQL_PT_PROCEDURE if the returned object is a procedure; that is, it does not have a return value. SQL_PT_UNKNOWN if it cannot be determined whether the procedure returns a value.

Applications should not rely on data returned in columns described as ''Reserved for future use.''

SEE ALSO

For information about	See
Overview of catalog functions	Chapter 7
Overview of procedures	Section 9.3.3 on page 97
Standard syntax (XDBC escape clause) for calling a procedure	Section 8.3.6 on page 88
Binding a buffer to a column in a result set	*SQLBindCol*()
Canceling statement processing	*SQLCancel*()
Fetching a block of data or scrolling through a result set	*SQLFetchScroll*()
Fetching a single row or a block of data in a forward-only direction	*SQLFetch*()
Returning information about an implementation	*SQLGetInfo*()
Returning the parameters and result set columns of a procedure	*SQLProcedureColumns*()

CHANGE HISTORY

Version 2

Function added in this version.

16504 **NAME**

16505 SQLPutData — Supply data for a parameter or column at statement execution time.

16506 **SYNOPSIS**

```
16507 SQLRETURN SQLPutData(
16508   SQLHSTMT StatementHandle,
16509   SQLPOINTER DataPtr,
16510   SQLINTEGER StrLen_or_Ind);
```

16511 **ARGUMENTS**

16512 *StatementHandle* [Input]
16513 Statement handle.

16514 *DataPtr* [Input]
16515 Pointer to a buffer containing the actual data for the parameter or column. The data must
16516 be in the C data type specified in *ValueType* of *SQLBindParameter*() (for parameter data) or
16517 *TargetType* of *SQLBindCol*() (for column data).

16518 *StrLen_or_Ind* [Input]
16519 Length of **DataPtr*. Specifies the amount of data sent in a call to *SQLPutData*(). The amount
16520 of data can vary with each call for a given parameter or column. *StrLen_or_Ind* is ignored
16521 unless it is one of the following:

16522 • SQL_NTS, SQL_NULL_DATA, or SQL_DEFAULT_PARAM

16523 • The C data type specified in *SQLBindParameter*() or *SQLBindCol*() is SQL_C_CHAR or
16524 SQL_C_BINARY

16525 • The C data type is SQL_C_DEFAULT and the default C data type for the specified SQL
16526 data type is SQL_C_CHAR or SQL_C_BINARY.

16527 For all other types of C data, if *StrLen_or_Ind* is not SQL_NULL_DATA or
16528 SQL_DEFAULT_PARAM, the implementation assumes that the size of the **DataPtr* buffer is
16529 the size of the C data type specified with *ValueType* or *TargetType* and sends the entire data
16530 value. For more information, see Section D.7 on page 587.

16531 **RETURN VALUE**

16532 SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or
16533 SQL_INVALID_HANDLE.

16534 **DIAGNOSTICS**

16535 When *SQLPutData*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated
16536 SQLSTATE value can be obtained by calling *SQLGetDiagRec*() with a *HandleType* of
16537 SQL_HANDLE_STMT and a *Handle* of *StatementHandle*. The following table lists the SQLSTATE
16538 values commonly returned by *SQLPutData*(). The return code associated with each SQLSTATE
16539 value is SQL_ERROR, except that for SQLSTATE values in class 01, the return code is
16540 SQL_SUCCESS_WITH_INFO.

16541 01000 — General warning
16542 Implementation-defined informational message.

16543 01004 — String data, right truncation
16544 String or binary data returned for an output parameter resulted in the truncation of non-
16545 blank character or non-null binary data. If it was a string value, it was right-truncated.

16546 07006 — Restricted data type attribute violation
16547 The data value identified by *ValueType* in *SQLBindParameter*() for the bound parameter
16548 could not be converted to the data type identified by *ParameterType* in *SQLBindParameter*().

16549 07S01 — Invalid use of default parameter
16550 *StrLen_or_Ind* was SQL_DEFAULT_PARAM, and the corresponding parameter was not a
16551 parameter for a procedure called using the XDBC escape sequence (see Section 8.3 on page
16552 84).

16553 08S01 — Communication link failure
16554 The communication link to the data source failed before the function completed processing.

16555 22001 — String data, right truncation
16556 The assignment of a character or binary value to a column resulted in the truncation of
16557 non-blank (character) or non-null (binary) characters or octets.

16558 The SQL_NEED_LONG_DATA_LEN option in *SQLGetInfo*() was ''Y'' and more data was
16559 sent for a long parameter (the data type was SQL_LONGVARCHAR,
16560 SQL_LONGVARBINARY, or a long, data source-specific data type) than was specified with
16561 *StrLen_or_IndPtr* in *SQLBindParameter*().

16562 The SQL_NEED_LONG_DATA_LEN option in *SQLGetInfo*() was ''Y'' and more data was
16563 sent for a long column (the data type was SQL_LONGVARCHAR,
16564 SQL_LONGVARBINARY, or a long, data-source-specific data type) than was specified in
16565 the length buffer corresponding to a column in a row of data that was added or updated
16566 with *SQLBulkOperations*(), or updated with *SQLSetPos*().

16567 22003 — Numeric value out of range
16568 The data sent for a bound numeric parameter or column caused the whole (as opposed to
16569 fractional) part of the number to be truncated when assigned to the associated table column.

16570 Returning a numeric value (as numeric or string) for one or more input/output or output
16571 parameters would have caused the whole (as opposed to fractional) part of the number to
16572 be truncated.

16573 22007 — Invalid date/time format
16574 The data sent for a parameter or column that was bound to a date, time, or timestamp
16575 structure was, respectively, an invalid date, time, or timestamp.

16576 An input/output or output parameter was bound to a date, time, or timestamp C structure,
16577 and a value in the returned parameter was invalid for the data type.

16578 22008 — Date/time field overflow
16579 A date/time expression computed for an input/output or output parameter resulted in a
16580 date, time, or timestamp C structure that was invalid.

16581 22012 — Division by zero
16582 An arithmetic expression calculated for an input/output or output parameter resulted in
16583 division by zero.

16584 22015 — Interval field overflow
16585 The data sent for an exact numeric column or parameter to an interval structure was
16586 truncated with a loss of significant digits.

16587 The data sent for an interval column or parameter to an interval structure was truncated
16588 with a loss of significant digits.

16589 Column or parameter data was bound to an interval structure and there was no
16590 representation of the data in the interval structure.

16591 An input/output or output parameter that was an exact numeric value at the data source
16592 was bound to an interval C structure and returning the data caused a loss of significant
16593 digits.

16594 An input/output or output parameter that was an interval value at the data source was
16595 bound to an interval C structure and returning the data caused a loss of significant digits in
16596 the leading field.

16597 An input/output or output parameter was bound to an interval C structure, but there was
16598 no representation of the data in the interval data structure.

16599 22018 — Invalid character value for cast specification
16600 A character parameter or column was bound to an approximate numeric buffer and could
16601 not be cast to a valid approximate numeric value.

16602 A character parameter or column was bound to an exact numeric buffer and could not be
16603 cast to a valid exact numeric value.

16604 A character parameter or column was bound to date/time or interval buffer and could not
16605 be cast to a valid date/time or interval value.

16606 An input/output or output parameter that was a character value at the data source was
16607 bound to an approximate numeric C buffer and a value in the parameter could not be cast
16608 to a valid approximate numeric value.

16609 An input/output or output parameter was bound to an exact numeric C buffer and a value
16610 in the parameter could not be cast to a valid exact numeric value.

16611 An input/output or output parameter was bound to a date/time or interval C buffer and a
16612 value in the parameter could not be cast to a valid date/time or interval value.

16613 An input/output or output parameter was bound to a character C buffer and the parameter
16614 contained a character for which there was no representation in the character set of the
16615 target.

16616 HY000 — General error
16617 An error occurred for which there was no specific SQLSTATE and for which no
16618 implementation-specific SQLSTATE was defined. The error message returned by
16619 *SQLGetDiagRec*() in the **MessageText* buffer describes the error and its cause.

16620 HY001 — Memory allocation error
16621 The implementation failed to allocate memory required to support execution or completion
16622 of the function.

16623 HY008 — Operation canceled
16624 Asynchronous processing was enabled for *StatementHandle*. The function was called and
16625 before it completed execution, *SQLCancel*() was called on *StatementHandle*. The function
16626 was then called again on *StatementHandle*.

16627 The function was called and, before it completed execution, *SQLCancel*() was called on
16628 *StatementHandle* from a different thread in a multithread application.

16629 HY009 — Invalid use of null pointer
16630 *DataPtr* was a null pointer and *StrLen_or_Ind* was not 0, SQL_DEFAULT_PARAM, or
16631 SQL_NULL_DATA.

16632 HY010 — Function sequence error
16633 The previous function call was not a call to *SQLPutData*() or *SQLParamData*().

16634 An asynchronously executing function (not this one) was called for *StatementHandle* and
16635 was still executing when this function was called.

16636 HY011 — Attribute cannot be set now
16637 One or more calls to *SQLPutData*() for a single parameter have returned SQL_SUCCESS,
16638 and *SQLPutData*() was again called for that parameter with StrLen_or_IndPtrcbValue set to

16639 SQL_NULL_DATA.

16640 HY019 — Non-character and non-binary data sent in pieces
16641 *SQLPutData*() was called more than once for a parameter or column and it was not being
16642 used to send character C data to a column with a character, binary, or data source-specific
16643 data type or to send binary C data to a column with a character, binary, or data source-
16644 specific data type.

16645 HY020 — Attempt to concatenate a null value
16646 *SQLPutData*() was called more than once since the call that returned SQL_NEED_DATA,
16647 and in one of those calls, *StrLen_or_Ind* contained SQL_NULL_DATA or
16648 SQL_DEFAULT_PARAM.

16649 HY090 — Invalid string or buffer length
16650 *DataPtr* was not a null pointer and *StrLen_or_Ind* was less than 0, but not equal to SQL_NTS
16651 or SQL_NULL_DATA.

16652 HYT01 — Connection timeout expired
16653 The connection timeout period expired before the data source responded to the request. The
16654 connection timeout period is set through *SQLSetConnectAttr*(),
16655 SQL_ATTR_CONNECTION_TIMEOUT.

16656 IM001 — Function not supported
16657 The function is not supported on the current connection to the data source.

16658 **COMMENTS**

16659 *SQLPutData*() can be called to supply data-at-execution data for two uses: parameter data to be
16660 used in a call to *SQLExecute*() or *SQLExecDirect*(), or column data to be used when a row is
16661 updated or added by a call to *SQLBulkOperations*(), or updated by a call to *SQLSetPos*(). |

16662 When an application calls *SQLParamData*() to determine which data it should send, the |
16663 implementation returns the value that the application placed in the row-set buffer. The |
16664 application uses this value to determine which parameter data the implementation is requesting |
16665 on this occasion, or the address of the row where column data can be found. It also returns
16666 SQL_NEED_DATA, which is an indicator to the application that it should call *SQLPutData*() to
16667 send the data. The application points *DataPtr* to the buffer containing the actual data for the
16668 parameter or column.

16669 *SQLPutData*() returns SQL_NEED_DATA if more data needs to be sent, in which case the
16670 application calls *SQLPutData*() again. It returns SQL_SUCCESS if all data-at-execution data has
16671 been sent. The application then calls *SQLParamData*() again. If *SQLParamData*() returns |
16672 SQL_NEED_DATA, then it requires data for another parameter or column, and *SQLPutData*() is |
16673 called again. If *SQLParamData*() returns SQL_SUCCESS, then all data-at-execution data has been |
16674 sent, and the SQL statement can be executed or the *SQLBulkOperations*() or *SQLSetPos*() call can
16675 be processed.

16676 For more information on how data-at-execution parameter data is passed at statement execution
16677 time, see **Passing Parameter Values** on page 227. For more information on how data-at-
16678 execution column data is updated or added, see Section 12.3 on page 163 and |
16679 *SQLBulkOperations*().

16680 **Note:** An application can use *SQLPutData*() to send data in parts only when sending character C
16681 data to a column with a character, binary, or data source-specific data type or when sending
16682 binary C data to a column with a character, binary, or data source-specific data type. If
16683 *SQLPutData*() is called more than once under any other conditions, it returns SQL_ERROR and
16684 SQLSTATEHY019 (Non-character and non-binary data sent in pieces).

16685 This function can be used to send character or binary data values in parts to a column with a
16686 character, binary, or data source-specific data type (for example, parameters of the

16687 SQL_LONGVARBINARY or SQL_LONGVARCHAR types).

16688 **SEE ALSO**

16689	**For information about**	**See**
16690	Canceling statement processing	*SQLCancel*()
16691	Executing an SQL statement	*SQLExecDirect*()
16692	Executing a prepared SQL statement	*SQLExecute*()
16693	Returning the next parameter to send data for	*SQLParamData*()
16694	Binding a buffer to a parameter	*SQLBindParameter*()

16695 **CHANGE HISTORY**

16696 **Version 2**

16697 Revised generally. See **Alignment with Popular Implementations** on page 2.

16698 **NAME**

16699 SQLRowCount — Return the number of rows affected by certain database operations.

16700 **SYNOPSIS**

```
16701 SQLRETURN SQLRowCount(
16702   SQLHSTMT StatementHandle,
16703   SQLINTEGER * RowCountPtr);
```

16704 **ARGUMENTS**

16705 *StatementHandle* [Input]

16706 Statement handle.

16707 *RowCountPtr* [Output]

16708 Points to a buffer in which to return a row count.

16709 **RETURN VALUE**

16710 SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.

16711 **DIAGNOSTICS**

16712 When *SQLRowCount*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated
16713 SQLSTATE value can be obtained by calling *SQLGetDiagRec*() with a *HandleType* of
16714 SQL_HANDLE_STMT and a *Handle* of *StatementHandle*. The following table lists the SQLSTATE
16715 values commonly returned by *SQLRowCount*(). The return code associated with each
16716 SQLSTATEvalue is SQL_ERROR, except that for SQLSTATEvalues in class 01, the return code is
16717 SQL_SUCCESS_WITH_INFO.

16718 01000 — General warning

16719 Implementation-defined informational message.

16720 HY000 — General error

16721 An error occurred for which there was no specific SQLSTATE and for which no
16722 implementation-specific SQLSTATE was defined. The error message returned by
16723 *SQLGetDiagRec*() in the **MessageText* buffer describes the error and its cause.

16724 HY001 — Memory allocation error

16725 The implementation failed to allocate memory required to support execution or completion
16726 of the function.

16727 HY010 — Function sequence error

16728 The function was called prior to calling *SQLBulkOperations*(), *SQLExecDirect*(),
16729 *SQLExecute*(), or *SQLSetPos*() for *StatementHandle*.

16730 An asynchronously executing function was called for *StatementHandle* and was still
16731 executing when this function was called.

16732 *SQLBulkOperations*(), *SQLExecDirect*(), *SQLExecute*(), or *SQLSetPos*() was called for
16733 *StatementHandle* and returned SQL_NEED_DATA. This function was called before data was
16734 sent for all data-at-execution parameters or columns.

16735 HYT01 — Connection timeout expired

16736 The connection timeout period expired before the data source responded to the request. The
16737 connection timeout period is set through *SQLSetConnectAttr*(),
16738 SQL_ATTR_CONNECTION_TIMEOUT.

16739 IM001 — Function not supported

16740 The function is not supported on the current connection to the data source.

16741 **COMMENTS**

16742 *SQLRowCount*() returns the number of rows affected by an UPDATE, INSERT, or DELETE
16743 statement; the SQL_ADD, SQL_UPDATE_BY_BOOKMARK, or SQL_DELETE_BY_BOOKMARK

16744 operation in *SQLBulkOperations*(); or the SQL_UPDATE or SQL_DELETE operation in
16745 *SQLSetPos*(). The value returned is SQL_NO_TOTAL if the number of affected rows is not
16746 available.

16747 For all other operations on *StatementHandle*, the value returned in **RowCountPtr* is undefined.
16748 (Some data sources may provide useful information in other cases — for example, a data source
16749 may be able to return the number of rows returned by a SELECT statement or a catalog function
16750 before fetching the rows — but portable applications should not rely on this behavior.)

16751 When *SQLBulkOperations*(), *SQLExecute*(), *SQLExecDirect*(), *SQLMoreResults*(), or *SQLSetPos*() is
16752 called, the implementation sets the SQL_DIAG_ROW_COUNT field of the diagnostic data
16753 structure to the row count. The implementation also associates this value with *StatementHandle*
16754 independently of the diagnostic data structure, in order to be able to return it when the
16755 application calls *SQLRowCount*(). This independent value remains valid until *StatementHandle* is
16756 set back to the prepared or allocated state, the statement is re-executed, or *SQLCloseCursor*() is
16757 called.

16758 If a function has been called since the SQL_DIAG_ROW_COUNT field was set, the value
16759 returned by *SQLRowCount*() might be different from the value in the SQL_DIAG_ROW_COUNT
16760 field, because the SQL_DIAG_ROW_COUNT field is reset to 0 by any function call.

16761 **SEE ALSO**

16762	**For information about**	**See**
16763	Executing an SQL statement	*SQLExecDirect*()
16764	Executing a prepared SQL statement	*SQLExecute*()

16765 **CHANGE HISTORY**

16766 **Version 2**

16767 Revised generally. See **Alignment with Popular Implementations** on page 2.

16768 **NAME**

16769 SQLSetConnectAttr — Set attributes that govern aspects of connections.

16770 **SYNOPSIS**

```
16771 SQLRETURN SQLSetConnectAttr(
16772   SQLHDBC ConnectionHandle,
16773   SQLINTEGER Attribute,
16774   SQLPOINTER ValuePtr,
16775   SQLINTEGER StringLength);
```

16776 **ARGUMENTS**

16777 *ConnectionHandle* [Input]

16778 Connection handle.

16779 *Attribute* [Input]

16780 Attribute to set, listed in **Connection Attributes** on page 459.

16781 *ValuePtr* [Input]

16782 Pointer to the value to be associated with *Attribute*. Depending on the value of *Attribute*,
16783 **ValuePtr* is a 32-bit unsigned integer value or points to a null-terminated character string.
16784 For implementation-defined values of *Attribute*, the value in **ValuePtr* may be a signed
16785 integer.

16786 *StringLength* [Input]

16787 If *ValuePtr* points to a character string or a binary buffer, then *StringLength* should be the
16788 length of **ValuePtr*. If *ValuePtr* is a pointer, but not to a string or binary buffer, then
16789 *StringLength* should have the value SQL_IS_POINTER. If *ValuePtr* is not a pointer, then
16790 *StringLength* should have the value SQL_IS_NOT_POINTER.

16791 **RETURN VALUE**

16792 SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.

16793 **DIAGNOSTICS**

16794 When *SQLSetConnectAttr*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated
16795 SQLSTATE value can be obtained by calling *SQLGetDiagRec*() with a *HandleType* of
16796 SQL_HANDLE_DBC and a *Handle* of *ConnectionHandle*. The following table lists the SQLSTATE
16797 values commonly returned by *SQLSetConnectAttr*(). The return code associated with each
16798 SQLSTATEvalue is SQL_ERROR, except that for SQLSTATEvalues in class 01, the return code is
16799 SQL_SUCCESS_WITH_INFO.

16800 The implementation can return SQL_SUCCESS_WITH_INFO to provide information about the
16801 result of setting an option.

16802 01000 — General warning

16803 Implementation-defined informational message.

16804 01S02 — Attribute value changed

16805 The data source did not support the value specified in **ValuePtr* and substituted a similar
16806 value.

16807 08003 — Connection does not exist

16808 *Attribute* required an open connection, but *ConnectionHandle* was not in a connected state.

16809 08007 — Connection failure during transaction

16810 The connection associated with *ConnectionHandle* failed during the execution of the function
16811 and it cannot be determined whether the requested COMMIT or ROLLBACK occurred
16812 before the failure. This would occur if the connection was in auto-commit mode and the
16813 connection failed while completing previous work.

16814 08S01 — Communication link failure
16815 The communication link to the data source failed before the function completed processing.

16816 24000 — Invalid cursor state
16817 *Attribute* was SQL_ATTR_CURRENT_CATALOG and a result set was pending.

16818 HY000 — General error
16819 An error occurred for which there was no specific SQLSTATE and for which no
16820 implementation-specific SQLSTATE was defined. The error message returned by
16821 *SQLGetDiagRec*() in the **MessageText* buffer describes the error and its cause.

16822 HY001 — Memory allocation error
16823 The implementation failed to allocate memory required to support execution or completion
16824 of the function.

16825 HY009 — Invalid use of null pointer
16826 *Attribute* identified an attribute that required a string value, and *ValuePtr* is a null pointer.

16827 HY010 — Function sequence error
16828 An asynchronously executing function was called for a statement handle associated with
16829 *ConnectionHandle* and was still executing when *SQLSetConnectAttr*() was called.

16830 *SQLBulkOperations*(), *SQLExecDirect*(), *SQLExecute*(), or *SQLSetPos*() was called for a
16831 statement handle associated with *ConnectionHandle* and returned SQL_NEED_DATA. This
16832 function was called before data was sent for all data-at-execution parameters or columns.

16833 *SQLBrowseConnect*() was called for *ConnectionHandle* and returned SQL_NEED_DATA. This
16834 function was called before *SQLBrowseConnect*() returned SQL_SUCCESS_WITH_INFO or
16835 SQL_SUCCESS.

16836 HY011 — Attribute cannot be set now
16837 *Attribute* was SQL_ATTR_TXN_ISOLATION and a transaction was open.

16838 *Attribute* was SQL_ATTR_PACKET_SIZE and the connection has already been established.

16839 HY024 — Invalid attribute value
16840 A value was specified in **ValuePtr* that is inapplicable to *Attribute*, or **ValuePtr* was an
16841 empty string and *Attribute* requires a non-empty string.

16842 HY090 — Invalid string or buffer length
16843 *StringLength* was less than 0, but was not SQL_NTS.

16844 HY092 — Invalid attribute identifier
16845 *Attribute* was not valid for this connection to this data source.

16846 *Attribute* identifies a read-only attribute.

16847 HYC00 — Optional feature not implemented
16848 *Attribute* was a valid connection or statement attribute but is not supported by the data
16849 source.

16850 HYT01 — Connection timeout expired
16851 The connection timeout period expired before the data source responded to the request. The
16852 connection timeout period is set through *SQLSetConnectAttr*(),
16853 SQL_ATTR_CONNECTION_TIMEOUT.

16854 IM001 — Function not supported
16855 The function is not supported on the current connection to the data source.

16856 When *Attribute* is a statement attribute, *SQLSetConnectAttr*() can return any SQLSTATEs
16857 returned by *SQLSetStmtAttr*().

16858 **COMMENTS**

16859 The currently-defined attributes are shown below; additional attributes are likely to be defined
16860 to take advantage of different data sources. A range of attributes is reserved by XDBC;
16861 implementors must reserve values for vendor-specific uses from X/Open (see Section 1.8 on
16862 page 21).

16863 The information in the **ValuePtr* buffer must follow a format determined by the specified
16864 attribute:

16865 • Some attributes are character strings. For variable-length strings, *StringLength* specifies the
16866 length of the string in octets. For strings whose length is dictated by a specification, the
16867 implementation ignores *StringLength*. (There are no fixed-length string attributes in XDBC.)

16868 • Some attributes are 32-bit integers; for these, the implementation ignores *StringLength*.

16869 The type of data required for each attribute is indicated in the list of valid values for *Attribute*.

16870 **SQLSetConnectAttr() and Statement Attributes**

16871 An application can call *SQLSetConnectAttr*() and specify as *Attribute* a manifest constant that this
16872 specification lists as a statement attribute (see **Statement Attributes** on page 506). This call sets
16873 the value of that statement attribute for any statements already associated with
16874 *ConnectionHandle*, and establishes the value as a default value for any statements later allocated
16875 for *ConnectionHandle*. If *SQLSetConnectAttr*() returns an error when a statement attribute is set
16876 on one of multiple active statements, the statement attribute is established as the default for
16877 statements later allocated on *ConnectionHandle*, but it is unspecified which attributes of existing
16878 statements are changed by the call.

16879 **Note:** Applications should set a statement attribute on the connection level only to establish the
16880 default value for future statements allocated on the connection. Setting a statement attribute for
16881 multiple active statements on a connection is problematic and may result in undefined effects.

16882 For statement attributes that serve to set the header field of a descriptor, use of
16883 *SQLSetConnectAttr*() to set the statement attribute serves to modify all application descriptors
16884 currently associated with all statements on *ConnectionHandle*. It also becomes the default value
16885 for use in the four implicit descriptors that are allocated when a new statement handle is
16886 allocated on *ConnectionHandle*. However, the value does not become a default value for
16887 descriptors that may be associated with the statements on *ConnectionHandle* in the future.

16888 SQL_ATTR_ASYNC_ENABLE is a special example of a statement attribute that can be set by
16889 calling *SQLSetConnectAttr*(). For implementations that provide asynchrony on the connection
16890 level, calling *SQLSetConnectAttr*() may be the only meaningful way to set this statement
16891 attribute.

16892 **Persistence of Connection Attributes**

16893 An application can call *SQLSetConnectAttr*() at any time between the time the connection is
16894 allocated and the time it is freed. All connection and statement attributes successfully set by the
16895 application for the connection persist until *SQLFreeHandle*() is called on the connection. For
16896 example, if an application calls *SQLSetConnectAttr*() before connecting to a data source, the
16897 attribute persists even if *SQLSetConnectAttr*() the data source rejects the attribute. If an
16898 application specifies a data-source-specific attribute, the implementation retains the attribute
16899 value even if the application connects to a different data source.

16900 **Changed Connection Attributes**

16901 Some connection and statement attributes support substitution of a similar value if the data
16902 source does not support the value specified in **ValuePtr*. In such cases, the function returns
16903 SQL_SUCCESS_WITH_INFO and SQLSTATE 01S02 (Attribute value changed). For example, if
16904 *Attribute* is SQL_ATTR_PACKET_SIZE and **ValuePtr* exceeds the maximum packet size the data |
16905 source supports, the data source can substitute a lower value. If *Attribute* is |
16906 SQL_ATTR_PACKET_SIZE and the packet size cannot be set on *ConnectionHandle*, this is |
16907 signified by a substituted value of 0. To determine the substituted value, an application calls
16908 *SQLGetConnectAttr*() (for connection attributes) or *SQLGetStmtAttr*() (for statement attributes).

16909 **Connection Attributes**

16910 The caller sets *Attribute* to one of the values listed below to obtain the following connection
16911 attribute in **ValuePtr*.

16912 SQL_ATTR_ACCESS_MODE
16913 A 32-bit integer value that indicates the access mode: |

16914 SQL_MODE_READ_WRITE |
16915 The default value. Reads and writes may occur on the connection. |

16916 SQL_MODE_READ_ONLY |
16917 An indication that the implementation need not support SQL statements that cause
16918 updates to occur. This mode can be used to optimize locking strategies, transaction
16919 management, or other areas as appropriate to the implementation. The effect if the |
16920 application submits such an SQL statement is implementation-defined. |

16921 SQL_ATTR_ASYNC_ENABLE |
16922 This is technically a statement attribute and is described fully in the list of statement |
16923 attributes in *SQLSetStmtAttr*(). For implementations that provide asynchrony on the |
16924 connection level, calling *SQLSetConnectAttr*() may be the only meaningful way to set this |
16925 statement attribute.

16926 SQL_ATTR_AUTO_IPD
16927 A read-only 32-bit integer value that indicates automatic population of the IPD:

16928 SQL_TRUE The implementation automatically populates the IPD after a call to
16929 *SQLPrepare*().

16930 SQL_FALSE The implementation does not automatically populate the IPD after
16931 a call to *SQLPrepare*(). Any data source that does not support |
16932 prepared statements returns SQL_FALSE.

16933 If SQL_TRUE is returned for the SQL_ATTR_AUTO_IPD connection attribute, the
16934 statement attribute SQL_ATTR_ENABLE_AUTO_IPD can be set to turn automatic
16935 population of the IPD on or off. If SQL_ATTR_AUTO_IPD is SQL_FALSE,
16936 SQL_ATTR_ENABLE_AUTO_IPD cannot be set to SQL_TRUE. The default value of
16937 SQL_ATTR_ENABLE_AUTO_IPD is equal to the value of SQL_ATTR_AUTO_IPD.

16938 This connection attribute can be returned by *SQLGetConnectAttr*(), but cannot be set by
16939 *SQLSetConnectAttr*().

16940 SQL_ATTR_AUTOCOMMIT
16941 A 32-bit integer value that specifies whether to use auto-commit or manual-commit mode |
16942 (see Section 14.1.2 on page 182):

16943 SQL_AUTOCOMMIT_OFF
16944 The implementation uses manual-commit mode, and the application must explicitly
16945 commit or roll back transactions with *SQLEndTran*().

16946 SQL_AUTOCOMMIT_ON
16947 The data source uses auto-commit mode. Each statement is committed immediately
16948 after it is executed. This is the default.

16949 It is implementation-defined whether changing from manual-commit mode to auto-commit
16950 mode commits any open transactions on the connection.

16951 **Transaction completion may have side-effects on cursors and on access plans of prepared**
16952 **statements. See Section 14.1.3 on page 184.**

16953 SQL_ATTR_CONNECTION_TIMEOUT
16954 A 32-bit integer value corresponding to the number of seconds to wait for any request on
16955 the connection to complete before returning to the application. The implementation should
16956 return SQLSTATE HYT00 (Timeout expired) whenever it is possible to timeout in a
16957 situation not associated with query execution or login.

16958 SQL_ATTR_CURRENT_CATALOG
16959 A character string containing the name of the catalog to be used by the data source. For
16960 example, if the catalog is a database, an implementation might send a USE statement to the
16961 data source. If the catalog is a directory, an implementation might make the specified
16962 directory the current directory.

16963 SQL_ATTR_LOGIN_TIMEOUT
16964 A 32-bit integer value corresponding to the number of seconds to wait for a login request to
16965 complete before returning to the application. The default is implementation-defined. If
16966 **ValuePtr* is 0, the timeout is disabled and connection attempts wait indefinitely.

16967 If the specified timeout exceeds the maximum login timeout in the data source, the
16968 implementation substitutes that value and returns SQLSTATE 01S02 (Attribute value
16969 changed).

16970 SQL_ATTR_PACKET_SIZE
16971 A 32-bit integer value specifying the network packet size in octets.

16972 **Note:** Many data sources either do not support this option or can only return the network
16973 packet size.

16974 If the specified size exceeds the maximum packet size or is smaller than the minimum
16975 packet size, the implementation substitutes that value and returns SQLSTATE 01S02
16976 (Attribute value changed).

16977 If the packet size is set after a connection has already been made, the implementation
16978 returns SQLSTATEHY011 (Attribute cannot be set now).

16979 SQL_ATTR_QUIET_MODE
16980 A 32-bit pointer to a context for user interaction. For instance, this attribute might be a
16981 window handle inside which dialog boxes appear.

16982 Setting this attribute to a null pointer inhibits all interaction between the implementation
16983 and the user.

16984 This attribute does not apply to user interaction pursuant to a call to *SQLDriverConnect*();
16985 instead, any user interaction takes place in a context specified by the *WindowHandle*
16986 argument of that function.

16987 SQL_ATTR_TXN_ISOLATION
16988 A 32-bit bitmask that sets the transaction isolation level for the current connection. An
16989 application must call *SQLEndTran*() to complete all open transactions on *ConnectionHandle*
16990 before calling *SQLSetConnectAttr*() with this option. The valid values for **ValuePtr* can be
16991 determined by calling *SQLGetInfo*() with the SQL_TXN_ISOLATION_OPTION option.

16992 Transaction isolation is discussed in detail in Section 14.2.2 on page 186.

16993 **SEE ALSO**

For information about	See
Returning the setting of a connection attribute	*SQLGetConnectAttr*()
Returning the setting of a statement attribute	*SQLGetStmtAttr*()
Setting a statement attribute	*SQLSetStmtAttr*()
Allocating a handle	*SQLAllocHandle*()

16999 **CHANGE HISTORY**

17000 **Version 2**

17001 Revised generally. See **Alignment with Popular Implementations** on page 2.

17002 **New Connection Attributes in Version 2**

17003 The following connection attributes are new in this issue:

17004 SQL_ATTR_ACCESS_MODE SQL_ATTR_LOGIN_TIMEOUT
17005 SQL_ATTR_ASYNC_ENABLE SQL_ATTR_PACKET_SIZE |
17006 SQL_ATTR_AUTOCOMMIT SQL_ATTR_QUIET_MODE |
17007 SQL_ATTR_CONNECTION_TIMEOUT SQL_ATTR_TXN_ISOLATION |
17008 SQL_ATTR_CURRENT_CATALOG |

17009 **NAME**

17010 SQLSetCursorName — Set the name of a cursor.

17011 **SYNOPSIS**

```
17012 SQLRETURN SQLSetCursorName(
17013     SQLHSTMT StatementHandle,
17014     SQLCHAR * CursorName,
17015     SQLSMALLINT NameLength);
```

17016 **ARGUMENTS**

17017 *StatementHandle* [Input]
17018 Statement handle.

17019 *CursorName* [Input]
17020 Cursor name. For efficient processing, the cursor name should not include any leading or
17021 trailing spaces in the cursor name, and if the cursor name includes a delimited identifier, the
17022 delimiter should be the first character in the cursor name.

17023 *NameLength* [Input]
17024 Length of **CursorName*.

17025 **RETURN VALUE**

17026 SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.

17027 **DIAGNOSTICS**

17028 When *SQLSetCursorName*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated
17029 SQLSTATE value can be obtained by calling *SQLGetDiagRec*() with a *HandleType* of
17030 SQL_HANDLE_STMT and a *Handle* of *StatementHandle*. The following table lists the SQLSTATE
17031 values commonly returned by *SQLSetCursorName*(). The return code associated with each
17032 SQLSTATEvalue is SQL_ERROR, except that for SQLSTATEvalues in class 01, the return code is
17033 SQL_SUCCESS_WITH_INFO.

17034 01000 — General warning
17035 Implementation-defined informational message.

17036 01004 — String data, right truncation
17037 The cursor name exceeded the maximum length and only that number of characters has
17038 been used. Portable applications should not generate cursor names longer than
17039 SQL_MAX_ID_LENGTH characters.

17040 24000 — Invalid cursor state
17041 *StatementHandle* was already in an executed or cursor-positioned state.

17042 34000 — Invalid cursor name
17043 The cursor name specified in **CursorName* was invalid, because it exceeded the
17044 implementation-defined maximum length, or started with "SQLCUR" or "SQL_CUR", or
17045 already exists.

17046 HY000 — General error
17047 An error occurred for which there was no specific SQLSTATE and for which no
17048 implementation-specific SQLSTATE was defined. The error message returned by
17049 *SQLGetDiagRec*() in the **MessageText* buffer describes the error and its cause.

17050 HY001 — Memory allocation error
17051 The implementation failed to allocate memory required to support execution or completion
17052 of the function.

17053 HY009 — Invalid use of null pointer
17054 *CursorName* was a null pointer.

17055 HY010 — Function sequence error
17056 An asynchronously executing function was called for *StatementHandle* and was still
17057 executing when this function was called.

17058 *SQLBulkOperations*(), *SQLExecDirect*(), *SQLExecute*(), or *SQLSetPos*() was called for
17059 *StatementHandle* and returned SQL_NEED_DATA. This function was called before data was
17060 sent for all data-at-execution parameters or columns.

17061 HY090 — Invalid string or buffer length
17062 *NameLength* was less than 0, but not equal to SQL_NTS.

17063 HYT01 — Connection timeout expired
17064 The connection timeout period expired before the data source responded to the request. The
17065 connection timeout period is set through *SQLSetConnectAttr*(),
17066 SQL_ATTR_CONNECTION_TIMEOUT.

17067 IM001 — Function not supported
17068 The function is not supported on the current connection to the data source.

17069 **COMMENTS**

17070 Cursor names are used only in positioned UPDATE and DELETE statements. If the application
17071 does not call *SQLSetCursorName*() to define a cursor name, then when it executes a query, the
17072 implementation generates a name that begins with SQL_CUR and does not exceed
17073 SQL_MAX_ID_LENGTH characters in length.

17074 All cursor names within the connection must be unique. The maximum length of a cursor name
17075 is implementation-defined. Portable applications should limit cursor names to
17076 SQL_MAX_ID_LENGTH characters. If a cursor name is a quoted identifier, it is treated in a
17077 case-sensitive manner, and it can contain characters otherwise not permitted in identifiers, such
17078 as blanks or reserved keywords. If an application requires a cursor name to be treated in a case-
17079 sensitive manner, it must pass it as a quoted identifier.

17080 A cursor name that is set either explicitly or implicitly remains set until the statement with
17081 which it is associated is dropped, using *SQLFreeHandle*(). *SQLSetCursorName*() can be called to
17082 rename a cursor on a statement as long as the cursor is in an allocated or prepared state.

17083 **SEE ALSO**

17084	**For information about**	**See**
17085	Executing an SQL statement	*SQLExecDirect*()
17086	Executing a prepared SQL statement	*SQLExecute*()
17087	Returning a cursor name	*SQLGetCursorName*()

17088 **CHANGE HISTORY**

17089 **Version 2**

17090 Revised generally. See **Alignment with Popular Implementations** on page 2.

SQLSetDescField — Set the value of a single field of a descriptor record.

SYNOPSIS

```
SQLRETURN SQLSetDescField(
  SQLHDESC DescriptorHandle,
  SQLSMALLINT RecNumber,
  SQLSMALLINT FieldIdentifier,
  SQLPOINTER ValuePtr,
  SQLINTEGER BufferLength);
```

ARGUMENTS

DescriptorHandle [Input]
: Descriptor handle.

RecNumber [Input]
: Indicates the descriptor record containing the field that the application seeks to set. Descriptor records are numbered from 0, with record number 0 being the bookmark record. The implementation ignores *RecNumber* if *FieldIdentifier* specifies a header field.

FieldIdentifier [Input]
: Indicates the field of the descriptor whose value is to be set. For more information, see **FieldIdentifier Argument** on page 472.

ValuePtr [Input]
: Pointer to a buffer containing the descriptor information, or a 4-octet value. The data type depends on the value of *FieldIdentifier*. If *ValuePtr* is a 4-octet value, either all four octets are used, or just two of the four are used, depending on the value of *FieldIdentifier*.

BufferLength [Input]
: If *ValuePtr* points to a character string or a binary buffer, this argument should be the length of **ValuePtr*. If *ValuePtr* is a pointer, but not to a string or binary buffer, then *BufferLength* should have the value SQL_IS_POINTER. If *ValuePtr* is not a pointer, then *BufferLength* should have the value SQL_IS_NOT_POINTER.

RETURN VALUE

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.

DIAGNOSTICS

When *SQLSetDescField*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling *SQLGetDiagRec*() with a *HandleType* of SQL_HANDLE_DESC and a *Handle* of *DescriptorHandle*. The following table lists the SQLSTATE values commonly returned by *SQLSetDescField*(). The return code associated with each SQLSTATEvalue is SQL_ERROR, except that for SQLSTATEvalues in class 01, the return code is SQL_SUCCESS_WITH_INFO.

01000 — General warning
: Implementation-defined informational message.

01S02 — Attribute value changed
: The data source did not support the value specified in **ValuePtr* (if *ValuePtr* was a pointer) or the value in *ValuePtr* (if *ValuePtr* was a 4-octet value), or **ValuePtr* was invalid because of SQL constraints or requirements, so the implementation substituted a similar value.

07006 — Restricted data type attribute violation
: *DescriptorHandle* referred to an application descriptor, *RecNumber* was 0, *FieldIdentifier* was SQL_DESC_TYPE or SQL_DESC_CONCISE_TYPE, and *ValuePtr* was not SQL_C_VARBOOKMARK.

17137 07009 — Invalid descriptor index
17138 *FieldIdentifier* was a record field, *RecNumber* was 0, and *DescriptorHandle* referred to an IPD.

17139 *RecNumber* was less than 0 and *DescriptorHandle* referred to an APD or an ARD.

17140 *RecNumber* was greater than the maximum number of columns or parameters that the data
17141 source supports, and *DescriptorHandle* referred to an APD or an ARD.

17142 *FieldIdentifier* was SQL_DESC_COUNT, and *ValuePtr* was less than 0.

17143 08S01 — Communication link failure
17144 The communication link to the data source failed before the function completed processing.

17145 HY000 — General error
17146 An error occurred for which there was no specific SQLSTATE and for which no
17147 implementation-specific SQLSTATE was defined. The error message returned by
17148 *SQLGetDiagRec*() in the **MessageText* buffer describes the error and its cause.

17149 HY001 — Memory allocation error
17150 The implementation failed to allocate memory required to support execution or completion
17151 of the function.

17152 HY009 — Invalid use of null pointer
17153 *FieldIdentifier* was SQL_DESC_NAME and *ValuePtr* was a null pointer.

17154 HY010 — Function sequence error
17155 *DescriptorHandle* was associated with a statement handle for which an asynchronously
17156 executing function (not this one) was called and was still executing when this function was
17157 called.

17158 *SQLBulkOperations*(), *SQLExecDirect*(), *SQLExecute*(), or *SQLSetPos*() was called for the
17159 statement handle with which *DescriptorHandle* was associated and returned
17160 SQL_NEED_DATA. This function was called before data was sent for all data-at-execution
17161 parameters or columns.

17162 HY016 — Cannot modify an implementation row descriptor
17163 *DescriptorHandle* referred to an IRD, and *FieldIdentifier* was not
17164 SQL_DESC_ARRAY_STATUS_PTR.

17165 HY021 — Inconsistent descriptor information
17166 The SQL_DESC_TYPE field, or any other field associated with it in the descriptor, was not
17167 valid or consistent.

17168 *DescriptorHandle* referred to an application descriptor, and the SQL_DESC_TYPE field was
17169 not one of the valid XDBC C types or an implementation-defined C type.

17170 The descriptor consistency check failed (see **Consistency Checks** on page 486).

17171 HY091 — Invalid descriptor field identifier
17172 *FieldIdentifier* was not an XDBC-defined value nor an implementation-defined value.

17173 *FieldIdentifier* was an invalid value for *DescriptorHandle*.

17174 *RecNumber* was greater than the value in the SQL_DESC_COUNT field.

17175 *FieldIdentifier* was a field defined by this specification as a read-only field.

17176 *FieldIdentifier* was SQL_DESC_UNNAMED and **ValuePtr* was SQL_NAMED.

17177 HY105 — Invalid parameter type
17178 The SQL_DESC_PARAMETER_TYPE field was invalid. (For more information, see
17179 **InputOutputType Argument** on page 223.)

HYT01 — Connection timeout expired

The connection timeout period expired before the data source responded to the request. The connection timeout period is set through *SQLSetConnectAttr*(), SQL_ATTR_CONNECTION_TIMEOUT.

IM001 — Function not supported

The function is not supported on the current connection to the data source.

COMMENTS

An application can call *SQLSetDescField*() to set any single descriptor field, except read-only fields.

The information in the **ValuePtr* buffer must follow a format determined by the specified attribute:

- Some fields are character strings. For variable-length strings, *BufferLength* specifies the length of the string in octets. For strings whose length is dictated by a specification, the implementation ignores *BufferLength*. (There are no fixed-length string fields in XDBC.)
- Some fields are 32-bit integers; for these, the implementation ignores *BufferLength*.

The type of data required for each field is indicated in the list of valid values for *FieldIdentifier*.

If a call to *SQLSetDescField*() fails, the content of the descriptor field it would have set is undefined.

Other functions can be called to set multiple descriptor fields at once. *SQLSetDescRec*() sets a variety of fields that affect the data type and buffer bound to a column or parameter. *SQLBindCol*() or *SQLBindParameter*() makes a complete specification for the binding of a column or parameter. These functions set a specific group of descriptor fields with one function call.

SQLSetDescField() can be called to specify a binding offset when row-wise binding is used. This changes the effective addresses of the binding pointers (SQL_DESC_DATA_PTR, SQL_DESC_INDICATOR_PTR, or SQL_DESC_OCTET_LENGTH_PTR) without requiring calls to *SQLBindCol*() or *SQLBindParameter*(). This lets an application change SQL_DESC_DATA_PTR without changing other fields, for instance SQL_DESC_DATA_TYPE.

Descriptor header fields are set by calling *SQLSetDescField*() with a *RecNumber* of 0, and the appropriate *FieldIdentifier*. Header fields that contain statement attributes can also be set by a call to *SQLSetStmtAttr*(). This lets applications set a statement attribute without first obtaining a descriptor handle.

The application sets *RecNumber* to 0 to set bookmark fields. (The application should always set the SQL_ATTR_USE_BOOKMARKS statement attribute before calling *SQLSetDescField*() to set bookmark fields.)

Initialization of Descriptor Fields

The following tables describe the usage and defaulting of descriptor fields. This information depends on whether the descriptor is an ARD, APD, IRD, or IPD.

The **R/W** column shows whether the field is read/write (R/W), read-only (R/O), or unused by any of the functions that use descriptors. Only read-write fields can be set by calling *SQLSetDescField*().

The **Default** column shows the initial value of the field when a descriptor is allocated. The legend D indicates that there is a default. (For IRDs, the default depends on the prepared or executed statement.) ND indicates that there is no default. For unused fields, the default is undefined and the word Unused is repeated in this column. Any other text in this column indicates a specific default value for the field.

The initialization of header fields is as follows:

Field Name and (Type)	R/W	Default
SQL_DESC_ALLOC_TYPE (SQLSMALLINT)	ARD: R/O APD: R/O IRD: R/O IPD: R/O	ARD: APD: IRD: <AUTO> IPD: <AUTO>
SQL_DESC_ARRAY_SIZE (SQLUINTEGER)	ARD: R/W APD R/W IRD: Unused IPD: Unused	ARD: 1 APD: 1 IRD: Unused IPD: Unused
SQL_DESC_ARRAY_STATUS_PTR (SQLUSMALLINT*)	ARD: R/W APD: R/W IRD: R/W IPD: R/W	ARD: Null ptr APD: Null ptr IRD: Null ptr IPD: Null ptr
SQL_DESC_BIND_OFFSET_PTR (SQLINTEGER*)	ARD: R/W APD: R/W IRD: Unused IPD: Unused	ARD: Null ptr APD: Null ptr IRD: Unused IPD: Unused
SQL_DESC_BIND_TYPE (SQLINTEGER)	ARD: R/W APD: R/W IRD: Unused IPD: Unused	ARD: 0 APD: 0 IRD: Unused IPD: Unused
SQL_DESC_COUNT (SQLSMALLINT)	ARD: R/W APD: R/W IRD: R/O IPD: R/W	ARD: 0 APD: 0 IRD: D IPD: 0
SQL_DESC_ROWS_PROCESSED_PTR (SQLUINTEGER*)	ARD: Unused APD: Unused IRD: R/W IPD: R/W	ARD: Unused APD: Unused IRD: Null ptr IPD: Null ptr

This field, which specifies the allocation type of the field, is set to SQL_DESC_ALLOC_AUTO for automatically-allocated descriptors (including all IRDs and IPDs) and SQL_DESC_ALLOC_USER for descriptors the user explicitly allocates.

The initialization of record fields is as follows:

Field Name and (Type)	R/W	Default
SQL_DESC_AUTO_UNIQUE_VALUE (SQLINTEGER)	ARD: Unused APD: Unused IRD: R/O IPD: Unused	ARD: Unused APD: Unused IRD: D IPD: Unused
SQL_DESC_BASE_COLUMN_NAME (SQLCHAR)	ARD: Unused APD: Unused IRD: R/O IPD: Unused	ARD: Unused APD: Unused IRD: D IPD: Unused
SQL_DESC_BASE_TABLE_NAME (SQLCHAR)	ARD: Unused APD: Unused IRD: R/O IPD: Unused	ARD: Unused APD: Unused IRD: D IPD: Unused
SQL_DESC_CASE_SENSITIVE (SQLINTEGER)	ARD: Unused APD: Unused IRD: R/O IPD: R/O	ARD: Unused APD: Unused IRD: D IPD: D [1]
SQL_DESC_CATALOG_NAME (SQLCHAR)	ARD: Unused APD: Unused IRD: R/O IPD: Unused	ARD: Unused APD: Unused IRD: D IPD: Unused
SQL_DESC_CONCISE_TYPE (SQLSMALLINT)	ARD: R/W APD: R/W IRD: R/O IPD: R/W	ARD: SQL_C_DEFAULT APD: SQL_C_DEFAULT IRD: D IPD: ND
SQL_DESC_DATA_PTR (SQLPOINTER)	ARD: R/W APD: R/W IRD: Unused IPD: Unused	ARD: Null ptr APD: Null ptr IRD: Unused IPD: Unused [2]
SQL_DESC_DATETIME- _INTERVAL_CODE (SQLSMALLINT)	ARD: R/W APD: R/W IRD: R/O IPD: R/W	ARD: ND APD: ND IRD: D IPD: ND
SQL_DESC_DATETIME- _INTERVAL_PRECISION (SQLINTEGER)	ARD: R/W APD: R/W IRD: R/O IPD: R/W	ARD: ND APD: ND IRD: D IPD: ND
SQL_DESC_DISPLAY (SQLINTEGER)	ARD: Unused APD: Unused IRD: R/O IPD: Unused	ARD: Unused APD: Unused IRD: D IPD: Unused

Line	Field	Access	Default
17300	SQL_DESC_FIXED-	ARD: Unused	ARD: Unused
17301	_PREC_SCALE	APD: Unused	APD: Unused
17302	(SQLSMALLINT)	IRD: R/O	IRD: D
17303		IPD: R/O	IPD: D [1]
17304	SQL_DESC_INDICATOR_PTR	ARD: R/W	ARD: Null ptr
17305	(SQLINTEGER *)	APD: R/W	APD: Null ptr
17306		IRD: Unused	IRD: Unused
17307		IPD: Unused	IPD: Unused
17308	SQL_DESC_LABEL	ARD: Unused	ARD: Unused
17309	(SQLCHAR)	APD: Unused	APD: Unused
17310		IRD: R/O	IRD: D
17311		IPD: Unused	IPD: Unused
17312	SQL_DESC_LENGTH	ARD: R/W	ARD: ND
17313	(SQLUINTEGER)	APD: R/W	APD: ND
17314		IRD: R/O	IRD: D
17315		IPD: R/W	IPD: ND
17316	SQL_DESC_LITERAL-PREFIX	ARD: Unused	ARD: Unused
17317	(SQLCHAR)	APD: Unused	APD: Unused
17318		IRD: R/O	IRD: D
17319		IPD: Unused	IPD: Unused
17320	SQL_DESC_LITERAL-SUFFIX	ARD: Unused	ARD: Unused
17321	(SQLCHAR)	APD: Unused	APD: Unused
17322		IRD: R/O	IRD: D
17323		IPD: Unused	IPD: Unused
17324	SQL_DESC_LOCAL-	ARD: Unused	ARD: Unused
17325	_TYPE_NAME	APD: Unused	APD: Unused
17326	(SQLCHAR)	IRD: R/O	IRD: D
17327		IPD: R/O	IPD: D
17328	SQL_DESC_NAME	ARD: Unused	ARD: ND
17329	(SQLCHAR *)	APD: Unused	APD: ND
17330		IRD: R/O	IRD: D
17331		IPD: R/W	IPD: ND
17332	SQL_DESC_NULLABLE	ARD: Unused	ARD: ND
17333	(SQLSMALLINT)	APD: Unused	APD: ND
17334		IRD: R/O	IRD: D
17335		IPD: R/O	IPD: ND
17336	SQL_DESC_OCTET_LENGTH	ARD: R/W	ARD: ND
17337	(SQLINTEGER *)	APD: R/W	APD: ND
17338		IRD: R/O	IRD: D
17339		IPD: R/W	IPD: ND
17340	SQL_DESC_OCTET_LENGTH_PTR	ARD: R/W	ARD: Null ptr
17341	(SQLINTEGER)	APD: R/W	APD: Null ptr
17342		IRD: Unused	IRD: Unused
17343		IPD: Unused	IPD: Unused

Line	Field	Access	Default
17344	SQL_DESC_PARAMETER_TYPE	ARD: Unused	ARD: Unused
17345	(SQLSMALLINT)	APD: Unused	APD: Unused
17346		IRD: Unused	IRD: Unused
17347		IPD: R/W	IPD: D=SQL_PARAM_INPUT
17348	SQL_DESC_PRECISION	ARD: R/W	ARD: ND
17349	(SQLSMALLINT)	APD: R/W	APD: ND
17350		IRD: R/O	IRD: D
17351		IPD: R/W	IPD: ND
17352	SQL_DESC_SCALE	ARD: R/W	ARD: ND
17353	(SQLSMALLINT)	APD: R/W	APD: ND
17354		IRD: R/O	IRD: D
17355		IPD: R/W	IPD: ND
17356	SQL_DESC_SCHEMA_NAME	ARD: Unused	ARD: Unused
17357	(SQLCHAR)	APD: Unused	APD: Unused
17358		IRD: R/O	IRD: D
17359		IPD: Unused	IPD: Unused
17360	SQL_DESC_SEARCHABLE	ARD: Unused	ARD: Unused
17361	(SQLSMALLINT)	APD: Unused	APD: Unused
17362		IRD: R/O	IRD: D
17363		IPD: Unused	IPD: Unused
17364	SQL_DESC_TABLE_NAME	ARD: Unused	ARD: Unused
17365	(SQLCHAR)	APD: Unused	APD: Unused
17366		IRD: R/O	IRD: D
17367		IPD: Unused	IPD: Unused
17368	SQL_DESC_TYPE	ARD: R/W	ARD: SQL_C_DEFAULT
17369	(SQLSMALLINT)	APD: R/W	APD: SQL_C_DEFAULT
17370		IRD: R/O	IRD: D
17371		IPD: R/W	IPD: ND
17372	SQL_DESC_TYPE_NAME	ARD: Unused	ARD: Unused
17373	(SQLCHAR)	APD: Unused	APD: Unused
17374		IRD: R/O	IRD: D
17375		IPD: R/O	IPD: D [1]
17376	SQL_DESC_UNNAMED	ARD: Unused	ARD: ND
17377	(SQLSMALLINT)	APD: Unused	APD: ND
17378		IRD: R/O	IRD: D
17379		IPD: R/W	IPD: ND
17380	SQL_DESC_UNSIGNED	ARD: Unused	ARD: Unused
17381	(SQLSMALLINT)	APD: Unused	APD: Unused
17382		IRD: R/O	IRD: D
17383		IPD: R/O	IPD: D [1]
17384	SQL_DESC_UPDATABLE	ARD: Unused	ARD: Unused
17385	(SQLSMALLINT)	APD: Unused	APD: Unused
17386		IRD: R/O	IRD: D
17387		IPD: Unused	IPD: Unused

17388 [1] These fields are defined only when the implementation automatically populates the IPD. If
17389 it does not, they are undefined. If an application tries to set these fields, the implementation
17390 returns SQLSTATEHY091 (Invalid descriptor field identifier).

17391 2 The SQL_DESC_DATA_PTR field in the IPD can be set to force a consistency check.
17392 Subsequent calls to *SQLGetDescField*() or *SQLGetDescRec*(), need not return the value
17393 provided for SQL_DESC_DATA_PTR.

17394 **FieldIdentifier Argument**

17395 *FieldIdentifier* indicates the descriptor field to be set. A descriptor contains the descriptor header,
17396 consisting of the header fields described in the next section, and zero or more descriptor records,
17397 consisting of the record fields described in the following section.

17398 **Fields of the Descriptor Header**

17399 Each descriptor has a header consisting of the following fields.

17400 SQL_DESC_ALLOC_TYPE [All]
17401 This read-only SQLSMALLINT header field specifies whether the descriptor was allocated
17402 automatically by the implementation or explicitly by the application. The application can
17403 obtain, but not modify, this field. The implementation sets this field to
17404 SQL_DESC_ALLOC_AUTO in descriptors it automatically allocates, and to
17405 SQL_DESC_ALLOC_USER in descriptors explicitly allocated by the application.

17406 SQL_DESC_ARRAY_SIZE
17407 In ARDs, this SQLUINTEGER header field specifies the number of rows in the row-set. This
17408 is the number of rows to be returned by a call to *SQLFetch*() or *SQLFetchScroll*(), or operated
17409 on by a call to *SQLBulkOperations*() or *SQLSetPos*(). The default value is 1. The field is also
17410 set through the SQL_ATTR_ROW_ARRAY_SIZE statement attribute.

17411 In APDs, this SQLUINTEGER header field specifies the number of values for each
17412 parameter. This field is set to 1 by default. The field is also set through the
17413 SQL_ATTR_PARAMSET_SIZE statement attribute.

17414 If SQL_DESC_ARRAY_SIZE is greater than 1, SQL_DESC_DATA_PTR,
17415 SQL_DESC_INDICATOR_PTR, and SQL_DESC_OCTET_LENGTH_PTR of the APD or
17416 ARD point to arrays. The cardinality of each array is equal to the value of this field.

17417 This field in the ARD can also be set by calling *SQLSetStmtAttr*() with the
17418 SQL_ATTR_ROW_ARRAY_SIZE attribute.

17419 SQL_DESC_ARRAY_STATUS_PTR [All]
17420 In the IRD, this SQLUSMALLINT * header field points to an array of SQLUSMALLINT
17421 values containing row status values after a call to *SQLBulkOperations*(), *SQLFetch*(),
17422 *SQLFetchScroll*(), or *SQLSetPos*(). The array has as many elements as there are rows in the
17423 row-set. The application must allocate an array of SQLUSMALLINTs and set this field to
17424 point to the array. The field is set to a null pointer by default. The implementation populates
17425 the array, unless the SQL_DESC_ARRAY_STATUS_PTR field is set to a null pointer, in
17426 which case no status values are generated and the array is not populated.

17427 **Caution:** The effect is undefined if the application sets the elements of the row status array
17428 pointed to by the SQL_DESC_ARRAY_STATUS_PTR of the IRD.

17429 The array is initially populated by a call to *SQLBulkOperations*(), *SQLFetch*(), or
17430 *SQLFetchScroll*(). If such a call did not return SQL_SUCCESS or
17431 SQL_SUCCESS_WITH_INFO, the contents of the array pointed to by this field are
17432 undefined. The elements in the array can contain the following values:

17433 SQL_ROW_SUCCESS
17434 The row was successfully fetched and has not changed since it was last fetched.

17435 SQL_ROW_SUCCESS_WITH_INFO
17436 The row was successfully fetched and has not changed since it was last fetched.
17437 However, a warning was returned about the row.

17438 SQL_ROW_ERROR
17439 An error occurred while fetching the row.

17440 SQL_ROW_UPDATED
17441 The row was successfully fetched and has been updated since it was last fetched. If the
17442 row is fetched again, its status is SQL_ROW_SUCCESS.

17443 SQL_ROW_DELETED
17444 The row has been deleted since it was last fetched.

17445 SQL_ROW_ADDED
17446 The row was inserted by *SQLBulkOperations*(). If the row is fetched again, its status is |
17447 SQL_ROW_SUCCESS.

17448 SQL_ROW_NOROW |
17449 The row-set overlapped the end of the result set and no row was returned that
17450 corresponded to this element of the row status array.

17451 This field in the ARD can also be set by calling *SQLSetStmtAttr*() with the
17452 SQL_ATTR_ROW_STATUS_PTRattribute.

17453 In the ARD, this SQLUSMALLINT * header field points to an array of SQLUSMALLINT
17454 values that can be set by the application to indicate whether this row is to be ignored for |
17455 *SQLBulkOperations*() and *SQLSetPos*() operations. The elements in the array can contain the
17456 following values:

17457 SQL_ROW_PROCEED
17458 The row is included in the bulk operation using *SQLBulkOperations*() or *SQLSetPos*(). |
17459 (This setting does not guarantee that the operation will occur on the row. If the row has |
17460 the status SQL_ROW_ERROR in the IRD row status array, the implementation might |
17461 not be able to perform the operation in the row.)

17462 SQL_ROW_IGNORE
17463 The row is excluded from the bulk operation using *SQLBulkOperations*() or |
17464 *SQLSetPos*().

17465 If no elements of the array are set, all rows are included in the bulk operation. If the value in
17466 the SQL_DESC_ARRAY_STATUS_PTR field of the ARD is a null pointer, all rows are
17467 included in the bulk operation, as though it pointed to a valid array all of whose elements
17468 were SQL_ROW_PROCEED. If an element in the array is set to SQL_ROW_IGNORE, the |
17469 value in the row status array for the ignored row is not changed.

17470 In the IPD, this SQLUSMALLINT * header field points to an array of SQLUSMALLINT
17471 values containing status information for each row of parameter values after a call to
17472 *SQLExecute*() or *SQLExecDirect*(). If the call to *SQLExecute*() or *SQLExecDirect*() did not
17473 return SQL_SUCCESS or SQL_SUCCESS_WITH_INFO, the contents of the array pointed to
17474 by this field are undefined. The application must allocate an array of SQLUSMALLINTs and
17475 set this field to point to the array. The implementation will populate the array, unless the |
17476 SQL_DESC_ARRAY_STATUS_PTR field is set to a null pointer, in which case no status
17477 values are generated and the array is not populated. The elements in the array can contain
17478 the following values:

17479 SQL_PARAM_SUCCESS
17480 The SQL statement was successfully executed for this set of parameters.

17481 SQL_PARAM_SUCCESS_WITH_INFO
17482 The SQL statement was successfully executed for this set of parameters; however,
17483 warning information is available in the diagnostics data structure.

17484 SQL_PARAM_ERROR
17485 An error occurred in processing this set of parameters. Additional error information is
17486 available in the diagnostics data structure.

17487 SQL_PARAM_UNUSED
17488 This parameter set was unused, possibly because a previous parameter set caused an
17489 error that aborted further processing, or because SQL_PARAM_IGNORE was set for
17490 that set of parameters in the array specified by the SQL_DESC_ARRAY_STATUS_PTR
17491 field of the APD.

17492 SQL_PARAM_DIAG_UNAVAILABLE
17493 Diagnostic information is not available. An example of this is when a data source treats
17494 arrays of parameters as a monolithic unit and so does not generate this level of error
17495 information.

17496 This field in the APD can also be set by calling *SQLSetStmtAttr*() with the
17497 SQL_ATTR_PARAM_STATUS_PTRattribute.

17498 In the APD, this SQLUSMALLINT * header field points to an array of SQLUSMALLINT
17499 values that can be set by the application to indicate whether this set of parameters is to be
17500 ignored when *SQLExecute*() or *SQLExecDirect*() is called. The elements in the array can
17501 contain the following values:

17502 SQL_PARAM_PROCEED
17503 The set of parameters is included in the *SQLExecute*() or *SQLExecDirect*() call.

17504 SQL_PARAM_IGNORE
17505 The set of parameters is excluded from the *SQLExecute*() or *SQLExecDirect*() call.

17506 If no elements of the array are set, all sets of parameters in the array are used in the
17507 *SQLExecute*() or *SQLExecDirect*() calls. If the value in the SQL_DESC_ARRAY_STATUS_PTR
17508 field of the APD is a null pointer, all sets of parameters are used, as though it pointed to a
17509 valid array all of whose elements were SQL_PARAM_PROCEED.

17510 SQL_DESC_BIND_OFFSET_PTR [Application descriptors]
17511 This SQLINTEGER * header field points to the bind offset. It is set to a null pointer by
17512 default. If this field is not a null pointer, the bind offset is added to each deferred field in the
17513 descriptor record (SQL_DESC_DATA_PTR, SQL_DESC_INDICATOR_PTR, and
17514 SQL_DESC_OCTET_LENGTH_PTR) to produce the effective address for the fetch. The
17515 bind offset is not cumulative; if the value is changed, any old bind offset ceases to have
17516 effect. A bind offset can be used only with row-wise binding. See Section 9.4 on page 102
17517 and **Bind Offsets** on page 217.

17518 This field is a *deferred field*: it is not used at the time it is set, but the implementation uses it
17519 later to retrieve data.

17520 This field in the ARD can also be set by calling *SQLSetStmtAttr*() with the
17521 SQL_ATTR_ROW_BIND_OFFSET_PTR attribute. This field in the ARD can also be set by
17522 calling *SQLSetStmtAttr*() with the SQL_ATTR_PARAM_BIND_OFFSET_PTR attribute.

17523 SQL_DESC_BIND_TYPE [Application descriptors]
17524 This SQLINTEGER header field sets the binding orientation to be used for either binding
17525 columns or parameters.

17526 In ARDs, this field specifies the binding orientation when *SQLFetchScroll*() is called on the
17527 associated statement handle.

17528 To select column-wise binding for columns, this field is set to SQL_BIND_BY_COLUMN
17529 (the default).

17530 This field in the ARD can also be set by calling *SQLSetStmtAttr*() with the
17531 SQL_ATTR_ROW_BIND_TYPE attribute.

17532 In APDs, this field specifies the binding orientation to be used for dynamic parameters.

17533 To select column-wise binding for parameters, this field is set to SQL_BIND_BY_COLUMN
17534 (the default).

17535 This field in the APD can also be set by calling *SQLSetStmtAttr*() with the |
17536 SQL_ATTR_PARAM_BIND_TYPE attribute.

17537 SQL_DESC_COUNT [All]
17538 This SQLSMALLINT header field specifies the one-based index of the highest-numbered |
17539 record that contains data. When the implementation sets the data structure for the
17540 descriptor, it must also set the SQL_DESC_COUNT field to show how many records are |
17541 significant. When an application allocates an instance of this data structure, it does not have
17542 to specify how many records to reserve room for. As the application specifies the contents |
17543 of the records, the implementation takes any required action to ensure that the descriptor
17544 handle refers to a data structure of the adequate size.

17545 SQL_DESC_COUNT is not a count of all records that are used, but the number of the |
17546 highest-numbered bound record. If the application unbinds the record with this number, the |
17547 implementation implicitly resets SQL_DESC_COUNT to the highest-numbered bound |
17548 record remaining. If the result is that there are no more bound records (or if the application |
17549 calls *SQLFreeStmt*() with the SQL_UNBIND option to achieve this explicitly), then the |
17550 implementation sets SQL_DESC_COUNT to 0. If the application binds additional records |
17551 with numbers greater than the highest-numbered bound record, the implementation |
17552 increases the SQL_DESC_COUNT field to this record number.

17553 The value in SQL_DESC_COUNT can be set explicitly by an application by calling
17554 *SQLSetDescField*(). If the value in SQL_DESC_COUNT is explicitly decreased, all records
17555 with numbers greater than the new value in SQL_DESC_COUNT are removed, unbinding
17556 the columns. If the value in SQL_DESC_COUNT is explicitly set to 0, and the field is in an
17557 APD, all parameter columns are unbound. If the value in SQL_DESC_COUNT is explicitly
17558 set to 0, and the field is in an ARD, all data buffers except a bound bookmark column are
17559 released.

17560 The record count in this field of an ARD does not include a bound bookmark column. To |
17561 unbind a bookmark column, the application sets the DATA_PTR field of record number 0 to |
17562 a null pointer.

17563 SQL_DESC_ROWS_PROCESSED_PTR [Implementation descriptors]
17564 In an IRD, this SQLUINTEGER * header field points to a buffer containing the number of
17565 rows fetched after a call to *SQLFetch*() or *SQLFetchScroll*(), or the number of rows affected in
17566 a bulk operation performed by a call to *SQLBulkOperations*() or *SQLSetPos*().

17567 In an IPD, this SQLUINTEGER * header field points to a buffer containing the number of |
17568 sets of parameters that have been processed, including error rows. No row number is |
17569 returned if this is a null pointer.

17570 SQL_DESC_ROWS_PROCESSED_PTR is valid only after SQL_SUCCESS or
17571 SQL_SUCCESS_WITH_INFO has been returned after a call to *SQLFetch*() or
17572 *SQLFetchScroll*() (for an IRD field) or *SQLExecute*() or *SQLExecDirect*() (for an IPD field). If |
17573 the return code is not one of these, the location pointed to by
17574 SQL_DESC_ROWS_PROCESSED_PTR is undefined.

17575 If the call to *SQLExecDirect*(), *SQLExecute*(), *SQLFetch*(), *SQLFetchScroll*(), or |
17576 *SQLParamData*() that fills in the buffer pointed to by this field did not return SQL_SUCCESS
17577 or SQL_SUCCESS_WITH_INFO, the contents of the buffer are undefined.

17578 This field in the ARD can also be set by calling *SQLSetStmtAttr*() with the
17579 SQL_ATTR_ROWS_FETCHED_PTR attribute. This field in the ARD can also be set by

17580 calling *SQLSetStmtAttr*() with the SQL_ATTR_PARAMS_PROCESSED_PTR attribute.

17581 The buffer pointed to by this field is allocated by the application. It is a deferred output
17582 buffer that the implementation sets. It is set to a null pointer by default.

17583 **Fields of Each Descriptor Record**

17584 Each descriptor contains one or more records consisting of fields that define either column data
17585 or dynamic parameters, depending on the type of descriptor. Each record is a complete
17586 definition of a single column or parameter.

17587 SQL_DESC_AUTO_UNIQUE_VALUE [IRDs]
17588 This read-only SQLINTEGER record field contains SQL_TRUE if the column is an auto-
17589 incrementing column, or SQL_FALSE if the column is not an auto-incrementing column.
17590 This field is read-only, but the underlying auto-incrementing column is not necessarily
17591 read-only.

17592 An application can insert values into a row containing an autoincrement column, but
17593 typically cannot update values in the column. When an insert is made into an auto-
17594 increment column, a unique value is inserted into the column at insert time. The increment
17595 is not defined, but is data-source-specific. An application should not assume that an auto-
17596 increment column starts at any particular point or increments by any particular value.

17597 SQL_DESC_BASE_COLUMN_NAME [IRDs]
17598 This read-only SQLCHAR record field contains the base column name for the result set
17599 column. If a base column name does not exist (as in the case of columns that are
17600 expressions), then this variable contains an empty string.

17601 SQL_DESC_BASE_TABLE_NAME [IRDs]
17602 This read-only SQLCHAR record field contains the base table name for the result set
17603 column. If a base table name cannot be defined or is not applicable, then this variable
17604 contains an empty string.

17605 SQL_DESC_CASE_SENSITIVE [Implementation descriptors]
17606 This read-only SQLINTEGER record field contains SQL_TRUE if the column or parameter is
17607 treated as case-sensitive for collations and comparisons, or SQL_FALSE if the column is not
17608 treated as case-sensitive for collations and comparisons, or if it is a non-character column.

17609 SQL_DESC_CATALOG_NAME [IRDs]
17610 This read-only SQLCHAR record field contains the catalog name for the base table that
17611 contains the column. The return value is implementation-defined if the column is an
17612 expression or if the column is part of a view. If the data source does not support catalogs or
17613 the catalog name cannot be determined, this variable contains an empty string.

17614 SQL_DESC_CONCISE_TYPE [All]
17615 This SQLSMALLINT header field specifies the concise data type for all data types, including
17616 the date/time and interval data types.

17617 The values in the SQL_DESC_CONCISE_TYPE and SQL_DESC_TYPE fields are
17618 interdependent. Each time one of the fields is set, the other must also be set.
17619 SQL_DESC_CONCISE_TYPE can be set by a call to *SQLBindCol*() or *SQLBindParameter*(), or
17620 *SQLSetDescField*(). SQL_DESC_TYPE can be set by a call to *SQLSetDescField*() or
17621 *SQLSetDescRec*().

17622 If SQL_DESC_CONCISE_TYPE is set to a concise data type other than an interval or
17623 date/time data type, the SQL_DESC_TYPE field is set to the same value, and the
17624 SQL_DESC_DATETIME_INTERVAL_CODE field is set to 0.

17625 If SQL_DESC_CONCISE_TYPE is set to the concise date/time or interval data type, the
17626 SQL_DESC_TYPE field is set to the corresponding verbose type (SQL_DATETIME or
17627 SQL_INTERVAL), and the SQL_DESC_DATETIME_INTERVAL_CODE field is set to the
17628 appropriate subcode.

17629 SQL_DESC_DATA_PTR [Application descriptors and IPDs]
17630 This SQLPOINTER record field points to a variable that will contain the parameter value
17631 (for APDs) or the column value (for ARDs). This field is a deferred field: it is not used at the
17632 time it is set, but the implementation uses it later to retrieve data.

17633 The column specified by the SQL_DESC_DATA_PTR field of the ARD is unbound if
17634 *TargetValuePtr* in a call to *SQLBindCol*() is a null pointer, or the SQL_DESC_DATA_PTR field
17635 in the ARD is set by a call to *SQLSetDescField*() or *SQLSetDescRec*() to a null pointer. Other
17636 fields are not affected if the SQL_DESC_DATA_PTR field is set to a null pointer.

17637 If the call to *SQLFetch*() or *SQLFetchScroll*() that fills in the buffer pointed to by this field did
17638 not return SQL_SUCCESS or SQL_SUCCESS_WITH_INFO, the contents of the buffer are
17639 undefined.

17640 Whenever the SQL_DESC_DATA_PTR field of an APD, ARD, or IPD is set, a consistency
17641 check occurs; see **Consistency Checks** on page 486. Causing this check is the only use of
17642 this field in an IPD.

17643 SQL_DESC_DATETIME_INTERVAL_CODE [All]
17644 This SQLSMALLINT record field contains the subcode for the specific date/time or interval
17645 data type when the SQL_DESC_TYPE field is SQL_DATETIME or SQL_INTERVAL. This is
17646 true for both SQL and C data types.

17647 For date/time data types, this field can be set to the following:

17648	**Datetime types**	**DATETIME_INTERVAL_CODE**
17649	SQL_TYPE_DATE/SQL_C_TYPE_DATE	SQL_CODE_DATE
17650	SQL_TYPE_TIME/SQL_C_TYPE_TIME	SQL_CODE_TIME
17651	SQL_TYPE_TIMESTAMP/SQL_C_TYPE_TIMESTAMP	SQL_CODE_TIMESTAMP

17652 For interval data types, for all the SQL data types whose name is of the form
17653 SQL_INTERVAL_*suffix*, and for the corresponding C data types whose name is of the form
17654 SQL_C_INTERVAL_*suffix*, there is a subcode of the form SQL_CODE_*suffix*, which can be
17655 used to set this field. (See also Section D.4 on page 569.)

17656 SQL_DESC_DATETIME_INTERVAL_PRECISION [All]
17657 This SQLINTEGER record field contains the interval leading precision if the
17658 SQL_DESC_TYPE field is SQL_INTERVAL. When the
17659 SQL_DESC_DATETIME_INTERVAL_CODE field is set to an interval data type, this field is
17660 set to the default interval leading precision.

17661 SQL_DESC_DISPLAY_SIZE [IRDs]
17662 This read-only SQLINTEGER record field contains the maximum number of characters
17663 required to display the data from the column. The value in this field is not the same as the
17664 descriptor field SQL_DESC_LENGTH because the SQL_DESC_LENGTH field is undefined
17665 for all numeric types.

17666 SQL_DESC_FIXED_PREC_SCALE [Implementation descriptors]
17667 This read-only SQLSMALLINT record field is set to SQL_TRUE if the column is an exact
17668 numeric column and has a fixed precision and non-zero scale, or SQL_FALSE if the column
17669 is not an exact numeric column with a fixed precision and scale.

17670 SQL_DESC_INDICATOR_PTR [Application descriptors]
17671 In ARDs, this SQLINTEGER * record field points to the indicator variable. This variable
17672 contains SQL_NULL_DATA if the column value is a NULL. For APDs, the indicator variable
17673 is set to SQL_NULL_DATA to specify NULL dynamic arguments. Otherwise, the variable is
17674 zero (unless the values in SQL_DESC_INDICATOR_PTR and
17675 SQL_DESC_OCTET_LENGTH_PTR are the same pointer).

17676 If the SQL_DESC_INDICATOR_PTR field in an ARD is a null pointer, the implementation is |
17677 prevented from returning information about whether the column is NULL or not. If the |
17678 column is NULL and SQL_DESC_INDICATOR_PTR is a null pointer, SQLSTATE 22002 |
17679 (Indicator variable required but not supplied) is returned when the implementation tries to |
17680 populate the buffer after a call to *SQLFetch*() or *SQLFetchScroll*(). If the call to *SQLFetch*() or
17681 *SQLFetchScroll*() did not return SQL_SUCCESS or SQL_SUCCESS_WITH_INFO, the
17682 contents of the buffer are undefined.

17683 The SQL_DESC_INDICATOR_PTR field determines whether the field pointed to by
17684 SQL_DESC_OCTET_LENGTH_PTR is set. If the data value for a column is NULL, the |
17685 implementation sets the indicator variable to SQL_NULL_DATA. The field pointed to by
17686 SQL_DESC_OCTET_LENGTH_PTR is then not set. If a NULL value is not encountered
17687 during the fetch, the buffer pointed to by SQL_DESC_INDICATOR_PTR is set to zero, and
17688 the buffer pointed to by SQL_DESC_OCTET_LENGTH_PTR is set to the length of the data.

17689 If the SQL_DESC_INDICATOR_PTR field in an APD is a null pointer, the application |
17690 cannot use this descriptor record to specify NULL arguments. |

17691 This field is a deferred field: it is not used at the time it is set, but the implementation uses it |
17692 later to store data.

17693 SQL_DESC_LABEL [IRDs]
17694 This read-only SQLCHAR record field contains the column label or title. If the column does
17695 not have a label, this variable contains the column name. If the column is unnamed and
17696 unlabeled, this variable contains an empty string.

17697 SQL_DESC_LENGTH [All]
17698 This SQLUINTEGER record field is either the maximum or actual character length of a
17699 character string or a binary data type. It is the maximum character length for a fixed-length
17700 data type, or the actual character length for a variable-length data type. Its value always |
17701 excludes the null terminator that ends the character string. For date/time and interval data |
17702 types, this field has the length in characters of the character-string representation of the |
17703 value. This field is a count of characters, not octets.

17704 SQL_DESC_LITERAL_PREFIX [IRDs]
17705 This read-only SQLCHAR record field contains the character or characters that the data |
17706 source recognizes as a prefix for a literal of this data type. This variable contains an empty
17707 string for a data type for which a literal prefix is not applicable.

17708 SQL_DESC_LITERAL_SUFFIX [IRDs]
17709 This read-only SQLCHAR record field contains the character or characters that the data |
17710 source recognizes as a suffix for a literal of this data type. This variable contains an empty
17711 string for a data type for which a literal suffix is not applicable.

17712 SQL_DESC_LOCAL_TYPE_NAME [Implementation descriptors]
17713 This read-only SQLCHAR record field contains any localized (native language) name for
17714 the data type that may be different from the regular name of the data type. If there is no
17715 localized name, then an empty string is returned. This field is for display purposes only. The
17716 character set of the string is locale-dependent and is typically the default character set of the
17717 data source.

17718 SQL_DESC_NAME [Implementation descriptors]
17719 In a row descriptor, this field contains the column name, or any applicable column alias. If
17720 there is no column name or column alias, this field contains an empty string (and the
17721 SQL_DESC_UNNAMED field contains SQL_UNNAMED).

17722 An application can set the SQL_DESC_NAME field of an IPD to a parameter name or alias.
17723 The SQL_DESC_NAME field of an IRD is a read-only field; SQLSTATE HY091 (Invalid
17724 descriptor field identifier) is returned if an application tries to set it. If an application sets
17725 the SQL_DESC_UNNAMED field of an IPD to SQL_UNNAMED, the SQL_DESC_NAME
17726 field of the IPD is set to NULL.

17727 In IPDs, this field contains the parameter name if the data source supports named
17728 parameters and is capable of describing parameters. Otherwise, this field is undefined.

17729 SQL_DESC_NULLABLE [Implementation descriptors]
17730 In IRDs, this read-only SQLSMALLINT record field is SQL_NULLABLE if the column can
17731 have NULL values; SQL_NO_NULLS if the column does not have NULL values; or
17732 SQL_NULLABLE_UNKNOWN if it is not known whether the column accepts NULL
17733 values. This field pertains to the result set column, not the base column.

17734 In IPDs, this field is always set to SQL_NULLABLE, since dynamic parameters are always
17735 nullable, and cannot be set by an application.

17736 SQL_DESC_OCTET_LENGTH [All]
17737 This SQLINTEGER record field contains the length, in octets, of a character string or binary
17738 data type. For fixed-length character types, this is the actual length in octets. For variable-
17739 length character or binary types, this is the maximum length in octets. This value always
17740 excludes space for the null terminator for implementation descriptors and always includes
17741 space for the null terminator for application descriptors. For application data, this field
17742 contains the size of the buffer. For APDs, this field is defined only for output or
17743 input/output parameters.

17744 SQL_DESC_OCTET_LENGTH_PTR [Application descriptors]
17745 This SQLINTEGER * record field points to a variable that will contain the total length in
17746 octets of a dynamic argument (for parameter descriptors) or of a bound column value (for
17747 row descriptors).

17748 For an APD, this value is ignored for all arguments except character string and binary; if this
17749 field points to SQL_NTS, the dynamic argument must be null-terminated. To indicate that a
17750 bound parameter is a data-at-execute parameter, an application sets this field in the
17751 appropriate record of the APD to a variable that, at execute time, will contain the value
17752 SQL_DATA_AT_EXEC. If there is more than one such field, SQL_DESC_DATA_PTR can be
17753 set to a value uniquely identifying the parameter to help the application determine which
17754 parameter is being requested.

17755 If the OCTET_LENGTH_PTR field of an ARD is a null pointer, the implementation does not
17756 return length information for the column. If the SQL_DESC_OCTET_LENGTH_PTR field
17757 of an APD is a null pointer, the implementation assumes that character strings and binary
17758 values are null-terminated. (Binary values should not be null-terminated, but should be
17759 given a length, in order to avoid truncation.)

17760 If the call to *SQLFetch*() or *SQLFetchScroll*() that fills in the buffer pointed to by this field did
17761 not return SQL_SUCCESS or SQL_SUCCESS_WITH_INFO, the contents of the buffer are
17762 undefined.

17763 This field is a deferred field: it is not used at the time it is set, but the implementation uses it
17764 later to buffer data.

17765 SQL_DESC_PARAMETER_TYPE [IPDs]
17766 This SQLSMALLINT record field is set to SQL_PARAM_INPUT for an input parameter,
17767 SQL_PARAM_INPUT_OUTPUT for an input/output parameter, or
17768 SQL_PARAM_OUTPUT for an output parameter. Set to SQL_PARAM_INPUT by default.

17769 For an IPD, the field is set to SQL_PARAM_INPUT by default if the implementation does |
17770 not automatically populate the IPD (if the SQL_ATTR_ENABLE_AUTO_IPD statement |
17771 attribute is SQL_FALSE). An application should set this field in the IPD for parameters that |
17772 are not input parameters.

17773 SQL_DESC_PRECISION [All]
17774 This SQLSMALLINT record field contains the precision for a numeric data type. For data
17775 types time, timestamp, and all the interval data types that represent a time interval, this
17776 field contains the precision of the fractional seconds component.

17777 SQL_DESC_SCALE [All]
17778 This SQLSMALLINT record field contains the defined scale for DECIMAL and NUMERIC
17779 data types. The field is undefined for all other data types.

17780 SQL_DESC_SCHEMA_NAME [IRDs]
17781 This read-only SQLCHAR record field contains the schema name of the base table that |
17782 contains the column. This is implementation-defined if the column is an expression or if the
17783 column is part of a view. If the data source does not support schemas or the schema name |
17784 cannot be determined, this variable contains an empty string.

17785 SQL_DESC_SEARCHABLE [IRDs]
17786 This read-only SQLSMALLINT record field is set to one of the following values:

17787	SQL_PRED_NONE	The column cannot be used in a WHERE clause.
17788 17789	SQL_PRED_CHAR	The column can be used in a WHERE clause, but only with the LIKE predicate.
17790 17791	SQL_PRED_BASIC	The column can be used in a WHERE clause with all the comparison operators except LIKE.
17792 17793	SQL_PRED_SEARCHABLE	The column can be used in a WHERE clause with any comparison operator.

17794 For data of type SQL_LONGVARCHAR and SQL_LONGVARBINARY, the value |
17795 SQL_PRED_CHAR is typical.

17796 SQL_DESC_TABLE_NAME [IRDs]
17797 This read-only SQLCHAR record field contains the name of the base table that contains this
17798 column. The value is undefined if the column is an expression or part of a view.

17799 SQL_DESC_TYPE [All]
17800 This SQLSMALLINT record field specifies the concise SQL or C data type for all data types |
17801 except date/time and interval data types. For the date/time and interval data types, this
17802 field specifies the verbose data type, SQL_DATETIME or SQL_INTERVAL. (For an |
17803 overview of verbose versus concise identifiers, see **Data Type Identification in Descriptors** |
17804 on page 574.)

17805 Whenever this field contains SQL_DATETIME or SQL_INTERVAL, the
17806 SQL_DESC_DATETIME_INTERVAL_CODE field must contain the appropriate subcode for |
17807 the concise type. For date/time data types, SQL_DESC_TYPE contains SQL_DATETIME,
17808 and the SQL_DESC_DATETIME_INTERVAL_CODE field contains a subcode for the |
17809 specific date/time data type. For interval data types, SQL_DESC_TYPE contains
17810 SQL_INTERVAL, and the SQL_DESC_DATETIME_INTERVAL_CODE field contains a
17811 subcode for the specific interval data type.

17812 The values in the SQL_DESC_TYPE and SQL_DESC_CONCISE_TYPE fields are
17813 interdependent. Each time one of the fields is set, the other must also be set.
17814 SQL_DESC_TYPE can be set by a call to *SQLSetDescField*() or *SQLSetDescRec*().
17815 SQL_DESC_CONCISE_TYPE can be set by a call to *SQLBindCol*() or *SQLBindParameter*(), or
17816 *SQLSetDescField*().

17817 If SQL_DESC_TYPE is set to a concise data type other than an interval or date/time data
17818 type, the SQL_DESC_CONCISE_TYPE field is set to the same value, and the
17819 SQL_DESC_DATETIME_INTERVAL_CODEfield is set to 0.

17820 If SQL_DESC_TYPE is set to the verbose date/time or interval data type, (namely,
17821 SQL_DATETIME or SQL_INTERVAL), and the SQL_DESC_DATETIME_INTERVAL_CODE
17822 field is set to the appropriate subcode, then the SQL_DESC_CONCISE TYPE field is set to
17823 the corresponding concise type. Setting SQL_DESC_TYPE to one of the concise date/time
17824 or interval types returns SQLSTATE HY021 (Inconsistent descriptor information).

17825 **Default Values for Certain Data Types**

17826 When the SQL_DESC_TYPE field is set by a call to *SQLSetDescField*(), the following fields
17827 are set to the following default values. The values of the remaining fields of the same
17828 record are undefined:

	Value of SQL_DESC_TYPE	Other fields implicitly set
17829–17831	SQL_CHAR, SQL_VARCHAR, SQL_C_CHAR, SQL_C_VARCHAR	SQL_DESC_LENGTH is set to 1. SQL_DESC_PRECISION is set to 0.
17832–17836	SQL_DATETIME	When SQL_DESC_DATETIME_INTERVAL_CODE is set to SQL_CODE_DATE or SQL_CODE_TIME, SQL_DESC_PRECISION is set to 0. When it is set to SQL_DESC_TIMESTAMP, SQL_DESC_PRECISION is set to 6.
17837–17840	SQL_DECIMAL, SQL_NUMERIC SQL_C_NUMERIC	SQL_DESC_SCALE is set to 0. SQL_DESC_PRECISION is set to the implementation-defined precision for the respective data type.
17841–17843	SQL_FLOAT,SQL_C_FLOAT	SQL_DESC_PRECISION is set to the implementation-defined default precision for SQL_FLOAT.
17844–17850	SQL_INTERVAL	When SQL_DESC_DATETIME_INTERVAL_CODE is set to an interval data type, SQL_DESC_DATETIME_INTERVAL_PRECISION is set to 2 (the default interval leading precision). When the interval has a seconds component, SQL_DESC_PRECISION is set to 6 (the default interval seconds precision).

17851 When an application calls *SQLSetDescField*() to set fields of a descriptor, rather than calling
17852 *SQLSetDescRec*(), the application must first declare the data type. When it does, the other
17853 fields indicated in the table above are implicitly set. If any of the values implicitly set are
17854 unacceptable, the application can then call *SQLSetDescField*() or *SQLSetDescRec*() to set the
17855 unacceptable value explicitly.

17856 SQL_DESC_TYPE_NAME [Implementation descriptors]
17857 This read-only SQLCHAR record field contains the data-source-dependent type name (for
17858 example, CHAR, VARCHAR, and so on). If the data type name is unknown, this variable
17859 contains an empty string.

17860 SQL_DESC_UNNAMED [Implementation descriptors]
17861 This SQLSMALLINT record field in a row descriptor is set to either SQL_NAMED or
17862 SQL_UNNAMED when the SQL_DESC_NAME field is set. If the SQL_DESC_NAME field
17863 contains a column alias, or if the column alias does not apply, the UNNAMED field is set to
17864 SQL_NAMED. If an application sets the SQL_DESC_NAME field of an IPD to a parameter
17865 name or alias, the driver sets the SQL_DESC_UNNAMED field of the IPD to SQL_NAMED.
17866 If there is no column name or a column alias, the UNNAMED field is set to
17867 SQL_UNNAMED.

17868 An application can set the SQL_DESC_UNNAMED field of an IPD to SQL_UNNAMED, in
17869 which case the implementation sets the SQL_DESC_NAME field of the IPD to NULL. The
17870 implementation returns SQLSTATE HY091 (Invalid descriptor field identifier) if an
17871 application tries to set the SQL_DESC_UNNAMED field of an IPD to SQL_NAMED. The
17872 SQL_DESC_UNNAMED field of an IRD is read-only; SQLSTATE HY091 (Invalid descriptor
17873 field identifier) is returned if an application tries to set it.

17874 SQL_DESC_UNSIGNED [Implementation descriptors]
17875 This read-only SQLSMALLINT record field is set to SQL_TRUE if the column type is
17876 unsigned or non-numeric, or SQL_FALSE if the column type is signed.

17877 SQL_DESC_UPDATABLE[IRDs]
17878 This read-only SQLSMALLINT record field is set to one of the following values:

17879 SQL_ATTR_READONLY The result set column is read-only.

17880 SQL_ATTR_WRITE The result set column is read-write.

17881 SQL_ATTR_READWRITE_UNKNOWN
17882 It is not known whether the result set column is
17883 updatable.

17884 This describes the updatability of the column in the result set, not the column in the base
17885 table, which may be different. Whether a column is updatable can be based on the data
17886 type, user privileges, and the definition of the result set itself.

17887 **SEE ALSO**

17888	For information about	See
17889	Setting multiple descriptor fields	*SQLSetDescRec*()
17890	Getting a descriptor field	*SQLGetDescField*()
17891	Getting multiple descriptor fields	*SQLGetDescRec*()
17892	Binding a column	*SQLBindCol*()
17893	Binding a parameter	*SQLBindParam*()

17894 **CHANGE HISTORY**

17895 **Version 2**
17896 Revised generally. See **Alignment with Popular Implementations** on page 2.

17897 **Descriptor Fields Added in Version 2**

17898 The following descriptor fields are new in this issue:

17899	SQL_DESC_ALLOC_TYPE	SQL_DESC_LABEL
17900	SQL_DESC_ARRAY_SIZE	SQL_DESC_LITERAL_PREFIX
17901	SQL_DESC_ARRAY_STATUS_PTR	SQL_DESC_LITERAL_SUFFIX
17902	SQL_DESC_BIND_OFFSET_PTR	SQL_DESC_LOCAL_TYPE_NAME
17903	SQL_DESC_BIND_TYPE	SQL_DESC_NAME
17904	SQL_DESC_ROWS_PROCESSED_PTR	SQL_DESC_NULLABLE
17905	SQL_DESC_AUTO_UNIQUE_VALUE	SQL_DESC_PARAMETER_TYPE
17906	SQL_DESC_BASE_COLUMN_NAME	SQL_DESC_SCHEMA_NAME
17907	SQL_DESC_BASE_TABLE_NAME	SQL_DESC_SEARCHABLE
17908	SQL_DESC_CASE_SENSITIVE	SQL_DESC_TABLE_NAME
17909	SQL_DESC_CATALOG_NAME	SQL_DESC_TYPE_NAME
17910	SQL_DESC_CONCISE_TYPE	SQL_DESC_UNNAMED
17911	SQL_DESC_DATETIME_INTERVAL_PRECISION	SQL_DESC_UNSIGNED
17912	SQL_DESC_DISPLAY_SIZE	SQL_DESC_UPDATABLE
17913	SQL_DESC_FIXED_PREC_SCALE	

17914 **NAME**

17915 SQLSetDescRec — Set multiple descriptor fields.

17916 **SYNOPSIS**

```
17917 SQLRETURN SQLSetDescRec(
17918   SQLHDESC DescriptorHandle,
17919   SQLSMALLINT RecNumber,
17920   SQLSMALLINT Type,
17921   SQLSMALLINT SubType,
17922   SQLINTEGER Length,
17923   SQLSMALLINT Precision,
17924   SQLSMALLINT Scale,
17925   SQLPOINTER DataPtr,
17926   SQLINTEGER * StringLengthPtr,
17927   SQLINTEGER * IndicatorPtr);
```

17928 **ARGUMENTS**

17929 *DescriptorHandle* [Input] |
17930 Descriptor handle. This must not be an IRD handle. |

17931 *RecNumber* [Input]
17932 Indicates the descriptor record that contains the fields to be set. Descriptor records are
17933 numbered from 0, with record number 0 being the bookmark record. This argument must be
17934 equal to or greater than 0. If *RecNumber* is greater than the value of SQL_DESC_COUNT,
17935 *RecNumber* is changed to the value of SQL_DESC_COUNT. |

17936 *Type* [Input]
17937 The value to which to set the SQL_DESC_TYPE field for the descriptor record. |

17938 *SubType* [Input]
17939 For records whose type is SQL_DATETIME or SQL_INTERVAL, this is the value to which to
17940 set the SQL_DESC_DATETIME_INTERVAL_CODE field. |

17941 *Length* [Input]
17942 The value to which to set the SQL_DESC_OCTET_LENGTH field for the descriptor record. |

17943 *Precision* [Input] |
17944 The value to which to set the SQL_DESC_PRECISION field for the descriptor record. |

17945 *Scale* [Input] |
17946 The value to which to set the SQL_DESC_SCALE field for the descriptor record. |

17947 *DataPtr* [Deferred Input or Output]
17948 The value to which to set the SQL_DESC_DATA_PTR field for the descriptor record.
17949 *DataPtr* can be set to a null pointer to set the SQL_DESC_DATA_PTR field to a null pointer. |
17950 If *DescriptorHandle* refers to an ARD, this unbinds the column. |

17951 *StringLengthPtr* [Deferred Input or Output]
17952 The value to which to set the SQL_DESC_OCTET_LENGTH_PTR field for the descriptor
17953 record. *StringLengthPtr* can be set to a null pointer to set the
17954 SQL_DESC_OCTET_LENGTH_PTR field to a null pointer. |

17955 *IndicatorPtr* [Deferred Input or Output]
17956 The value to which to set the SQL_DESC_INDICATOR_PTR field for the descriptor record.
17957 *IndicatorPtr* can be set to a null pointer to set the SQL_DESC_INDICATOR_PTR field to a
17958 null pointer. |

17959 **RETURN VALUE**

17960 SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.

17961 **DIAGNOSTICS**

17962 When *SQLSetDescRec*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated
17963 SQLSTATE value can be obtained by calling *SQLGetDiagRec*() with a *HandleType* of
17964 SQL_HANDLE_DESC and a *Handle* of *DescriptorHandle*. The following SQLSTATE values are
17965 commonly returned by *SQLSetDescRec*(). The return code associated with each SQLSTATE value
17966 is SQL_ERROR, except that for SQLSTATE values in class 01, the return code is
17967 SQL_SUCCESS_WITH_INFO.

17968 01000 — General warning
17969 Implementation-defined informational message.

17970 07006 — Restricted data type attribute violation
17971 *DescriptorHandle* referred to an application descriptor, *RecNumber* was 0, and *Type* was
17972 SQL_C_VARBOOKMARK.

17973 07009 — Invalid descriptor index
17974 *RecNumber* was set to 0, and *DescriptorHandle* referred to an IPD handle.

17975 *RecNumber* was less than 0.

17976 *RecNumber* was greater than the maximum number of columns or parameters that the data
17977 source supports, and *DescriptorHandle* referred to an APD or an ARD.

17978 08S01 — Communication link failure
17979 The communication link to the data source failed before the function completed processing.

17980 HY000 — General error
17981 An error occurred for which there was no specific SQLSTATE and for which no
17982 implementation-specific SQLSTATE was defined. The error message returned by
17983 *SQLGetDiagRec*() in the **MessageText* buffer describes the error and its cause.

17984 HY001 — Memory allocation error
17985 The implementation failed to allocate memory required to support execution or completion
17986 of the function.

17987 HY010 — Function sequence error
17988 *DescriptorHandle* was associated with a statement handle for which an asynchronously
17989 executing function (not this one) was called and was still executing when this function was
17990 called.

17991 *SQLBulkOperations*(), *SQLExecDirect*(), *SQLExecute*(), or *SQLSetPos*() was called for a
17992 statement handle which *DescriptorHandle* was associated and returned SQL_NEED_DATA.
17993 This function was called before data was sent for all data-at-execution parameters or
17994 columns.

17995 HY016 — Cannot modify an implementation row descriptor
17996 *DescriptorHandle* referred to an IRD.

17997 HY021 — Inconsistent descriptor information
17998 *Type*, or any other field associated with the TYPE field in the descriptor, was not valid or
17999 consistent.

18000 The descriptor consistency check failed (see **Consistency Checks** on page 486).

18001 HY091 — Invalid descriptor field identifier
18002 A field to be set was not defined for *DescriptorHandle.*

18003 HY104 — Invalid precision value
18004 *Length*, *Precision*, or *Scale* was outside the range of values supported by the data source for a
18005 column of the SQL data type specified by *Type* and/or *SubType*.

18006 HYT01 — Connection timeout expired
18007 The connection timeout period expired before the data source responded to the request. The
18008 connection timeout period is set through *SQLSetConnectAttr*(),
18009 SQL_ATTR_CONNECTION_TIMEOUT.

18010 IM001 — Function not supported
18011 The function is not supported on the current connection to the data source.

18012 **COMMENTS**

18013 An application can call *SQLSetDescRec*() to set the following fields for a single column or
18014 parameter:

18015 • SQL_DESC_TYPE
18016 • SQL_DESC_DATETIME_INTERVAL_CODE(for date/times and intervals only)
18017 • SQL_DESC_OCTET_LENGTH
18018 • SQL_DESC_PRECISION
18019 • SQL_DESC_SCALE
18020 • SQL_DESC_DATA_PTR
18021 • SQL_DESC_OCTET_LENGTH_PTR
18022 • SQL_DESC_INDICATOR_PTR

18023 When binding a column or parameter, *SQLSetDescRec*() sets multiple fields affecting the binding
18024 without calling *SQLBindCol*() or *SQLBindParameter*(), or making multiple calls to
18025 *SQLSetDescField*(). *SQLSetDescRec*() can set fields on a descriptor not currently associated with a
18026 statement. (*SQLBindParameter*() sets more fields than *SQLSetDescRec*(), can set fields on both an
18027 APD and an IPD in one call, and does not require a descriptor handle.)

18028 The application should set the statement attribute SQL_ATTR_USE_BOOKMARKS before
18029 calling *SQLSetDescRec*() with a *RecNumber* of 0 to set bookmark fields.

18030 If a call to *SQLSetDescRec*() fails, the contents of the above descriptor fields are undefined.

18031 **Consistency Checks**

18032 The implementation automatically performs a consistency check whenever the application sets
18033 the SQL_DESC_DATA_PTR field of an APD, ARD, or IPD. If any of the fields is inconsistent
18034 with other fields, *SQLSetDescRec*() returns SQLSTATE HY021 (Inconsistent descriptor
18035 information). (There is no check between the value of an ARD and an IPD.)

18036 Whenever an application sets the SQL_DESC_DATA_PTR field of an APD, ARD, or IPD, the
18037 implementation checks that the value of the SQL_DESC_TYPE field and the values applicable to
18038 that SQL_DESC_TYPE field are valid and consistent. This check is always performed when
18039 *SQLBindParameter*() or *SQLBindCol*() is called, or when *SQLSetDescRec*() is called for an APD,
18040 ARD, or IPD. This consistency check includes the following checks on descriptor fields:

18041 • The SQL_DESC_TYPE record field is verified to be one of the valid XDBC C or SQL types, or
18042 an implementation-defined C or SQL type.

18043 • The SQL_DESC_CONCISE_TYPE field is verified to be one of the valid XDBC C or SQL
18044 types or an implementation-defined C or SQL type, including the concise date/time and
18045 interval types.

18046 • If the SQL_DESC_TYPE record field is SQL_DATETIME or SQL_INTERVAL, then the
18047 SQL_DESC_DATETIME_INTERVAL_CODE field is verified to be one of the valid date/time
18048 or interval codes (see the description of the SQL_DESC_DATETIME_INTERVAL_CODE

18049 descriptor field in *SQLSetDescField*()).

18050 • If the SQL_DESC_TYPE of an ARD or APD is SQL_C_NUMERIC, the
18051 SQL_DESC_PRECISION and SQL_DESC_SCALE fields are verified to be valid.

18052 • If the SQL_DESC_CONCISE_TYPE field is a time or timestamp data type, or one of the
18053 interval data types with a seconds component, the SQL_DESC_PRECISION field is verified
18054 to be a valid seconds precision.

18055 • If the SQL_DESC_CONCISE_TYPE field is an interval data type, the
18056 SQL_DESC_DATETIME_INTERVAL_PRECISION field is verified to be a valid interval
18057 leading precision value.

18058 An application can prompt a consistency check by setting the SQL_DESC_DATA_PTR field of an
18059 IPD. An application would set this field only to force the consistency check; it is undefined
18060 whether the value the application provides is stored in the IPD and can be retrieved from the
18061 IPD.

18062 Consistency checks are not performed for IRDs.

18063 A failure of any part of the consistency check causes the XDBC function to return SQLSTATE
18064 HY021 (Inconsistent descriptor information). If the XDBC function was called to set the field to
18065 an inconsistent value, the resulting contents of that descriptor record are undefined. If the field
18066 is set in a record whose number is greater than the value of the SQL_DESC_COUNT field, the
18067 value in SQL_DESC_COUNT is not incremented.

18068 The descriptor record may undergo other validity checks at execute time as a result of a call to
18069 *SQLExecDirect*(), *SQLExecute*(), or *SQLPrepare*(); or at fetch time as a result of a call to
18070 *SQLFetch*(), *SQLFetchScroll*(), or *SQLSetPos*().

18071 **SEE ALSO**

18072	**For information about**	**See**
18073	Setting single descriptor fields	*SQLSetDescField*()
18074	Getting a single descriptor field	*SQLGetDescField*()
18075	Getting multiple descriptor fields	*SQLGetDescRec*()
18076	Binding a column	*SQLBindCol*()
18077	Binding a parameter	*SQLBindParam*()

18078 **CHANGE HISTORY**

18079 **Version 2**

18080 Revised generally. See **Alignment with Popular Implementations** on page 2. Also see the list in
18081 **Descriptor Fields Added in Version 2** on page 483.

18082 **NAME**

18083 SQLSetEnvAttr — Set attributes that govern aspects of environments.

18084 **SYNOPSIS**

```
18085 SQLRETURN SQLSetEnvAttr(
18086   SQLHENV EnvironmentHandle,
18087   SQLINTEGER Attribute,
18088   SQLPOINTER ValuePtr,
18089   SQLINTEGER StringLength);
```

18090 **ARGUMENTS**

18091 *EnvironmentHandle* [Input]
18092 Environment handle.

18093 *Attribute* [Input]
18094 Attribute to set, listed in **Environment Attribute** on page 489.

18095 *ValuePtr* [Input]
18096 Pointer to the value to be associated with Attribute. Depending on the value of *Attribute*,
18097 **ValuePtr* is a 32-bit integer value or points to a null-terminated character string.

18098 *StringLength* [Input]
18099 If *ValuePtr* points to a character string or a binary buffer, this argument should be the length
18100 of **ValuePtr*. If *ValuePtr* is a pointer, but not to a string or binary buffer, then *StringLength*
18101 should have the value SQL_IS_POINTER. If *ValuePtr* is not a pointer, then *StringLength*
18102 should have the value SQL_IS_NOT_POINTER.

18103 **RETURN VALUE**

18104 SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.

18105 **DIAGNOSTICS**

18106 When *SQLSetEnvAttr*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated
18107 SQLSTATE value can be obtained by calling *SQLGetDiagRec*() with a *HandleType* of
18108 SQL_HANDLE_ENV and a *Handle* of *EnvironmentHandle*. The following table lists the
18109 SQLSTATE values commonly returned by *SQLSetEnvAttr*(). The return code associated with
18110 each SQLSTATE value is SQL_ERROR, except that for SQLSTATE values in class 01, the return
18111 code is SQL_SUCCESS_WITH_INFO. If a data source does not support an environment
18112 attribute, the error can be returned only during connect time.

18113 01000 — General warning
18114 Implementation-defined informational message.

18115 01S02 — Attribute value changed
18116 The data source did not support the value specified in **ValuePtr* and substituted a similar
18117 value.

18118 HY000 — General error
18119 An error occurred for which there was no specific SQLSTATE and for which no
18120 implementation-specific SQLSTATE was defined. The error message returned by
18121 *SQLGetDiagRec*() in the **MessageText* buffer describes the error and its cause.

18122 HY001 — Memory allocation error
18123 The implementation failed to allocate memory required to support execution or completion
18124 of the function.

18125 HY009 — Invalid use of null pointer
18126 *Attribute* identified an attribute that required a string value, and *ValuePtr* was a null pointer.

18127 HY011 — Attribute cannot be set now
18128 A connection handle has been allocated on *EnvironmentHandle*.

18129 HY024 — Invalid attribute value
18130 A value was specified in **ValuePtr* that is inapplicable to *Attribute*, or **ValuePtr* was an
18131 empty string and *Attribute* requires a non-empty string.

18132 HY090 — Invalid string or buffer length
18133 *StringLength* was less than 0, but was not SQL_NTS.

18134 HY092 — Invalid attribute identifier
18135 *Attribute* was not valid for this connection to this data source. •

18136 HYC00 — Optional feature not implemented |
18137 *Attribute* was a valid environment attribute but is not supported by the data source. |

18138 *Attribute* was SQL_ATTR_OUTPUT_NTS, **ValuePtr* was SQL_FALSE, and the |
18139 implementation does not allow null termination to be disabled. |

18140 HYT01 — Connection timeout expired |
18141 The connection timeout period expired before the data source responded to the request. The |
18142 connection timeout period is set through *SQLSetConnectAttr*(), |
18143 SQL_ATTR_CONNECTION_TIMEOUT. |

18144 **COMMENTS** |

18145 An application can call *SQLSetEnvAttr*() only if no connection handle is allocated on the |
18146 environment. All environment attributes successfully set by the application for the environment
18147 persist until *SQLFreeHandle*() is called on the the environment. More than one environment
18148 handle can be allocated simultaneously.

18149 The format of information set through **ValuePtr* depends on the specified attribute.
18150 *SQLSetEnvAttr*() accepts attribute information in one of two formats: a null-terminated character
18151 string or a 32-bit integer value. The format of each is noted in the attribute's description.

18152 **Environment Attribute**

18153 The caller sets *Attribute* to the value listed below to obtain the following environment attribute
18154 in **ValuePtr*.

18155 SQL_ATTR_OUTPUT_NTS
18156 This attribute controls the implementation's use of null termination in output arguments. |
18157 (See **Null Termination** on page 44.) This attribute affects all XDBC functions called for the •
18158 environment (and for any connection allocated under the environment) that have
18159 character-string parameters. |

18160 If this attribute has the value SQL_TRUE, then the implementation uses null termination to |
18161 indicate the length of output character strings. If this attribute has the value SQL_FALSE, |
18162 then the implementation does not use null termination. |

18163 The initial value is SQL_TRUE on all X/Open-compliant implementations. Moreover, it is |
18164 implementation-defined whether the application is permitted to change the value to |
18165 SQL_FALSE.

18166 **SEE ALSO** •

18167 **For information about** **See**

18168 Returning the setting of an environment attribute *SQLGetEnvAttr*()
18169 Allocating a handle *SQLAllocHandle*()

18170 **CHANGE HISTORY**

18171 **Version 2**
18172 Revised generally. See **Alignment with Popular Implementations** on page 2.

18173 **NAME**

18174 SQLSetPos — Set the cursor position in a row-set and refresh, update, or delete data in the result
18175 set.

18176 **SYNOPSIS**

```
18177 SQLRETURN SQLSetPos(
18178     SQLHSTMT StatementHandle,
18179     SQLUSMALLINT RowNumber,
18180     SQLUSMALLINT Operation,
18181     SQLUSMALLINT LockType);
```

18182 **ARGUMENTS**

18183 *StatementHandle* [Input]
18184 Statement handle.

18185 *RowNumber* [Input]
18186 Position of the row in the row-set on which to perform the operation specified with
18187 *Operation*. If *RowNumber* is 0, the operation applies to every row in the row-set. See
18188 **RowNumber Argument** on page 495.

18189 *Operation* [Input]
18190 Operation to perform. Must be one of the following:

- 18191 SQL_POSITION
- 18192 SQL_REFRESH
- 18193 SQL_UPDATE
- 18194 SQL_DELETE

18195 See **Operation Argument** on page 495.

18196 *LockType* [Input]
18197 Specifies how to lock the row after performing the operation specified by *Operation*. Must
18198 be one of the following:

- 18199 SQL_LOCK_NO_CHANGE
- 18200 SQL_LOCK_EXCLUSIVE
- 18201 SQL_LOCK_UNLOCK

18202 See **LockType Argument** on page 497.

18203 **RETURN VALUE**

18204 SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NEED_DATA, SQL_STILL_EXECUTING,
18205 SQL_ERROR, or SQL_INVALID_HANDLE.

18206 **DIAGNOSTICS**

18207 When *SQLSetPos*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated
18208 SQLSTATE value can be obtained by calling *SQLGetDiagRec*() with a *HandleType* of
18209 SQL_HANDLE_STMT and a *Handle* of *StatementHandle*. The following SQLSTATE values are
18210 commonly returned by *SQLSetPos*().

18211 The return code associated with each SQLSTATE value is SQL_ERROR, except that for
18212 SQLSTATE values in class 01, the return code is SQL_SUCCESS_WITH_INFO, and except that, if
18213 the row-set size is greater than 1 and the operation was applied to at least one row successfully,
18214 the return code is SQL_SUCCESS_WITH_INFO.

18215 01000 — General warning
18216 Implementation-defined informational message.

18217 01001 — Cursor operation conflict
18218 *Operation* was SQL_DELETE or SQL_UPDATE, and no rows or more than one row were

18219 deleted or updated. (For more information about updates to more than one row, see the
18220 description of the SQL_ATTR_SIMULATE_CURSORattribute in *SQLSetStmtAttr*().)

18221 *Operation* was SQL_DELETE or SQL_UPDATE, and the operation failed because of
18222 optimistic concurrency, discussed in Section 14.3.2 on page 192.

18223 01004 — String data, right truncation
18224 String or binary data returned for a column or columns with a data type of SQL_C_CHAR
18225 or SQL_C_BINARY resulted in the truncation of non-blank character or non-NULL binary
18226 data.

18227 01S01 — Error in row
18228 *RowNumber* was 0 and an error occurred in one or more rows while performing the
18229 operation specified with *Operation*.

18230 01S07 — Fractional truncation
18231 *Operation* was SQL_REFRESH, the data type of the application buffer was not
18232 SQL_C_CHAR or SQL_C_BINARY, and the data returned to application buffers for one or
18233 more columns was truncated. For numeric data types, the fractional part of the number was
18234 truncated. For time, timestamp, and interval data types containing a time component, the
18235 fractional portion of the time was truncated.

18236 07006 — Restricted data type attribute violation
18237 The data value of a column in the result set could not be converted to the data type
18238 specified by *TargetType*in the call to *SQLBindCol*().

18239 21S02 — Degree of derived table does not match column list
18240 *Operation* was SQL_UPDATE, and no columns were updatable because all columns were
18241 either unbound, read-only, or the value in the bound length/indicator buffer was
18242 SQL_COLUMN_IGNORE.

18243 22001 — String data, right truncation
18244 The assignment of a character or binary value to a column resulted in the truncation of
18245 non-blank (for characters) or non-null (for binary) characters or octets.

18246 22003 — Numeric value out of range
18247 *Operation* was SQL_UPDATE, and the assignment of a numeric value to a column in the
18248 result set caused the whole (as opposed to fractional) part of the number to be truncated.

18249 *Operation* was SQL_REFRESH, and returning the numeric value for one or more bound
18250 columns would have caused a loss of significant digits.

18251 22007 — Invalid date/time format
18252 *Operation* was SQL_UPDATE, and an invalid date or timestamp value was assigned to a
18253 column in the result set.

18254 *Operation* was SQL_REFRESH, and an invalid date or timestamp value would have been
18255 returned for one or more bound columns.

18256 22008 — Date/time field overflow
18257 *Operation* was SQL_UPDATE, and the performance of date/time arithmetic on data being
18258 sent to the result set resulted in a date/time field (i.e., the year, month, day, hour, minute, or
18259 second field) of the result being outside the permissible range of values for the field, or
18260 being invalid based on the natural rules for date/times based on the Gregorian calendar.

18261 *Operation* was SQL_REFRESH, and the performance of date/time arithmetic on data being
18262 retrieved from the result set resulted in a date/time field (i.e., the year, month, day, hour,
18263 minute, or second field) of the result being outside the permissible range of values for the
18264 field, or being invalid based on the natural rules for date/times based on the Gregorian
18265 calendar.

18266 22015 — Interval field overflow
18267 *Operation* was SQL_UPDATE, and the assignment of an exact numeric value to a column in
18268 the result set with an interval data type caused a loss of significant digits.

18269 *Operation* was SQL_UPDATE, and the assignment of an interval value to a column in the
18270 result set with an interval data type caused a loss of significant digits in the leading field of
18271 the interval.

18272 *Operation* was SQL_UPDATE, and there was no representation of the data in the interval
18273 data type of the result set.

18274 *Operation* was SQL_REFRESH, and returning an exact numeric value to an application
18275 buffer with an interval data type caused a loss of significant digits.

18276 *Operation* was SQL_REFRESH, and returning an interval value to an application buffer with
18277 an interval data type caused a loss of significant digits in the leading field of the interval.

18278 *Operation* argument was SQL_REFRESH, and there was no representation of the data in the
18279 interval C structure in the application buffer.

18280 22018 — Invalid character value for cast specification
18281 *Operation* was SQL_UPDATE, a character column in the result set was bound to an exact
18282 numeric or an approximate numeric C buffer, and a character value in the result set could
18283 not be cast to a valid exact numeric or approximate numeric value, respectively.

18284 *Operation* was SQL_UPDATE, a character column in the result set was bound to a date, time,
18285 timestamp, or interval C buffer, and a character value in the result set could not be cast to a
18286 valid date, time, timestamp, or interval value, respectively.

18287 *Operation* was SQL_REFRESH, a character column in an application buffer was bound to an
18288 exact numeric or approximate numeric data type in the result set, and a character value in
18289 the application buffer could not be cast to a valid exact numeric or approximate numeric
18290 value, respectively.

18291 *Operation* was SQL_REFRESH, a character column in an application buffer was bound to a
18292 date, time, timestamp, or interval data type in the result set, and a value in the application
18293 buffer could not be cast to a valid date, time, timestamp, or interval value, respectively. •

18294 23000 — Integrity constraint violation
18295 *Operation* was SQL_DELETE or SQL_UPDATE, and an integrity constraint was violated.

18296 24000 — Invalid cursor state
18297 *StatementHandle* was in an executed state but no result set was associated with the
18298 *StatementHandle.*

18299 A cursor was open on *StatementHandle*. |

18300 *Operation* was SQL_DELETE, SQL_REFRESH, or SQL_UPDATE, and the cursor was
18301 positioned before the start of the result set or after the end of the result set.

18302 42000 — Syntax error or access violation
18303 The data source was unable to lock the row as needed to perform the operation requested in |
18304 *Operation.* |

18305 The data source was unable to lock the row as requested in *LockType*. |

18306 HY000 — General error |
18307 An error occurred for which there was no specific SQLSTATE and for which no |
18308 implementation-specific SQLSTATE was defined. The error message returned by |
18309 *SQLGetDiagRec*() in the **MessageText* buffer describes the error and its cause. |

18310 HY001 — Memory allocation error
18311 The implementation was unable to allocate memory required to support execution or
18312 completion of the function.

18313 HY008 — Operation canceled
18314 Asynchronous processing was enabled for *StatementHandle*. The function was called and
18315 before it completed execution, *SQLCancel*() was called on *StatementHandle*. The function
18316 was then called again on *StatementHandle*.

18317 The function was called and, before it completed execution, *SQLCancel*() was called on the
18318 *StatementHandle* from a different thread in a multithread application.

18319 HY010 — Function sequence error
18320 *StatementHandle* was not in an executed state. The function was called without first calling
18321 *SQLExecDirect*(), *SQLExecute*(), or a catalog function.

18322 An asynchronously executing function (not this one) was called for *StatementHandle* and
18323 was still executing when this function was called.

18324 *SQLBulkOperations*(), *SQLExecDirect*(), *SQLExecute*(), or *SQLSetPos*() was called for the
18325 *StatementHandle* and returned SQL_NEED_DATA. This function was called before data was
18326 sent for all data-at-execution parameters or columns. *SQLSetPos*() was called for
18327 *StatementHandle* before *SQLFetchScroll*() or *SQLFetch*() was called.

18328 HY090 — Invalid string or buffer length
18329 *Operation* was SQL_UPDATE, a data value was a null pointer, and the column length value
18330 was not 0, SQL_DATA_AT_EXEC, SQL_COLUMN_IGNORE, SQL_NULL_DATA, or less
18331 than or equal to SQL_LEN_DATA_AT_EXEC_OFFSET.

18332 *Operation* was SQL_UPDATE, a data value was not a null pointer, and the column length
18333 value was less than 0, but not equal to SQL_DATA_AT_EXEC, SQL_COLUMN_IGNORE,
18334 SQL_NTS, or SQL_NULL_DATA, or less than or equal to
18335 SQL_LEN_DATA_AT_EXEC_OFFSET. (This error is reported only if the application data
18336 type is SQL_C_BINARY or SQL_C_CHAR.)

18337 The value in a length/indicator buffer was SQL_DATA_AT_EXEC; the SQL type was either
18338 SQL_LONGVARCHAR, SQL_WLONGVARCHAR, SQL_LONGVARBINARY, or a long,
18339 data source-specific data type; and the SQL_NEED_LONG_DATA_LEN option in
18340 *SQLGetInfo*() was "Y".

18341 HY092 — Invalid attribute identifier
18342 *Operation* was invalid.

18343 *LockType* was invalid.

18344 *Operation* was SQL_UPDATE or SQL_DELETE, and the SQL_CONCURRENCY statement
18345 attribtue was SQL_ATTR_CONCUR_READ_ONLY.

18346 HY107 — Row value out of range
18347 *RowNumber* was greater than the number of rows in the row-set.

18348 HY109 — Invalid cursor position
18349 The cursor associated with *StatementHandle* was defined as forward-only, so the cursor
18350 could not be positioned within the row-set. See the description for the
18351 SQL_ATTR_CURSOR_TYPE attribute in *SQLSetStmtAttr*().

18352 *Operation* was SQL_UPDATE, SQL_DELETE, or SQL_REFRESH, and the row identified by
18353 *RowNumber* had been deleted or had not been fetched.

18354 *RowNumber* was 0, *Operation* was SQL_POSITION, and *SQLSetPos*() was called after
18355 *SQLBulkOperations*() was called, and before *SQLFetchScroll*() or *SQLFetch*() was called.

18356 HYC00 — Optional feature not implemented
18357 The implementation does not support the operation requested in *Operation* or *LockType*.

18358 HYT00 — Timeout expired
18359 The query timeout period expired before the data source returned the result set. The
18360 timeout period is set through *SQLSetStmtAttr*() with an Attribute of
18361 SQL_ATTR_QUERY_TIMEOUT.

18362 HYT01 — Connection timeout expired
18363 The connection timeout period expired before the data source responded to the request. The
18364 connection timeout period is set through *SQLSetConnectAttr*(),
18365 SQL_ATTR_CONNECTION_TIMEOUT.

18366 IM001 — Function not supported
18367 The function is not supported on the current connection to the data source.

18368 **Comments**

18369 **RowNumber Argument**

18370 *RowNumber* specifies the number of the row in the row-set on which to perform the operation
18371 specified by *Operation*. If *RowNumber* is 0, the operation applies to every row in the row-set.
18372 *RowNumber* must be a value from 0 to the number of rows in the row-set.

18373 **Note:** In the C language, arrays are 0-based, while *RowNumber* is 1-based. For example, to
18374 update the fifth row of the row-set, an application modifies the row-set buffers at array index 4,
18375 but specifies a *RowNumber* of 5.

18376 All operations position the cursor on the row specified by *RowNumber*. The following operations
18377 require a cursor position:

18378 • Positioned UPDATE and DELETE statements.
18379 • Calls to *SQLGetData*().
18380 • Calls to *SQLSetPos*() with the SQL_DELETE, SQL_REFRESH, and SQL_UPDATE options.

18381 For example, if *RowNumber* is 2 for a call to *SQLSetPos*() with an *Operation* of SQL_DELETE, the
18382 cursor is positioned on the second row of the row-set, and that row is deleted. The entry in the
18383 implementation row status array (pointed to by the SQL_ATTR_ROW_STATUS_PTR statement
18384 attribute) for the second row is changed to SQL_ROW_DELETED.

18385 An application can specify a cursor position when it calls *SQLSetPos*(). Generally, it calls
18386 *SQLSetPos*() with the SQL_POSITION or SQL_REFRESH operation to position the cursor before
18387 executing a positioned UPDATE or DELETE statement or calling *SQLGetData*().

18388 **Operation Argument**

18389 *Operation* supports the following operations. (To determine which options are supported by a
18390 data source, an application calls *SQLGetInfo*() as described in **Detecting Cursor Capabilities**
18391 **with SQLGetInfo()** on page 402):

18392 SQL_POSITION
18393 The implementation positions the cursor on the row specified by *RowNumber*.

18394 The contents of the row status array pointed to by the
18395 SQL_ATTR_ROW_OPERATION_PTR statement attribute are ignored for the
18396 SQL_POSITION *Operation*.

18397 SQL_REFRESH
18398 The implementation positions the cursor on the row specified by *RowNumber* and refreshes
18399 data in the row-set buffers for that row. For more information about how the

18400 implementation returns data in the row-set buffers, see the descriptions of row-wise and
18401 column-wise binding in *SQLBindCol*().

18402 *SQLSetPos*() with an *Operation* of SQL_REFRESH updates the status and content of the rows
18403 within the current fetched row-set. This includes refreshing the bookmarks. Because the
18404 data in the buffers is refreshed, but not refetched, the membership in the row-set is fixed.
18405 This is different from the refresh performed by a call to *SQLFetchScroll*() with a
18406 *FetchOrientation* of SQL_FETCH_RELATIVE and a *RowNumber* equal to 0, which refetches
18407 the row-set from the result set, so it can show added data and remove deleted data.

18408 Added rows do not appear when a refresh with *SQLSetPos*() is performed. This rule differs
18409 from *SQLFetchScroll*() with a *FetchType* of SQL_FETCH_RELATIVE and a *RowNumber* equal
18410 to 0, which also refreshes the current row-set, but shows added records and packs deleted
18411 records if these operations are supported by the cursor.

18412 If the row status array exists, a successful refresh with *SQLSetPos*() changes a row status of
18413 SQL_ROW_ADDED to SQL_ROW_SUCCESS, and changes a row status of
18414 SQL_ROW_UPDATED to the row's new status. If an error occurs in a *SQLSetPos*()
18415 operation on a row, the row status is set to SQL_ROW_ERROR.

18416 A refresh with *SQLSetPos*() does not change the row status of a row that is marked
18417 SQL_ROW_DELETED. Deleted rows within the row-set continue to be marked as deleted
18418 until the next fetch. The rows disappear at the next fetch if the cursor supports packing (in
18419 which a subsequent *SQLFetch*() or *SQLFetchScroll*() does not return deleted rows).

18420 The contents of the row status array pointed to by the
18421 SQL_ATTR_ROW_OPERATION_PTR statement attribute are ignored for the
18422 SQL_REFRESH *Operation*.

18423 On some implementations, for a cursor opened with an SQL_ATTR_CONCURRENCY
18424 statement attribute of SQL_CONCUR_ROWVER or SQL_CONCUR_VALUES, a refresh
18425 with *SQLSetPos*() updates the optimistic concurrency values used by the data source to
18426 detect that the row has changed. This occurs for each row that is refreshed.

18427 SQL_UPDATE

18428 The implementation positions the cursor on the row specified by *RowNumber* and updates
18429 the underlying row of data with the values in the row-set buffers (*TargetValuePtr* in
18430 *SQLBindCol*()). It retrieves the lengths of the data from the length/indicator buffers
18431 (*StrLen_or_IndPtr* in *SQLBindCol*()). If the length of any column is
18432 SQL_COLUMN_IGNORE, the column is not updated. After updating the row, the
18433 implementation changes the corresponding element of the row status array to
18434 SQL_ROW_UPDATED or SQL_ROW_SUCCESS_WITH_INFO (if the row status array
18435 exists).

18436 It is implementation-defined what the behavior is if *SQLSetPos*() with *Operation* of
18437 SQL_UPDATE is called on a cursor that contains duplicate columns.

18438 SQL_DELETE

18439 The implementation positions the cursor on the row specified by *RowNumber* and deletes
18440 the underlying row of data. It changes the corresponding element of the row status array to
18441 SQL_ROW_DELETED. After the row has been deleted, the following are not valid for the
18442 row: positioned UPDATE and DELETE statements, calls to *SQLGetData*(), and calls to
18443 *SQLSetPos*() with *Operation* set to anything except SQL_POSITION. Implementations that
18444 support packing delete the row from the cursor when new data is retrieved from the data
18445 source.

18446 Whether the row remains visible depends on the cursor type. For example, deleted rows are
18447 visible to static and keyset-driven cursors but invisible to dynamic cursors.

18448 **LockType Argument**

18449 *LockType* gives applications a way to control concurrency. Generally, data sources that support
18450 concurrency levels and transactions will only support the SQL_LOCK_NO_CHANGE value of
18451 *LockType*.

18452 *LockType* specifies the lock state of the row after *SQLSetPos*() has been executed. If the
18453 implementation cannot lock the row either to perform the requested operation or to satisfy
18454 *LockType*, it returns SQL_ERROR and SQLSTATE 42000 (Syntax error or access violation).

18455 Although *LockType* is specified for a single statement, the lock accords the same privileges to all
18456 statements on the connection. In particular, a lock that is acquired by one statement on a
18457 connection can be unlocked by a different statement on the same connection.

18458 A row locked through *SQLSetPos*() remains locked until the application calls *SQLSetPos*() for the
18459 row with *LockType* set to SQL_LOCK_UNLOCK, or the application calls *SQLFreeHandle*() for the
18460 statement or *SQLFreeStmt*() with the SQL_CLOSE option. For a data source that supports |
18461 transactions, a row locked through *SQLSetPos*() is unlocked when the application calls
18462 *SQLEndTran*() to commit or roll back a transaction on the connection (if a cursor is closed when
18463 a transaction is committed or rolled back, as indicated by the
18464 SQL_CURSOR_COMMIT_BEHAVIOR and SQL_CURSOR_ROLLBACK_BEHAVIOR options in |
18465 *SQLGetInfo*()).

18466 *LockType* supports the following types of locks. To determine which locks are supported by a
18467 data source, an application calls *SQLGetInfo*() with the
18468 SQL_DYNAMIC_CURSOR_ATTRIBUTES1, SQL_FORWARD_ONLY_CURSOR_ATTRIBUTES1,
18469 SQL_KEYSET_CURSOR_ATTRIBUTES1, or SQL_STATIC_CURSOR_ATTRIBUTES1 option |
18470 (depending on the type of the cursor).

18471 SQL_LOCK_NO_CHANGE
18472 The implementation ensures that the row is in the same locked or unlocked state as it was
18473 before *SQLSetPos*() was called. This value of *LockType* lets data sources that do not support
18474 explicit row-level locking use whatever locking is required by the current concurrency and
18475 transaction isolation levels.

18476 SQL_LOCK_EXCLUSIVE
18477 The implementation locks the row exclusively. A statement on a different connection or in a
18478 different application cannot be used to acquire any locks on the row.

18479 SQL_LOCK_UNLOCK
18480 The implementation unlocks the row. |

18481 If the implementation supports SQL_LOCK_EXCLUSIVE but not SQL_LOCK_UNLOCK, a row
18482 that is locked remains locked until the application unlocks it as described above.

18483 If the implementation supports SQL_LOCK_EXCLUSIVE but not SQL_LOCK_UNLOCK, a row
18484 that is locked remains locked until the application calls *SQLFreeHandle*() for the statement or
18485 *SQLFreeStmt*() with the SQL_CLOSE option. If the implementation supports transactions and
18486 closes the cursor upon committing or rolling back the transaction, the application calls
18487 *SQLEndTran*().

18488 For the update and delete operations in *SQLSetPos*(), the application uses *LockType* as follows:

18489 • To guarantee that a row does not change after it is retrieved, an application calls *SQLSetPos*()
18490 with *Operation* set to SQL_REFRESH and *LockType* set to SQL_LOCK_EXCLUSIVE.

18491 • If the application sets *LockType* to SQL_LOCK_NO_CHANGE, the implementation |
18492 guarantees that an update or delete operation succeeds only if the application specified
18493 SQL_CONCUR_LOCK for the SQL_ATTR_CONCURRENCY statement attribute.

- If the application specifies SQL_CONCUR_ROWVER or SQL_CONCUR_VALUES for the SQL_ATTR_CONCURRENCY statement attribute, the implementation compares row versions or values and rejects the operation if the row has changed since the application fetched the row.
- If the application specifies SQL_CONCUR_READ_ONLY for the SQL_ATTR_CONCURRENCY statement attribute, the implementation rejects any update or delete operation.

For more information about the SQL_ATTR_CONCURRENCY statement attribute, see *SQLSetStmtAttr*().

Row Status Arrays

Two row status arrays are used when calling *SQLSetPos*():

- The implementation row status array contains status values for each row of data in the row-set. The implementation sets the status values in this array after a call to *SQLFetch*(), *SQLFetchScroll*(), *SQLBulkOperations*(), or *SQLSetPos*(). This array is pointed to by the SQL_ATTR_ROW_STATUS_PTR statement attribute.
- The application row status array contains a value for each row in the row-set that indicates whether a call to *SQLSetPos*() for a bulk operation is ignored or performed. Each element in the array is set to either SQL_ROW_PROCEED (the default) or SQL_ROW_IGNORE. This array is pointed to by the SQL_ATTR_ROW_OPERATION_PTR statement attribute.

The number of elements in the row status arrays must equal the number of rows in the row-set (as defined by the SQL_ATTR_ROW_ARRAY_SIZE statement attribute).

For information about the implementation row status array, see *SQLFetch*(). For information about the application row status array, see **Ignoring a Row in a Bulk Operation** on page 501.

Using SQLSetPos()

Before an application calls *SQLSetPos*(), it must perform the following sequence of steps:

- If the application will call *SQLSetPos*() with *Operation* set to SQL_UPDATE, call *SQLBindCol*() (or *SQLSetDescRec*()) for each column to specify its data type and bind buffers for the column's data and length.
- If the application will call *SQLSetPos*() with *Operation* set to SQL_DELETE or SQL_UPDATE, call *SQLColAttribute*() to make sure that the columns to be deleted or updated are updatable.
- Call *SQLExecDirect*(), *SQLExecute*(), or a catalog function to create a result set.
- Call *SQLFetch*() or *SQLFetchScroll*() to retrieve the data.

Deleting Data Using SQLSetPos()

To delete data with *SQLSetPos*(), an application calls *SQLSetPos*() with *RowNumber* set to the number of the row to delete and *Operation* set to SQL_DELETE.

After deleting the data, the implementation changes the value in the implementation row status array for the appropriate row to SQL_ROW_DELETED (or SQL_ROW_ERROR).

18531 **Updating Data Using SQLSetPos()**

18532 An application can pass the value for a column either in the bound data buffer or with one or
18533 more calls to *SQLPutData*(). Columns whose data is passed with *SQLPutData*() are known as
18534 data-at-execution columns. These are commonly used to send data for SQL_LONGVARBINARY
18535 and SQL_LONGVARCHAR columns and can be mixed with other columns.

18536 To update data with *SQLSetPos*(), an application:

18537 1. Places values in the data and length/indicator buffers bound with *SQLBindCol*():

18538 — For normal columns, the application places the new column value in the **TargetValuePtr*
18539 buffer and the length of that value in the **StrLen_or_IndPtr* buffer. If the row should not
18540 be updated, the application places SQL_ROW_IGNORE in that row's element of the
18541 operation row status array.

18542 — For data-at-execution columns, the application places an application-defined value,
18543 such as the column number, in the **TargetValuePtr* buffer. The value can be used later to
18544 identify the column.

18545 The application places the result of the SQL_LEN_DATA_AT_EXEC(length) macro in
18546 the **StrLen_or_IndPtr* buffer. If the SQL data type of the column is
18547 SQL_LONGVARBINARY, SQL_LONGVARCHAR, or a long, data source-specific data |
18548 type and a call to *SQLGetInfo*() with the SQL_NEED_LONG_DATA_LEN option would |
18549 return "Y", length is the number of octets of data to be sent for the parameter; |
18550 otherwise, it must be a nonnegative value and is ignored.

18551 2. Calls *SQLSetPos*() with *Operation* set to SQL_UPDATE to update the row of data.

18552 — If there are no data-at-execution columns, the process is complete.

18553 — If there are any data-at-execution columns, the function returns SQL_NEED_DATA,
18554 and proceeds to step 3.

18555 3. Calls *SQLParamData*() to retrieve the address of the **TargetValuePtr* buffer for the first
18556 data-at-execution column to be processed. The application retrieves the application-
18557 defined value from the **TargetValuePtr* buffer.

18558 **Note:** Although data-at-execution parameters are similar to data-at-execution columns,
18559 the value returned by *SQLParamData*() is different for each.

18560 — Data-at-execution parameters are parameters in an SQL statement for which data will
18561 be sent with *SQLPutData*() when the statement is executed with *SQLExecDirect*() or
18562 *SQLExecute*(). They are bound with *SQLBindParameter*(), or by setting descriptors with
18563 *SQLSetDescRec*(). The value returned by *SQLParamData*() is a 32-bit value passed to
18564 *SQLBindParameter*() in *ParameterValuePtr*.

18565 — Data-at-execution columns are columns in a row-set for which data will be sent with |
18566 *SQLPutData*() when a row is updated with *SQLSetPos*(). They are bound with
18567 *SQLBindCol*(). The value returned by *SQLParamData*() is the address of the row in the
18568 **TargetValuePtr* buffer that is being processed.

18569 4. Calls *SQLPutData*() one or more times to send data for the column. More than one call is
18570 needed if all the data value cannot be returned in the **TargetValuePtr* buffer specified in
18571 *SQLPutData*(); multiple calls to *SQLPutData*() for the same column are allowed only when
18572 sending character C data to a column with a character, binary, or data-source-specific data
18573 type or when sending binary C data to a column with a character, binary, or data-source-
18574 specific data type.

18575 5. Calls *SQLParamData*() again to signal that all data has been sent for the column.

— If there are more data-at-execution columns, *SQLParamData*() returns SQL_NEED_DATA and the address of the *TargetValuePtr* buffer for the next data-at-execution column to be processed. The application repeats steps 4 and 5.

— If there are no more data-at-execution columns, the process is complete. If the statement was executed successfully, *SQLParamData*() returns SQL_SUCCESS or SQL_SUCCESS_WITH_INFO; if the execution failed, it returns SQL_ERROR. At this point, *SQLParamData*() can return any SQLSTATE that can be returned by *SQLSetPos*().

If data has been updated, the implementation changes the value in the implementation row status array for the appropriate row to SQL_ROW_UPDATED.

After *SQLSetPos*() returns SQL_NEED_DATA, and before data is sent for all data-at-execution columns, the operation is canceled, or an error occurs in *SQLParamData*() or *SQLPutData*(), the application can only call *SQLCancel*(), *SQLGetDiagField*(), *SQLGetDiagRec*(), *SQLGetFunctions*(), *SQLParamData*(), or *SQLPutData*() for the statement or the connection associated with the statement. If it calls any other function for the statement or the connection associated with the statement, the function returns SQL_ERROR and SQLSTATE HY010 (Function sequence error).

If the application calls *SQLCancel*() while the implementation still needs data for data-at-execution columns, the implementation cancels the operation. The application can then call *SQLSetPos*() again; canceling does not affect the cursor state or the current cursor position.

Performing Bulk Operations

If *RowNumber* is 0, the implementation performs the operation specified in *Operation* for every row in the row-set that has a value of SQL_ROW_PROCEED in its field in the row status array pointed to by SQL_ATTR_ROW_OPERATION_PTR statement attribute. This is a valid value of *RowNumber* if *Operation* is SQL_DELETE, SQL_REFRESH, or SQL_UPDATE, but not SQL_POSITION. *SQLSetPos*() with an *Operation* of SQL_POSITION and a *RowNumber* equal to 0 returns SQLSTATE HY109 (Invalid cursor position).

If an error occurs that pertains to the entire row-set, such as SQLSTATE HYT00 (Timeout expired), the implementation returns SQL_ERROR and the appropriate SQLSTATE. The contents of the row-set buffers are undefined and the cursor position is unchanged.

If an error occurs that pertains to a single row, the implementation:

- Sets the element for the row in the implementation row status array pointed to by the SQL_ATTR_ROW_STATUS_PTR statement attribute to SQL_ROW_ERROR.
- Posts one or more additional SQLSTATEs for the error in the error queue, and sets the SQL_DIAG_ROW_NUMBER field in the diagnostic data structure.

After it has processed the error or warning, if the implementation completes the operation for the remaining rows in the row-set, it returns SQL_SUCCESS_WITH_INFO. Thus, for each row that returned an error, the error queue contains zero or more additional SQLSTATEs. If the implementation stops the operation after it has processed the error or warning, it returns SQL_ERROR.

If the implementation returns any warnings, such as SQLSTATE 01004 (Data truncated), it returns warnings that apply to the entire row-set or to unknown rows in the row-set before it returns the error information that applies to specific rows. It returns warnings for specific rows along with any other error information about those rows.

If *RowNumber* is equal to 0 and *Operation* is SQL_UPDATE, SQL_REFRESH, or SQL_DELETE, then the number of rows that *SQLSetPos*() operates on is pointed to by the SQL_ATTR_ROWS_FETCHED_PTR statement attribute.

18621 If *RowNumber* is equal to 0 and *Operation* is SQL_DELETE, SQL_REFRESH, or SQL_UPDATE, the
18622 current row after the operation is the same as the current row before the operation.

18623 **Ignoring a Row in a Bulk Operation**

18624 The application row status array can be used to indicate that a row in the current row-set should
18625 be ignored during a bulk operation using *SQLSetPos*(). To direct the implementation to ignore
18626 one or more rows during a bulk operation, an application performs the following steps:

18627 • Call *SQLSetStmtAttr*() to set the SQL_ATTR_ROW_OPERATION_PTR statement attribute to
18628 point to an array of SQLUSMALLINTs to contain status information. This field can also be
18629 set by calling *SQLSetDescField*() to set the SQL_DESC_ARRAY_STATUS_PTR header field of
18630 the ARD, which requires that an application obtains a descriptor handle.

18631 • Set each element of the row operation array to one of two values:

18632 — SQL_ROW_IGNORE, to indicate that the row is excluded for the bulk operation.

18633 — SQL_ROW_PROCEED, to indicate that the row is included in the bulk operation. (This is
18634 the default value.)

18635 • Call *SQLSetPos*() to perform the bulk operation.

18636 The following rules apply to the application row status array:

18637 • SQL_ROW_IGNORE and SQL_ROW_PROCEED only affect bulk operations using
18638 *SQLSetPos*() with an *Operation* of SQL_DELETE or SQL_UPDATE. They do not affect calls to
18639 *SQLSetPos*() with an *Operation* of SQL_REFRESH or SQL_POSITION.

18640 • The pointer is set to null by default.

18641 • If the pointer is null, then all rows are updated, as if all elements were set to
18642 SQL_ROW_PROCEED.

18643 • Setting an element to SQL_ROW_PROCEED does not guarantee that the operation will occur
18644 on that particular row. For example, if a certain row in the row-set has the status
18645 SQL_ROW_ERROR, then the implementation may not be able to update that row regardless
18646 of whether the application specified SQL_ROW_PROCEED or not. An application must
18647 always check the implementation row status array to see whether the operation was
18648 successful.

18649 • Since both SQL_ROW_SUCCESS and SQL_ROW_PROCEED are defined as 0 in the header
18650 file, reusing the row status array obtained from a previous operation applies the current
18651 operation to every row where the previous operation succeeded.

18652 Another effect of defining SQL_ROW_PROCEED as 0 is that initializing the row status array
18653 so that every element is 0 applies the operation to every row.

18654 • If *SQLSetPos*() is called to perform a bulk update or delete operation, then in any element of
18655 the application row array set to SQL_ROW_IGNORE, the corresponding element of the
18656 application row status array is unchanged. *SQLSetPos*().

18657 • An application should automatically set a read-only column to SQL_ROW_IGNORE.

18658 **Ignoring a Column in a Bulk Operation**

18659 To avoid unnecessary processing errors from trying to update read-only columns, the
18660 application can set the value in the bound length/indicator buffer to SQL_COLUMN_IGNORE.

18661 **SEE ALSO**

For information about	See
Binding a buffer to a column in a result set	*SQLBindCol*()
Performing bulk operations that do not relate to the cursor position	*SQLBulkOperations*()
Canceling statement processing	*SQLCancel*()
Fetching a block of data or scrolling through a result set	*SQLFetchScroll*()
Getting a single field of a descriptor	*SQLGetDescField*()
Getting multiple fields of a descriptor	*SQLGetDescRec*()
Setting a single field of a descriptor	*SQLSetDescField*()
Setting multiple fields of a descriptor	*SQLSetDescRec*()
Setting a statement attribute	*SQLSetStmtAttr*()

18674 **NAME**

18675 SQLSetStmtAttr — Set attributes related to a statement.

18676 **SYNOPSIS**

```
18677 SQLRETURN SQLSetStmtAttr(
18678   SQLHSTMT StatementHandle,
18679   SQLINTEGER Attribute,
18680   SQLPOINTER ValuePtr,
18681   SQLINTEGER StringLength);
```

18682 **ARGUMENTS**

18683 *StatementHandle* [Input]

18684 Statement handle.

18685 *Attribute* [Input]

18686 Option to set, listed in **Statement Attributes** on page 506.

18687 *ValuePtr* [Input]

18688 Pointer to the value to be associated with the attribute. Depending on *Attribute*, **ValuePtr* is
18689 a 32-bit unsigned integer value or points to a null-terminated character string, a binary
18690 buffer, or a implementation-defined value. For implementation-defined values of *Attribute*,
18691 **ValuePtr* may be a signed integer.

18692 *StringLength* [Input]

18693 If *ValuePtr* points to a character string or a binary buffer, *StringLength* should be the length
18694 of **ValuePtr*. If *ValuePtr* is a pointer, but not to a string or binary buffer, then *StringLength*
18695 should have the value SQL_IS_POINTER. If *ValuePtr* is not a pointer, then *StringLength*
18696 should have the value SQL_IS_NOT_POINTER.

18697 **RETURN VALUE**

18698 SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.

18699 **DIAGNOSTICS**

18700 When *SQLSetStmtAttr*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated
18701 SQLSTATE value can be obtained by calling *SQLGetDiagRec*() with a *HandleType* of
18702 SQL_HANDLE_STMT and a *Handle* of *StatementHandle*. The following SQLSTATE values are
18703 commonly returned by *SQLSetStmtAttr*(). The return code associated with each SQLSTATE
18704 value is SQL_ERROR, except that for SQLSTATE values in class 01, the return code is
18705 SQL_SUCCESS_WITH_INFO.

18706 01000 — General warning

18707 Implementation-defined informational message.

18708 01S02 — Attribute value changed

18709 The data source did not support the value specified in **ValuePtr*, or the value specified in
18710 **ValuePtr* was invalid because of SQL constraints or requirements, so the implementation
18711 substituted a similar value.

18712 08S01 — Communication link failure

18713 The communication link to the data source failed before the function completed processing.

18714 24000 — Invalid cursor state

18715 *Attribute* was SQL_ATTR_CONCURRENCY, SQL_ATTR_CURSOR_TYPE,
18716 SQL_ATTR_SIMULATE_CURSOR, or SQL_ATTR_USE_BOOKMARKS and a cursor was
18717 open.

18718 HY000 — General error

18719 An error occurred for which there was no specific SQLSTATE and for which no
18720 implementation-specific SQLSTATE was defined. The error message returned by

18721 *SQLGetDiagRec*() in the **MessageText* buffer describes the error and its cause.

18722 HY001 — Memory allocation error
18723 The implementation failed to allocate memory required to support execution or completion
18724 of the function.

18725 HY009 — Invalid use of null pointer
18726 *Attribute* identified an attribute that required a string value and *ValuePtr* was a null pointer.

18727 HY010 — Function sequence error
18728 An asynchronously executing function was called for *StatementHandle* and was still
18729 executing when this function was called.

18730 *SQLBulkOperations*(), *SQLExecDirect*(), *SQLExecute*(), or *SQLSetPos*() was called for
18731 *StatementHandle* and returned SQL_NEED_DATA. This function was called before data was
18732 sent for all data-at-execution parameters or columns.

18733 HY011 — Attribute cannot be set now
18734 *Attribute* was SQL_ATTR_CONCURRENCY, SQL_ ATTR_CURSOR_TYPE,
18735 SQL_ATTR_SIMULATE_CURSOR, or SQL_ATTR_USE_BOOKMARKS and the statement
18736 was prepared.

18737 HY017 — Invalid use of an automatically allocated descriptor handle.
18738 *Attribute* was SQL_ATTR_IMP_ROW_DESC or SQL_ATTR_IMP_PARAM_DESC. •

18739 *Attribute* was SQL_ATTR_APP_ROW_DESC or SQL_ATTR_APP_PARAM_DESC, and the •
18740 value in **ValuePtr* was an implicitly-allocated descriptor handle other than the handle
18741 originally allocated for the ARD or APD.

18742 HY024 — Invalid attribute value
18743 A value was specified in **ValuePtr* that is inapplicable to *Attribute*

18744 **ValuePtr* was an empty string and *Attribute* requires a non-empty string.

18745 *Attribute* was SQL_ATTR_APP_ROW_DESC or SQL_ATTR_APP_PARAM_DESC, and the
18746 handle pointed to by **ValuePtr* was not allocated on the same connection as
18747 *StatementHandle*.

18748 HY090 — Invalid string or buffer length
18749 *StringLength* was less than 0, but was not SQL_NTS.

18750 HY092 — Invalid attribute identifier
18751 *Attribute* was not valid for this connection to this data source.

18752 *Attribute* identified a read-only attribute.

18753 HYC00 — Optional feature not implemented
18754 *Attribute* was a valid value but is not supported by the data source.

18755 *Attribute* was SQL_ATTR_ASYNC_ENABLE and a call to *SQLGetInfo*() with the
18756 SQL_ASYNC_MODE option returns SQL_AM_CONNECTION.

18757 HYT01 — Connection timeout expired
18758 The connection timeout period expired before the data source responded to the request. The
18759 connection timeout period is set through *SQLSetConnectAttr*(),
18760 SQL_ATTR_CONNECTION_TIMEOUT.

18761 IM001 — Function not supported
18762 The function is not supported on the current connection to the data source.

18763 **COMMENTS**

18764 Statement attributes for a statement remain in effect until they are changed by another call to

18765 *SQLSetStmtAttr*() or the statement is dropped by calling *SQLFreeHandle*(). Calling
18766 *SQLFreeStmt*() with the SQL_CLOSE, SQL_UNBIND, or SQL_RESET_PARAMS options does not
18767 reset statement attributes.

18768 Some statement attributes support substitution of a similar value if the data source does not
18769 support the value specified in **ValuePtr*. In such cases, the implementation returns
18770 SQL_SUCCESS_WITH_INFO and SQLSTATE 01S02 (Attribute value changed). For example, if
18771 *Attribute* is SQL_ATTR_CONCURRENCY, **ValuePtr* is SQL_CONCUR_ROWVER, and the data
18772 source does not support this, the implementation substitutes SQL_CONCUR_VALUES and
18773 returns SQL_SUCCESS_WITH_INFO. To determine the substituted value, an application calls
18774 *SQLGetStmtAttr*().

18775 The format of information set with *ValuePtr* depends on the attribute. *SQLSetStmtAttr*() accepts
18776 attribute information in one of two different formats: a character string or a 32-bit integer value. |
18777 The format of each is noted in the attribute's description. This format applies to the information
18778 returned for each attribute in *SQLGetStmtAttr*(). Character strings pointed to by *ValuePtr* of
18779 *SQLSetStmtAttr*() have a length of *StringLength*.

18780 **Setting Statement Attributes by Setting Descriptors**

18781 Many statement attributes correspond to a header field of one or more descriptors. These
18782 attributes may be set not only by a call to *SQLSetStmtAttr*(), but also by a call to
18783 *SQLSetDescField*(). Setting these options by a call to *SQLSetStmtAttr*(), rather than
18784 *SQLSetDescField*(), has the advantage that a descriptor handle does not have to be obtained first. |

18785 **Caution:** Calling *SQLSetStmtAttr*() for one statement affects other statements if the APD or
18786 ARD associated with the statement is explicitly allocated and is also associated with other
18787 statements. Any modifications made to a descriptor with *SQLSetStmtAttr*() apply to all
18788 statements with which the descriptor is associated. To prevent this effect, the application must |
18789 dissociate this descriptor from the other statements before calling *SQLSetStmtAttr*().

18790 When a statement attribute that is also a descriptor field is set by a call to *SQLSetStmtAttr*(), the
18791 corresponding field in the descriptor that is associated with the statement is also set. The field is
18792 set only for the applicable descriptors that are currently associated with the statement identified
18793 by *StatementHandle*, and the attribute setting does not affect any descriptors that may be
18794 associated with that statement in the future. When a descriptor field that is also a statement
18795 attribute is set by a call to *SQLSetDescField*(), the corresponding statement attribute is also set.

18796 When a statement is allocated (see *SQLAllocHandle*()), four descriptor handles are automatically |
18797 allocated and associated with the statement. Explicitly-allocated descriptor handles can be
18798 associated with the statement by calling *SQLAllocHandle*() with an fHandleType of
18799 SQL_HANDLE_DESC to allocate a descriptor handle, then calling *SQLSetStmtAttr*() to associate
18800 the descriptor handle with the statement.

18801 The following statement attributes correspond to descriptor header fields:

	Statement Attribute	Header Field	Desc.
18802			
18803	SQL_ATTR_PARAM_BIND_OFFSET_PTR	SQL_DESC_BIND_OFFSET_PTR	APD
18804	SQL_ATTR_PARAM_BIND_TYPE	SQL_DESC_BIND_TYPE	APD
18805	SQL_ATTR_PARAM_OPERATION_PTR	SQL_DESC_ARRAY_STATUS_PTR	APD
18806	SQL_ATTR_PARAM_STATUS_PTR	SQL_DESC_ARRAY_STATUS_PTR	IPD
18807	SQL_ATTR_PARAMS_PROCESSED_PTR	SQL_DESC_ROWS_PROCESSED_PTR	IPD
18808	SQL_ATTR_PARAMSET_SIZE	SQL_DESC_ARRAY_SIZE	APD

SQL_ATTR_ROW_ARRAY_SIZE	SQL_DESC_ARRAY_SIZE	ARD
SQL_ATTR_ROW_BIND_OFFSET_PTR	SQL_DESC_BIND_OFFSET_PTR	ARD
SQL_ATTR_ROW_BIND_TYPE	SQL_DESC_BIND_TYPE	ARD
SQL_ATTR_ROW_OPERATION_PTR	SQL_DESC_ARRAY_STATUS_PTR	ARD
SQL_ATTR_ROW_STATUS_PTR	SQL_DESC_ARRAY_STATUS_PTR	IRD
SQL_ATTR_ROWS_FETCHED_PTR	SQL_DESC_ROWS_PROCESSED_PTR	IRD

Statement Attributes

The defined statement attributes are listed below; implementors are likely to define additional attributes to take advantage of different data sources. A range of attributes is reserved by XDBC; implementors must reserve values for vendor-specific uses from X/Open (see Section 1.8 on page 21).

SQL_ATTR_APP_PARAM_DESC

The handle to the APD for subsequent call to *SQLExecute*() and *SQLExecDirect*() on the statement handle. The initial value of this attribute is the descriptor implicitly allocated when the statement was initially allocated. If the value of this attribute is set to SQL_NULL_DESC, or to the handle originally allocated for the descriptor, then an explicitly allocated APD handle that was previously associated with the statement handle is dissociated from it, and the statement handle reverts to the implicitly allocated APD handle.

This attribute cannot be set to a descriptor handle that was implicitly allocated for another statement or to another descriptor handle that was implicitly set on the same statement; implicitly-allocated descriptor handles cannot be associated with more than one statement or descriptor handle.

This attribute cannot be set at the connection level.

SQL_ATTR_APP_ROW_DESC

The handle to the ARD for subsequent fetches on the statement handle. The initial value of this attribute is the descriptor implicitly allocated when the statement was initially allocated. If the value of this attribute is set to SQL_NULL_DESC, or to the handle originally allocated for the descriptor, then an explicitly-allocated ARD handle that was previously associated with the statement handle is dissociated from it, and the statement handle reverts to the implicitly-allocated ARD handle.

This attribute cannot be set to a descriptor handle that was implicitly allocated for another statement or to another descriptor handle that was implicitly set on the same statement; implicitly-allocated descriptor handles cannot be associated with more than one statement or descriptor handle.

This attribute cannot be set at the connection level.

SQL_ATTR_ASYNC_ENABLE

A 32-bit integer value that specifies whether a function called with the specified statement is executed asynchronously:

SQL_ASYNC_ENABLE_OFF	Asynchrony is disabled (the default)
SQL_ASYNC_ENABLE_ON	Asynchrony is enabled

Asynchronous execution provides that certain XDBC functions return before the operation is complete. See Section 9.5 on page 116.

The application can determine the implementation's level of support for asynchrony by calling *SQLGetInfo*() with the SQL_ASYNC_MODE option.

18853 • For implementations with statement-level asynchronous-execution support,
18854 applications can set SQL_ATTR_ASYNC_ENABLE by calling *SQLSetStmtAttr*(), and can
18855 specify a default value for statement handles by treating it as a connection attribute (see
18856 **SQLSetConnectAttr() and Statement Attributes** on page 458).

18857 • For implementations with connection-level asynchronous-execution support,
18858 applications enable and disable asynchrony only by calling *SQLSetConnectAttr*(); as a
18859 statement attribute, SQL_ATTR_ASYNC_ENABLE is read-only, its value indicates the
18860 status of asynchrony on the connection, and trying to change its value by calling
18861 *SQLSetStmtAttr*() returns SQLSTATE HYC00 (Optional feature not implemented).

18862 SQL_ATTR_CONCURRENCY
18863 A 32-bit integer value that specifies the cursor concurrency:

18864 SQL_CONCUR_READ_ONLY Cursor is read-only. No updates are allowed. This
18865 is the default value.

18866 SQL_CONCUR_LOCK Cursor uses the lowest level of locking sufficient to
18867 ensure that the row can be updated.

18868 SQL_CONCUR_ROWVER Cursor uses optimistic concurrency control,
18869 comparing row versions.

18870 SQL_CONCUR_VALUES Cursor uses optimistic concurrency control,
18871 comparing values.

18872 If the SQL_ATTR_CURSOR_TYPE attribute is changed to a type that does not support the
18873 current value of SQL_ATTR_CONCURRENCY, the value of SQL_ATTR_CONCURRENCY
18874 is changed at execution time, and a warning is issued when *SQLExecDirect*() or
18875 *SQLPrepare*() is called.

18876 If the data source supports the SELECT FOR UPDATE statement, and such a statement is
18877 executed while the value of SQL_ATTR_CONCURRENCY is set to
18878 SQL_CONCUR_READ_ONLY, an error is returned. If the value of
18879 SQL_ATTR_CONCURRENCY is changed to a value that the data source supports for some
18880 value of SQL_ATTR_CURSOR_TYPE, but not for the current value of
18881 SQL_ATTR_CURSOR_TYPE, the value of SQL_ATTR_CURSOR_TYPE is changed at
18882 execution time, and SQLSTATE 01S02 (Attribute value changed) is issued when
18883 *SQLExecDirect*() or *SQLPrepare*() is called.

18884 If the specified concurrency is not supported by the data source, the data source substitutes
18885 a different concurrency and returns SQLSTATE 01S02 (Attribute value changed). For
18886 SQL_CONCUR_VALUES, the implementation substitutes SQL_CONCUR_ROWVER, and
18887 vice versa. For SQL_CONCUR_LOCK, the implementation substitutes, in order,
18888 SQL_CONCUR_ROWVER or SQL_CONCUR_VALUES. The validity of the substituted
18889 value is not checked until execution time.

18890 SQL_ATTR_CURSOR_TYPE
18891 A 32-bit integer value that specifies the cursor type:

18892 SQL_CURSOR_FORWARD_ONLY
18893 The cursor only scrolls forward.

18894 SQL_CURSOR_STATIC
18895 The data in the result set is static.

18896 SQL_CURSOR_KEYSET_DRIVEN
18897 The data source saves and uses the keys for the number of rows specified in the
18898 SQL_KEYSET_SIZE statement attribute.

18899 SQL_CURSOR_DYNAMIC
18900 The data source only saves and uses the keys for the rows in the row-set.

18901 The default value is SQL_CURSOR_FORWARD_ONLY. This option cannot be specified
18902 once the statement has been prepared.

18903 If the data source does not support the specified cursor type, the implementation
18904 substitutes a different cursor type and returns SQLSTATE 01S02 (Attribute value changed).
18905 For a mixed or dynamic cursor, the implementation substitutes, in order, a keyset-driven or
18906 static cursor. For a keyset-driven cursor, the implementation substitutes a static cursor.

18907 SQL_ATTR_ENABLE_AUTO_IPD
18908 A 32-bit integer value that specifies whether automatic population of the IPD is performed:

18909 SQL_TRUE Enables automatic population of the IPD after a call to *SQLPrepare*().

18910 SQL_FALSE Disables automatic population of the IPD after a call to *SQLPrepare*(). The
18911 application can still obtain this information, on implementations that
18912 support IPD population, by an explicit call to *SQLDescribeParam*().

18913 The default value of the statement attribute SQL_ATTR_ENABLE_AUTO_IPD is equal to
18914 the value of the connection attribute SQL_ATTR_AUTO_IPD. If the connection attribute
18915 SQL_ATTR_AUTO_IPD is SQL_FALSE, the statement attribute
18916 SQL_ATTR_ENABLE_AUTO_IPD cannot be set to SQL_TRUE.

18917 SQL_ATTR_FETCH_BOOKMARK_PTR
18918 A pointer that points to a binary bookmark value. When *SQLFetchScroll*() is called with
18919 *FetchOrientation* equal to SQL_FETCH_BOOKMARK, the data source uses the bookmark
18920 value from this attribute. The default value is a null pointer.

18921 The value pointed to by this field is not used for delete by bookmark, update by bookmark,
18922 or fetch by bookmark operations in *SQLBulkOperations*(), which use bookmarks cached in
18923 row-set buffers.

18924 SQL_ATTR_IMP_PARAM_DESC
18925 The handle to the IPD. The value of this attribute is the descriptor allocated when the
18926 statement was initially allocated. The application cannot set this attribute.

18927 This attribute can be retrieved by a call to *SQLGetStmtAttr*(), but not set by a call to
18928 *SQLSetStmtAttr*().

18929 SQL_ATTR_IMP_ROW_DESC
18930 The handle to the IRD. The value of this attribute is the descriptor allocated when the
18931 statement was initially allocated. The application cannot set this attribute.

18932 This attribute can be retrieved by a call to *SQLGetStmtAttr*(), but not set by a call to
18933 *SQLSetStmtAttr*().

18934 SQL_ATTR_KEYSET_SIZE
18935 A 32-bit integer value that specifies the number of rows in the keyset for a keyset-driven
18936 cursor. If the keyset size is 0 (the default), the cursor is fully keyset-driven. If the keyset size
18937 is greater than 0, the cursor is mixed (keyset-driven within the keyset and dynamic outside
18938 of the keyset). The default keyset size is 0.

18939 If the specified size exceeds the maximum keyset size, the implementation substitutes that
18940 size and returns SQLSTATE 01S02 (Attribute value changed).

18941 *SQLFetchScroll*() returns an error if the keyset size is greater than 0 and less than the row-set
18942 size.

18943 SQL_ATTR_MAX_LENGTH
18944 A 32-bit integer value that specifies the maximum amount of data that the implementation
18945 returns from a character or binary column. If **ValuePtr* is less than the length of the available
18946 data, *SQLFetch*() or *SQLGetData*() truncates the data and returns SQL_SUCCESS. If
18947 **ValuePtr* is 0 (the default), the implementation tries to return all available data.

18948 If the specified length is less than the minimum amount of data that the data source can
18949 return, or greater than the maximum amount of data that the data source can return, the
18950 implementation substitutes that value and returns SQLSTATE 01S02 (Attribute value
18951 changed).

18952 This attribute is intended to reduce network traffic and should only be supported in
18953 situations when the data source can actually reduce the size of its response based on the
18954 attribute. Applications should not use this attribute merely to force the implementation to
18955 truncate strings. A better way to truncate strings is to specify a low value for the maximum
18956 buffer length in *BufferLength* for calls to *SQLBindCol*() or *SQLGetData*().

18957 SQL_ATTR_MAX_ROWS
18958 A 32-bit integer value corresponding to the maximum number of rows to return to the
18959 application for a SELECT statement. If **ValuePtr* equals 0 (the default), then the data source
18960 returns all rows.

18961 This option is intended to reduce network traffic. Conceptually, it is applied when the result
18962 set is created and limits the result set to the first **ValuePtr* rows. If the number of rows in the
18963 result set is greater than **ValuePtr*, the result set is truncated.

18964 SQL_ATTR_MAX_ROWS applies to all result sets on the *Statement*, including those
18965 returned by catalog functions. SQL_ATTR_MAX_ROWS establishes a maximum value for
18966 the cursor row count.

18967 It is implementation-defined whether SQL_ATTR_MAX_ROWS applies to statements other
18968 than SELECT statements.

18969 SQL_ATTR_METADATA_ID
18970 A 32-bit integer value that determines how the string arguments of catalog functions are
18971 treated.

18972 If SQL_TRUE, the string argument of catalog functions are treated as identifiers. The case if
18973 not significant. For non-delimited strings, the implementation removes any trailing spaces,
18974 and the string is folded to upper case. For delimited strings, the implementation removes
18975 any leading or trailing spaces, and interprets the remainder literally. If one of these
18976 arguments is set to a null pointer, the function returns SQL_ERROR and SQLSTATE HY009
18977 (Invalid use of null pointer).

18978 If SQL_FALSE, the string arguments of catalog functions are not treated as identifiers. The
18979 case is significant. They can either contain a string search pattern or not, depending on the
18980 argument.

18981 The default value is SQL_FALSE.

18982 The *TableType* argument of *SQLTables*(), which takes a list of values, is not affected by this
18983 attribute.

18984 (For more information, see Section 7.4 on page 69.)

18985 SQL_ATTR_NOSCAN
18986 A 32-bit integer value that controls scanning for the XDBC escape clauses defined in Section
18987 8.3 on page 84:

18988 SQL_NOSCAN_OFF The implementation scans SQL strings for escape clauses and
18989 converts them to the SQL dialect that the data source accepts (the
18990 default).

18991 SQL_NOSCAN_ON The implementation does not scan SQL strings for escape clauses.
18992 Instead, it sends the SQL statement directly to the data source.

18993 When it is certain that no SQL statement contains escape clauses,
18994 the application can specify this value to eliminate this processing.
18995 This may boost performance.

18996 SQL_ATTR_PARAM_BIND_OFFSET_PTR
18997 A SQLINTEGER * value that points to the bind offset. Setting this statement attribute sets
18998 the SQL_DESC_BIND_OFFSET_PTR field in the APD header.

18999 If this attribute is not a null pointer, the bind offset is added to each deferred field in the
19000 descriptor record (SQL_DESC_DATA_PTR, SQL_DESC_INDICATOR_PTR, and
19001 SQL_DESC_OCTET_LENGTH_PTR) to produce the effective address for the fetch. The
19002 bind offset is not cumulative; if the value is changed, any old bind offset ceases to have
19003 effect. A bind offset can be used only with row-wise binding. See Section 9.4 on page 102
19004 and **Bind Offsets** on page 217.

19005 SQL_ATTR_PARAM_BIND_TYPE
19006 A 32-bit integer value that indicates the binding orientation to be used for dynamic
19007 parameters.

19008 This field is set to SQL_PARAMETER_BIND_BY_COLUMN (the default) to select column-
19009 wise binding.

19010 To select row-wise binding, this field is set to the length of the structure or an instance of a
19011 buffer that will be bound to a set of dynamic parameters. This length must include space
19012 for all of the bound parameters and any padding of the structure or buffer to ensure that
19013 when the address of a bound parameter is incremented with the specified length, the result
19014 will point to the beginning of the same parameter in the next set of parameters. When using
19015 the **sizeof** operator in ANSI C, this behavior is guaranteed.

19016 Setting this statement attribute sets the SQL_DESC_BIND_TYPE field in the APD header.

19017 SQL_ATTR_PARAM_OPERATION_PTR
19018 A SQLUSMALLINT * value that points to an array of SQLUSMALLINT values used to
19019 ignore a parameter during execution of a SQL statement. Each value is set to either
19020 SQL_PARAM_PROCEED (to execute a parameter) or SQL_PARAM_IGNORE (to ignore a
19021 parameter) as described in **Ignoring a Set of Parameters** on page 230.

19022 This statement attribute can be set to a null pointer, in which case the implementation acts
19023 as though every element were set to SQL_PARAM_PROCEED. This attribute can be set at
19024 any time, but the new value is not used until the next call to *SQLExecDirect*() or
19025 *SQLExecute*().

19026 Setting this statement attribute sets the SQL_DESC_ARRAY_STATUS_PTRfield in the APD.

19027 SQL_ATTR_PARAM_STATUS_PTR
19028 A SQLUSMALLINT * value that points to an array of SQLUSMALLINT values containing
19029 status information for each row of parameter values after a call to *SQLExecute*() or
19030 *SQLExecDirect*(). This field is required only if PARAMSET_SIZE is greater than 1.

19031 The status values can contain the following values:

19032 SQL_PARAM_SUCCESS
19033 The SQL statement was successfully executed for this set of parameters.

19034 SQL_PARAM_SUCCESS_WITH_INFO
19035 The SQL statement was successfully executed for this set of parameters; however,
19036 warning information is available in the diagnostics data structure.

19037 SQL_PARAM_ERROR
19038 There was an error in processing this set of parameters. Additional error information is
19039 available in the diagnostics data structure.

19040 SQL_PARAM_UNUSED
19041 This parameter set was unused, possibly due to the fact that some previous parameter
19042 set caused an error that aborted further processing, or because SQL_PARAM_IGNORE
19043 was set for that set of parameters in the array specified by the
19044 SQL_ATTR_PARAM_OPERATION_PTR.

19045 SQL_PARAM_DIAG_UNAVAILABLE
19046 The implementation treats arrays of parameters as a monolithic unit and so does not
19047 generate this level of error information.

19048 This statement attribute can be set to a null pointer, in which case the implementation does
19049 not return parameter status values. This attribute can be set at any time, but the new value
19050 is not used until the next call to *SQLFetch*() or *SQLFetchScroll*().

19051 Setting this statement attribute sets the SQL_DESC_ARRAY_STATUS_PTR field in the IPD
19052 header.

19053 SQL_ATTR_PARAMS_PROCESSED_PTR
19054 A SQLUINTEGER * record field that points to a buffer in which to return the number of sets
19055 of parameters that have been processed, including error sets. No number is returned if this
19056 is a null pointer.

19057 Setting this statement attribute sets the SQL_DESC_ROWS_PROCESSED_PTR field in the
19058 IPD header.

19059 If the call to *SQLExecDirect*() or *SQLExecute*() that fills in the buffer pointed to by this
19060 attribute does not return SQL_SUCCESS or SQL_SUCCESS_WITH_INFO, the contents of
19061 the buffer are undefined.

19062 SQL_ATTR_PARAMSET_SIZE
19063 A SQLUINTEGER value that specifies the number of values for each parameter. If
19064 SQL_ATTR_PARAMSET_SIZE is greater than 1, SQL_DESC_DATA_PTR,
19065 SQL_DESC_INDICATOR_PTR, and SQL_DESC_OCTET_LENGTH_PTR of the APD point
19066 to arrays. The cardinality of each array is equal to the value of this field.

19067 Setting this statement attribute sets the SQL_DESC_ARRAY_SIZE field in the APD header.

19068 SQL_ATTR_QUERY_TIMEOUT
19069 A 32-bit integer value corresponding to the number of seconds to wait for an SQL statement
19070 to execute before returning to the application. If **ValuePtr* equals 0 (default), then there is no
19071 timeout.

19072 If the specified timeout exceeds the maximum timeout in the data source or is smaller than
19073 the minimum timeout, *SQLSetStmtAttr*() substitutes that value and returns SQLSTATE
19074 01S02 (Attribute value changed).

19075 The application need not call *SQLCloseCursor*() to reuse the statement if a SELECT
19076 statement timed out.

19077 The query timeout set in this statement attribute is valid in both synchronous and
19078 asynchronous modes.

19079 SQL_ATTR_RETRIEVE_DATA
19080 A 32-bit integer value which is one of the following:

19081 SQL_RD_ON
19082 *SQLFetch*() and *SQLFetchScroll*() retrieve data after they position the cursor to the
19083 specified location. This is the default.

19084 SQL_RD_OFF
19085 *SQLFetch*() and *SQLFetchScroll*() do not retrieve data after they position the cursor.

19086 By setting SQL_RETRIEVE_DATA to SQL_RD_OFF, an application can verify that a row |
19087 exists or retrieve a bookmark for the row without incurring the overhead of retrieving rows.

19088 SQL_ATTR_ROW_ARRAY_SIZE
19089 A 32-bit integer value that specifies the number of rows returned by each call to *SQLFetch*() |
19090 or *SQLFetchScroll*(). This is also the number of rows in a bookmark array used in a bulk |
19091 bookmark operation in *SQLBulkOperations*(). The default value is 1.

19092 If the specified row-set size exceeds the maximum row-set size supported by the data |
19093 source, the implementation substitutes that value and returns SQLSTATE 01S02 (Attribute
19094 value changed). |

19095 Setting this statement attribute sets the SQL_DESC_ARRAY_SIZE field in the ARD header.

19096 SQL_ATTR_ROW_BIND_OFFSET_PTR
19097 A SQLINTEGER * value that points to an offset added to pointers to change binding of |
19098 column data. If this field is non-null, the implementation dereferences the pointer, adds the |
19099 dereferenced value to each of the deferred fields in the descriptor record |
19100 (SQL_DESC_DATA_PTR, SQL_DESC_INDICATOR_PTR, and |
19101 SQL_DESC_OCTET_LENGTH_PTR), and uses the new pointer values when binding. It is |
19102 set to null by default.

19103 Setting this statement attribute sets the SQL_DESC_BIND_OFFSET_PTR field in the ARD
19104 header.

19105 SQL_ATTR_ROW_BIND_TYPE
19106 A 32-bit integer value that sets the binding orientation to be used when *SQLFetch*() or
19107 *SQLFetchScroll*() is called on the associated statement. Column-wise binding is selected by
19108 supplying the defined constant SQL_BIND_BY_COLUMN in **ValuePtr*. Row-wise binding
19109 is selected by supplying a value in **ValuePtr* specifying the length of a structure or an
19110 instance of a buffer into which result columns will be bound.

19111 The length specified in **ValuePtr* must include space for all of the bound columns and any
19112 padding of the structure or buffer to ensure that when the address of a bound column is
19113 incremented with the specified length, the result points to the beginning of the same column
19114 in the next row. When using the **sizeof** operator with structures or unions in ANSI C, this
19115 behavior is guaranteed.

19116 Column-wise binding is the default binding orientation for *SQLFetch*() and *SQLFetchScroll*().

19117 Setting this statement attribute sets the SQL_DESC_BIND_TYPE field in the ARD header.

19118 SQL_ATTR_ROW_NUMBER
19119 A SQLINTEGER value that is the number of the current row in the entire result set. If the
19120 number of the current row cannot be determined or there is no current row, the |
19121 implementation returns 0.

19122 This attribute can be retrieved by a call to *SQLGetStmtAttr*(), but not set by a call to
19123 *SQLSetStmtAttr*().

19124 SQL_ATTR_ROW_OPERATION_PTR
19125 A SQLUSMALLINT * value that points to an array of SQLUINTEGER values used to ignore
19126 a row during a bulk operation using *SQLBulkOperations*() or *SQLSetPos*(). Each value is set
19127 to either SQL_ROW_PROCEED (for the row to be included in the bulk operation) or
19128 SQL_ROW_IGNORE (for the row to be excluded from the bulk operation).

19129 This statement attribute can be set to a null pointer, in which case the implementation does
19130 not return row status values. This attribute can be set at any time, but the new value is not
19131 used until the next call to *SQLBulkOperations*() or *SQLSetPos*().

19132 Setting this statement attribute sets the SQL_DESC_ARRAY_STATUS_PTRfield in the ARD.

19133 SQL_ATTR_ROW_STATUS_PTR
19134 A SQLUSMALLINT * value that points to an array of SQLUINTEGER values containing
19135 row status values after a call to *SQLFetch*() or *SQLFetchScroll*(). The array has as many
19136 elements as there are rows in the row-set.

19137 This statement attribute can be set to a null pointer, in which case the implementation acts
19138 as though every element were set to SQL_ROW_PROCEED. This attribute can be set at any
19139 time, but the new value is not used until the next call to *SQLBulkOperations*(), *SQLFetch*(),
19140 *SQLFetchScroll*(), or *SQLSetPos*().

19141 Setting this statement attribute sets the SQL_DESC_ARRAY_STATUS_PTRfield in the IRD
19142 header.

19143 SQL_ATTR_ROWS_FETCHED_PTR
19144 A SQLUINTEGER * value that points to a buffer in which to return the number of rows
19145 fetched after a call to *SQLFetch*() or *SQLFetchScroll*(), or the number of rows affected by a
19146 bulk operation performed by a call to *SQLSetPos*() with an *Operation* of SQL_REFRESH.
19147 This number includes error rows.

19148 Setting this statement attribute sets the SQL_DESC_ROWS_PROCESSED_PTR field in the
19149 IRD header.

19150 If the call to *SQLFetch*() or *SQLFetchScroll*() that fills in the buffer pointed to by this attribute
19151 does not return SQL_SUCCESS or SQL_SUCCESS_WITH_INFO, the contents of the buffer
19152 are undefined.

19153 SQL_ATTR_SIMULATE_CURSOR
19154 A 32-bit integer value that specifies whether the implementation, in simulating positioned
19155 UPDATE and DELETE statements, must guarantee that such statements affect only one
19156 row. If the data source has native support for these statements, the implementation does
19157 guarantee this, and returns SQL_SC_UNIQUE (see below).

19158 To simulate positioned UPDATE and DELETE statements for data sources that do not
19159 support these statements, the implementation typically constructs a searched UPDATE or
19160 DELETE statement containing a WHERE clause that specifies the value of each column in
19161 the current row.

19162 Unless these columns form a unique key, such a statement may affect more than one row.
19163 To guarantee that such statements affect only one row, the implementation determines
19164 which columns form a unique key and adds these columns to the result set it requests.

19165 The application can use SQL_ATTR_SIMULATE_CURSOR to indicate to the
19166 implementation that the columns in the result set the application has requested are a unique
19167 key. This means the implementation does not have to make the guarantee, which may
19168 reduce execution time.

19169 The attribute has one of the following values:

19170 SQL_SC_NON_UNIQUE
19171 The implementation does not guarantee that simulated positioned UPDATE and
19172 DELETE statements affect only one row; the application must do so. If a statement
19173 affects more than one row, *SQLExecute*(), *SQLExecDirect*(), or *SQLSetPos*() returns
19174 SQLSTATE01001 (Cursor operation conflict).

19175 SQL_SC_TRY_UNIQUE
19176 The implementation tries to guarantee that simulated positioned UPDATE and
19177 DELETE statements affect only one row. The statements are always executed, even if
19178 they might affect more than one row, such as when there is no unique key. If a
19179 statement affects more than one row, *SQLExecute*(), *SQLExecDirect*(), or *SQLSetPos*()
19180 returns SQLSTATE01001 (Cursor operation conflict).

19181 SQL_SC_UNIQUE
19182 The implementation guarantees that simulated positioned UPDATE and DELETE
19183 statements affect only one row. If it cannot guarantee this for a given statement,
19184 *SQLExecDirect*() or *SQLPrepare*() returns an error.

19185 If the data source provides native SQL support for positioned UPDATE and DELETE
19186 statements, and the implementation does not simulate cursors, SQL_SUCCESS is
19187 returned when SQL_SC_UNIQUE is requested for SQL_ATTR_SIMULATE_CURSOR.
19188 SQL_SUCCESS_WITH_INFO is returned if SQL_SC_TRY_UNIQUE or
19189 SQL_SC_NON_UNIQUE is requested. If the data source provides the
19190 SQL_SC_TRY_UNIQUE level of support, and the implementation does not, it returns
19191 SQL_SUCCESS for SQL_SC_TRY_UNIQUE and SQL_SUCCESS_WITH_INFO for
19192 SQL_SC_NON_UNIQUE.

19193 To determine what the implementation supports, and therefore the valid values for this
19194 statement attribute, the application calls *SQLGetInfo*() as described in **Detecting Cursor
19195 Capabilities with SQLGetInfo()** on page 402 and tests the bitmasks
19196 SQL_CA2_SIMULATE_NON_UNIQUE, SQL_CA2_SIMULATE_TRY_UNIQUE, and
19197 SQL_CA2_SIMULATE_UNIQUE. If the data source does not support the specified cursor
19198 simulation type, the implementation substitutes a different simulation type and returns
19199 SQLSTATE 01S02 (Attribute value changed). For SQL_SC_UNIQUE, the implementation
19200 substitutes, in order, SQL_SC_TRY_UNIQUE or SQL_SC_NON_UNIQUE. For
19201 SQL_SC_TRY_UNIQUE, the implementation substitutes SQL_SC_NON_UNIQUE.

19202 SQL_ATTR_USE_BOOKMARKS
19203 A 32-bit integer value that specifies whether an application will use bookmarks with a
19204 cursor:

19205 SQL_UB_OFF
19206 Off (the default)

19207 SQL_UB_VARIABLE
19208 An application will use bookmarks with a cursor. Bookmarks in XDBC are variable-
19209 length data structures.

19210 To use bookmarks with a cursor, the application must set this attribute to
19211 SQL_UB_VARIABLEbefore opening the cursor.

19212 **SEE ALSO**

19213 **For information about** **See**

Canceling statement processing	*SQLCancel*()
Returning the setting of a connection attribute	*SQLGetConnectAttr*()
Returning the setting of a statement attribute	*SQLGetStmtAttr*()
Setting a connection attribute	*SQLSetConnectAttr*()
Setting a single field of the descriptor	*SQLSetDescField*()

CHANGE HISTORY

Version 2

Revised generally. See **Alignment with Popular Implementations** on page 2.

New Statement Attributes in Version 2

The following statement attributes are new in this issue:

SQL_ATTR_ASYNC_ENABLE	SQL_ATTR_PARAMS_PROCESSED_PTR
SQL_ATTR_CONCURRENCY	SQL_ATTR_PARAMSET_SIZE
SQL_ATTR_CURSOR_TYPE	SQL_ATTR_QUERY_TIMEOUT
SQL_ATTR_ENABLE_AUTO_IPD	SQL_ATTR_RETRIEVE_DATA
SQL_ATTR_FETCH_BOOKMARK_PTR	SQL_ATTR_ROW_ARRAY_SIZE
SQL_ATTR_KEYSET_SIZE	SQL_ATTR_ROW_BIND_OFFSET_PTR
SQL_ATTR_MAX_LENGTH	SQL_ATTR_ROW_BIND_TYPE
SQL_ATTR_MAX_ROWS	SQL_ATTR_ROW_NUMBER
SQL_ATTR_NOSCAN	SQL_ATTR_ROW_OPERATION_PTR
SQL_ATTR_PARAM_BIND_OFFSET_PTR	SQL_ATTR_ROW_STATUS_PTR
SQL_ATTR_PARAM_BIND_TYPE	SQL_ATTR_ROWS_FETCHED_PTR
SQL_ATTR_PARAM_OPERATION_PTR	SQL_ATTR_SIMULATE_CURSOR
SQL_ATTR_PARAM_STATUS_PTR	SQL_ATTR_USE_BOOKMARKS

19237 **NAME**

19238 SQLSpecialColumns — Retrieve information about row-identifying columns of a table.

19239 **SYNOPSIS**

```
19240 SQLRETURN SQLSpecialColumns(
19241   SQLHSTMT StatementHandle,
19242   SQLSMALLINT IdentifierType,
19243   SQLCHAR * CatalogName,
19244   SQLSMALLINT NameLength1,
19245   SQLCHAR * SchemaName,
19246   SQLSMALLINT NameLength2,
19247   SQLCHAR * TableName,
19248   SQLSMALLINT NameLength3,
19249   SQLSMALLINT Scope,
19250   SQLSMALLINT Nullable);
```

19251 **ARGUMENTS**

19252 *StatementHandle* [Input]

19253 Statement handle.

19254 *IdentifierType* [Input]

19255 Type of column to return. Must be one of the following values:

19256 SQL_BEST_ROWID Returns the optimal column or set of columns that, by retrieving
19257 values from the column or columns, serves to uniquely identify any
19258 row in the specified table. The result can be either a pseudo-
19259 column specifically designed for this purpose, or the column or
19260 columns of any unique index for the table.

19261 SQL_ROWVER Returns the column or columns in the specified table, if any, that
19262 are automatically updated by the data source when any value in
19263 the row is updated by any transaction.

19264 *CatalogName* [Input]

19265 Catalog name for the table. If the data source supports catalogs, an empty string denotes
19266 those tables that do not have catalogs.

19267 If the SQL_ATTR_METADATA_ID statement attribute is SQL_TRUE, this argument is
19268 interpreted as specified in **Identifier (ID) Arguments** on page 72. If it is SQL_FALSE, this
19269 argument is interpreted as specified in **Ordinary Arguments (OA)** on page 71.

19270 *NameLength1* [Input]

19271 Length of **CatalogName*.

19272 *SchemaName* [Input]

19273 Schema name for the table. If the data source supports schemas, an empty string denotes
19274 those tables that do not have schemas.

19275 If the SQL_ATTR_METADATA_ID statement attribute is SQL_TRUE, this argument is
19276 interpreted as specified in **Identifier (ID) Arguments** on page 72. If it is SQL_FALSE, this
19277 argument is interpreted as specified in **Ordinary Arguments (OA)** on page 71.

19278 *NameLength2* [Input]

19279 Length of **SchemaName*.

19280 *TableName* [Input]

19281 Table name. This argument cannot be a null pointer.

19282 If the SQL_ATTR_METADATA_ID statement attribute is SQL_TRUE, this argument is
19283 interpreted as specified in **Identifier (ID) Arguments** on page 72. If it is SQL_FALSE, this
19284 argument is interpreted as specified in **Ordinary Arguments (OA)** on page 71.

19285 *NameLength3* [Input]
19286 Length of **TableName*.

19287 *Scope* [Input]
19288 Minimum required scope of the rowid. The returned rowid may be of greater scope. Must
19289 be one of the following:

19290 SQL_SCOPE_CURROW
19291 The rowid is guaranteed to be valid only while positioned on that row. A later reselect
19292 using rowid may not return a row if the row was updated or deleted by another
19293 transaction.

19294 SQL_SCOPE_TRANSACTION
19295 The rowid is guaranteed to be valid for the duration of the current transaction.

19296 SQL_SCOPE_SESSION
19297 The rowid is guaranteed to be valid for the duration of the session (across transaction
19298 boundaries).

19299 *Nullable* [Input]
19300 Determines whether to return special columns that can have a NULL value. Must be one of
19301 the following:

19302 SQL_NO_NULLS
19303 Exclude special columns that can have NULL values. Some implementations return an
19304 empty result set if SQL_NO_NULLS is specified.

19305 SQL_NULLABLE
19306 Return special columns even if they can have NULL values.

19307 **RETURN VALUE**
19308 SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or
19309 SQL_INVALID_HANDLE.

19310 **DIAGNOSTICS**
19311 When *SQLSpecialColumns*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated
19312 SQLSTATE value can be obtained by calling *SQLGetDiagRec*() with a *HandleType* of
19313 SQL_HANDLE_STMT and a *Handle* of *StatementHandle*. The following table lists the SQLSTATE
19314 values commonly returned by *SQLSpecialColumns*(). The return code associated with each
19315 SQLSTATE value is SQL_ERROR, except that for SQLSTATE values in class 01, the return code is
19316 SQL_SUCCESS_WITH_INFO.

19317 01000 — General warning
19318 Implementation-defined informational message.

19319 08S01 — Communication link failure
19320 The communication link to the data source failed before the function completed processing.

19321 24000 — Invalid cursor state
19322 A cursor was open on *StatementHandle*.

19323 HY000 — General error
19324 An error occurred for which there was no specific SQLSTATE and for which no
19325 implementation-specific SQLSTATE was defined. The error message returned by
19326 *SQLGetDiagRec*() in the **MessageText* buffer describes the error and its cause.

19327 HY001 — Memory allocation error
19328 The implementation failed to allocate memory required to support execution or completion
19329 of the function.

19330 HY008 — Operation canceled
19331 Asynchronous processing was enabled for *StatementHandle*. The function was called and
19332 before it completed execution, *SQLCancel*() was called on *StatementHandle*. The function
19333 was then called again on *StatementHandle*.

19334 The function was called and, before it completed execution, *SQLCancel*() was called on
19335 *StatementHandle* from a different thread in a multithread application.

19336 HY009 — Invalid use of null pointer
19337 *TableName* was a null pointer.

19338 The SQL_ATTR_METADATA_IDstatement attribute was set to SQL_TRUE, *CatalogName*
19339 was a null pointer, and the SQL_CATALOG_NAME option of *SQLGetInfo*() returns that
19340 catalog names are supported.

19341 The SQL_ATTR_METADATA_ID statement attribute was set to SQL_TRUE, and
19342 *SchemaName* or *TableName* was a null pointer.

19343 HY010 — Function sequence error
19344 An asynchronously executing function (not this one) was called for *StatementHandle* and
19345 was still executing when this function was called.

19346 *SQLBulkOperations*(), *SQLExecDirect*(), *SQLExecute*(), or *SQLSetPos*() was called for
19347 *StatementHandle* and returned SQL_NEED_DATA. This function was called before data was
19348 sent for all data-at-execution parameters or columns.

19349 HY090 — Invalid string or buffer length
19350 The value of one of the length arguments was less than 0, but not equal to SQL_NTS.

19351 The value of one of the length arguments exceeded the maximum length value for the
19352 corresponding name. The maximum length of each name can be obtained by calling
19353 *SQLGetInfo*() with the following options: SQL_MAX_CATALOG_NAME_LEN,
19354 SQL_MAX_SCHEMA_NAME_LEN, or SQL_MAX_TABLE_NAME_LEN.

19355 HY097 — Column type out of range
19356 *IdentifierType* was invalid.

19357 HY098 — Scope type out of range
19358 *Scope* was invalid.

19359 HY099 — Nullable type out of range
19360 *Nullable* was invalid.

19361 HYC00 — Optional feature not implemented
19362 A catalog was specified and the implementation does not support catalogs.

19363 A schema was specified and the implementation does not support schemas.

19364 The data source does not support the combination of the current settings of the
19365 SQL_ATTR_CONCURRENCY and SQL_ATTR_CURSOR_TYPE statement attributes.

19366 The SQL_ATTR_USE_BOOKMARKS statement attribute was set to SQL_UB_VARIABLE,
19367 and the SQL_ATTR_CURSOR_TYPE statement attribute was set to a cursor type for which
19368 the data source does not support bookmarks.

19369 HYT00 — Timeout expired
19370 The query timeout period expired before the data source returned the requested result set.
19371 The timeout period is set through *SQLSetStmtAttr*(), SQL_ATTR_QUERY_TIMEOUT.

HYT01 — Connection timeout expired
: The connection timeout period expired before the data source responded to the request. The connection timeout period is set through *SQLSetConnectAttr*(), SQL_ATTR_CONNECTION_TIMEOUT.

IM001 — Function not supported
: The function is not supported on the current connection to the data source.

COMMENTS

SQLSpecialColumns() retrieves the following information concerning *TableName*:

- The optimal set of columns that uniquely identifies a row in the table.
- Columns that are automatically updated when any value in the row is updated by a transaction.

When *IdentifierType* is SQL_BEST_ROWID, *SQLSpecialColumns*() returns the column or columns that uniquely identify each row in the table. These columns can always be used in a select-list or WHERE clause. This effect cannot be achieved by calling *SQLColumns*(), which does not return data-source-specific pseudo-columns that may be necessary to uniquely identify each row.

If there are no columns that uniquely identify each row in the table, *SQLSpecialColumns*() returns a row-set with no rows; a subsequent call to *SQLFetch*() or *SQLFetchScroll*() on the statement returns SQL_NO_DATA.

If *IdentifierType*, *Scope*, or *Nullable* specify characteristics that are not supported by the data source, *SQLSpecialColumns*() returns an empty result set.

If the SQL_ATTR_METADATA_ID statement attribute is set to SQL_TRUE, the *CatalogName*, *SchemaName*, and *TableName* arguments are treated as identifiers, so cannot be set to a null pointer in certain situations. (For more information, see Section 7.4 on page 69.)

SQLSpecialColumns() returns the results as a standard result set, ordered by SCOPE.

To determine the actual length of the COLUMN_NAME column, an application can call *SQLGetInfo*() with the SQL_MAX_COLUMN_NAME_LEN option.

The following table lists the columns in the result set. Additional columns beyond column 8 (PSEUDO_COLUMN) can be defined by the implementation. An application should gain access to implementation-defined columns by counting down from the end of the result set rather than by specifying an explicit ordinal position; see Section 7.3 on page 68.

Column Name	Col. No.	Data Type	Comments
SCOPE	1	Smallint	The actual scope of this rowid. The valid values and their meanings are the same as those defined for the *Scope* argument. NULL is returned when *IdentifierType* is SQL_ROWVER.

COLUMN_NAME	2	Varchar not NULL	Column identifier. This is an empty string for unnamed columns.
DATA_TYPE	3	Smallint not NULL	SQL data type. This can be an XDBC SQL data type or an implementation-defined data type. For a list of valid XDBC SQL data types, see Section D.1 on page 556.
TYPE_NAME	4	Varchar not NULL	Data source-dependent data type name; for example, ''CHAR'', ''VARCHAR'', ''MONEY'', ''LONG VARBINARY'', or ''CHAR ( ) FOR BIT DATA''.
COLUMN_SIZE	5	Integer	The size of the column on the data source, as defined in Section D.3.1 on page 562.
BUFFER_LENGTH	6	Integer	The length in octets of data transferred on an *SQLGetData*() or *SQLFetch*() operation if SQL_C_DEFAULT is specified. For numeric data, this size may be different from the size of the data stored on the data source. This value is the same as the COLUMN_SIZE column for character or binary data. For more information, see Section D.3 on page 562.
DECIMAL_DIGITS	7	Smallint	The decimal digits of the column on the data source. NULL is returned for data types where decimal digits is not applicable. For more information concerning decimal digits, see Section D.3 on page 562.
PSEUDO_COLUMN	8	Smallint	Indicates whether the column is a pseudo-column: SQL_PC_UNKNOWN SQL_PC_NOT_PSEUDO SQL_PC_PSEUDO Portable applications should not quote the names of pseudo-columns.

Once the application retrieves values for SQL_BEST_ROWID, the application can use these values to reselect that row within the defined scope. Such a SELECT statement returns either no rows or one row.

If an application reselects a row based on the rowid column or columns and the row is not found, then the application can assume that the row was deleted or the rowid columns were modified. The opposite is not true: even if the rowid has not changed, the other columns in the row may have changed.

Columns returned for column type SQL_BEST_ROWID are useful for applications that need to scroll forward and backward within a result set to retrieve the most recent data from a set of rows. The column or columns of the rowid are guaranteed not to change while positioned on that row.

The column or columns of the rowid may remain valid even when the cursor is not positioned on the row; the application can determine this by checking the SCOPE column in the result set.

19455 Columns returned for column type SQL_ROWVER are useful for applications that need the
19456 ability to check if any columns in a given row have been updated while the row was reselected
19457 using the rowid. For example, after reselecting a row using rowid, the application can compare
19458 the previous values in the SQL_ROWVER columns to the ones just fetched. If the value in a
19459 SQL_ROWVER column differs from the previous value, the application can alert the user that
19460 data on the display has changed.

19461 **SEE ALSO**

19462	**For information about**	**See**
19463	Overview of catalog functions	Chapter 7
19464	Binding a buffer to a column in a result set	*SQLBindCol*()
19465	Canceling statement processing	*SQLCancel*()
19466	Returning the columns in a table or tables	*SQLColumns*()
19467	Fetching a block of data or scrolling through a result set	*SQLFetchScroll*()
19468 19469	Fetching a single row or a block of data in a forward-only direction	*SQLFetch*()
19470	Returning the columns of a primary key	*SQLPrimaryKeys*()

19471 **CHANGE HISTORY**

19472 **Version 2**

19473 Revised generally. See **Alignment with Popular Implementations** on page 2.

19474 **NAME**

19475 SQLStatistics — Retrieve as a result set a list of statistics about a single table and the indexes
19476 associated with it.

19477 **SYNOPSIS**

```
19478 SQLRETURN SQLStatistics(
19479   SQLHSTMT StatementHandle,
19480   SQLCHAR * CatalogName,
19481   SQLSMALLINT NameLength1,
19482   SQLCHAR * SchemaName,
19483   SQLSMALLINT NameLength2,
19484   SQLCHAR * TableName,
19485   SQLSMALLINT NameLength3,
19486   SQLUSMALLINT Unique,
19487   SQLUSMALLINT Accuracy);
```

19488 **ARGUMENTS**

19489 *StatementHandle* [Input]
19490 Statement handle.

19491 *CatalogName* [Input]
19492 Catalog name. If the data source supports catalogs, an empty string denotes those tables
19493 that do not have catalogs.

19494 If the SQL_ATTR_METADATA_ID statement attribute is SQL_TRUE, this argument is
19495 interpreted as specified in **Identifier (ID) Arguments** on page 72. If it is SQL_FALSE, this
19496 argument is interpreted as specified in **Ordinary Arguments (OA)** on page 71.

19497 *NameLength1* [Input]
19498 Length of **CatalogName*.

19499 *SchemaName* [Input]
19500 Schema name. If the data source supports schemas, an empty string denotes those tables
19501 that do not have schemas.

19502 If the SQL_ATTR_METADATA_ID statement attribute is SQL_TRUE, this argument is
19503 interpreted as specified in **Identifier (ID) Arguments** on page 72. If it is SQL_FALSE, this
19504 argument is interpreted as specified in **Ordinary Arguments (OA)** on page 71.

19505 *NameLength2* [Input]
19506 Length of **SchemaName*.

19507 *TableName* [Input]
19508 Table name. This argument cannot be a null pointer.

19509 If the SQL_ATTR_METADATA_ID statement attribute is SQL_TRUE, this argument is
19510 interpreted as specified in **Identifier (ID) Arguments** on page 72. If it is SQL_FALSE, this
19511 argument is interpreted as specified in **Ordinary Arguments (OA)** on page 71.

19512 *NameLength3* [Input]
19513 Length of **TableName*.

19514 *Unique* [Input]
19515 Type of index: SQL_INDEX_UNIQUE or SQL_INDEX_ALL.

19516 *Accuracy* [Input]
19517 Indicates the importance of the CARDINALITY and PAGES columns in the result set. The
19518 following options affect the return of the CARDINALITY and PAGES columns only; index
19519 information is returned even if CARDINALITY and PAGES are not returned.

19520 SQL_ENSURE
19521 Directs the implementation to unconditionally retrieve the statistics. It is
19522 implementation-defined whether this option is supported.

19523 SQL_QUICK
19524 Directs the implementation to retrieve the CARDINALITY and PAGE only if they are
19525 readily available from the server. In this case, it is possible that the resulting values are
19526 not current.

19527 **RETURN VALUE**
19528 SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or
19529 SQL_INVALID_HANDLE.

19530 **DIAGNOSTICS**
19531 When *SQLStatistics*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated
19532 SQLSTATE value can be obtained by calling *SQLGetDiagRec*() with a *HandleType* of
19533 SQL_HANDLE_STMT and a *Handle* of *StatementHandle*. The following SQLSTATE values are
19534 commonly returned by *SQLStatistics*(). The return code associated with each SQLSTATE value
19535 is SQL_ERROR, except that for SQLSTATE values in class 01, the return code is
19536 SQL_SUCCESS_WITH_INFO.

19537 01000 — General warning
19538 Implementation-defined informational message.

19539 08S01 — Communication link failure
19540 The communication link to the data source failed before the function completed processing.

19541 24000 — Invalid cursor state
19542 A cursor was open on *StatementHandle*.

19543 HY000 — General error
19544 An error occurred for which there was no specific SQLSTATE and for which no
19545 implementation-specific SQLSTATE was defined. The error message returned by
19546 *SQLGetDiagRec*() in the **MessageText* buffer describes the error and its cause.

19547 HY001 — Memory allocation error
19548 The implementation failed to allocate memory required to support execution or completion
19549 of the function.

19550 HY008 — Operation canceled
19551 Asynchronous processing was enabled for *StatementHandle*. The function was called and
19552 before it completed execution, *SQLCancel*() was called on *StatementHandle*. The function
19553 was then called again on *StatementHandle*.

19554 The function was called and, before it completed execution, *SQLCancel*() was called on
19555 *StatementHandle* from a different thread in a multithread application.

19556 HY009 — Invalid use of null pointer
19557 *TableName* was a null pointer.

19558 The SQL_ATTR_METADATA_ID statement attribute was set to SQL_TRUE, *CatalogName*
19559 was a null pointer, and the SQL_CATALOG_NAME option of *SQLGetInfo*() returns that
19560 catalog names are supported.

19561 The SQL_ATTR_METADATA_ID statement attribute was set to SQL_TRUE, and
19562 *SchemaName* or *TableName* was a null pointer.

19563 HY010 — Function sequence error
19564 An asynchronously executing function (not this one) was called for *StatementHandle* and
19565 was still executing when this function was called.

19566 *SQLBulkOperations*(), *SQLExecDirect*(), *SQLExecute*(), or *SQLSetPos*() was called for
19567 *StatementHandle* and returned SQL_NEED_DATA. This function was called before data was
19568 sent for all data-at-execution parameters or columns.

19569 HY090 — Invalid string or buffer length
19570 The value of one of the name length arguments was less than 0, but not equal to SQL_NTS.

19571 The value of one of the name length arguments exceeded the maximum length value for the
19572 corresponding name.

19573 HY100 — Uniqueness option type out of range
19574 *Unique* was invalid.

19575 HY101 — Accuracy option type out of range
19576 *Accuracy* was invalid.

19577 HYC00 — Optional feature not implemented
19578 A catalog was specified and the implementation does not support catalogs.

19579 A schema was specified and the implementation does not support schemas.

19580 The data source does not support the combination of the current settings of the
19581 SQL_ATTR_CONCURRENCY and SQL_ATTR_CURSOR_TYPE statement attributes.

19582 The SQL_ATTR_USE_BOOKMARKS statement attribute was set to SQL_UB_VARIABLE,
19583 and the SQL_ATTR_CURSOR_TYPE statement attribute was set to a cursor type for which
19584 the data source does not support bookmarks.

19585 HYT00 — Timeout expired
19586 The query timeout period expired before the data source returned the requested result set.
19587 The timeout period is set through *SQLSetStmtAttr*(), SQL_ATTR_QUERY_TIMEOUT.

19588 HYT01 — Connection timeout expired
19589 The connection timeout period expired before the data source responded to the request. The
19590 connection timeout period is set through *SQLSetConnectAttr*(),
19591 SQL_ATTR_CONNECTION_TIMEOUT.

19592 IM001 — Function not supported
19593 The function is not supported on the current connection to the data source.

19594 **COMMENTS**

19595 *SQLStatistics*() returns information about a single table as a result set, ordered by
19596 NON_UNIQUE, TYPE, INDEX_QUALIFIER, INDEX_NAME, and ORDINAL_POSITION. The
19597 result set combines statistics information (in the CARDINALITY and PAGES columns of the
19598 result set) for the table with information about each index.

19599 To determine the actual lengths of the TABLE_CAT, TABLE_SCHEM, TABLE_NAME, and
19600 COLUMN_NAME columns, an application can call *SQLGetInfo*() with the
19601 SQL_MAX_CATALOG_NAME_LEN, SQL_MAX_SCHEMA_NAME_LEN,
19602 SQL_MAX_TABLE_NAME_LEN, and SQL_MAX_COLUMN_NAME_LEN options.

19603 The following table lists the columns in the result set. Additional columns beyond column 13
19604 (FILTER_CONDITION) can be defined by the implementation. An application should gain
19605 access to implementation-defined columns by counting down from the end of the result set
19606 rather than by specifying an explicit ordinal position; see Section 7.3 on page 68.

Column Name	Col. No.	Data Type	Comments
TABLE_CAT	1	Varchar	Catalog identifier of the table to which the statistic or index applies; NULL if not applicable to the data source. If a data source supports catalogs, it returns an empty string for those tables that do not have catalogs.
TABLE_SCHEM	2	Varchar	Schema identifier of the table to which the statistic or index applies; NULL if not applicable to the data source. If a data source supports schemas, it returns an empty string for those tables that do not have schemas.
TABLE_NAME	3	Varchar not NULL	Table identifier of the table to which the statistic or index applies.
NON_UNIQUE	4	Smallint	SQL_TRUE if the index values can be nonunique. SQL_FALSE if the index values must be unique. NULL is returned if TYPE is SQL_TABLE_STAT.
INDEX_QUALIFIER	5	Varchar	The identifier that is used to qualify the index name doing a DROP INDEX; NULL is returned if an index qualifier is not supported by the data source or if TYPE is SQL_TABLE_STAT. If a non-null value is returned in this column, it must be used to qualify the index name on a DROP INDEX statement; otherwise the TABLE_SCHEM should be used to qualify the index name.
INDEX_NAME	6	Varchar	Index identifier; NULL is returned if TYPE is SQL_TABLE_STAT.
TYPE	7	Smallint not NULL	The type of information being returned: SQL_TABLE_STAT indicates a statistic for the table (in the CARDINALITY or PAGES column). SQL_INDEX_BTREE indicates a B-Tree index. SQL_INDEX_CLUSTERED indicates a clustered index. SQL_INDEX_CONTENT indicates a content index. SQL_INDEX_HASHED indicates a hashed index.

			SQL_INDEX_OTHER indicates another type of index.
ORDINAL_POSITION	8	Smallint	Column sequence number in index (starting with 1); NULL is returned if TYPE is SQL_TABLE_STAT.
COLUMN_NAME	9	Varchar	Column identifier. If the column is based on an expression, such as SALARY + BENEFITS, the expression is returned; if the expression cannot be determined, an empty string is returned. NULL is returned if TYPE is SQL_TABLE_STAT.
ASC_OR_DESC	10	Char(1)	Sort sequence for the column; ''A'' for ascending; ''D'' for descending; NULL is returned if the data source does not support a column sort sequence or if TYPE is SQL_TABLE_STAT.
CARDINALITY	11	Integer	Cardinality of table or index; number of rows in table if TYPE is SQL_TABLE_STAT; number of unique values in the index if TYPE is not SQL_TABLE_STAT; NULL is returned if the value is not available from the data source.
PAGES	12	Integer	Number of pages used to store the index or table; number of pages for the table if TYPE is SQL_TABLE_STAT; number of pages for the index if TYPE is not SQL_TABLE_STAT; NULL is returned if the value is not available from the data source, or if not applicable to the data source.
FILTER_CONDITION	13	Varchar	If the index is a filtered index, this is the filter condition, such as SALARY > 30000; if the filter condition cannot be determined, this is an empty string. NULL if the index is not a filtered index, it cannot be determined whether the index is a filtered index, or TYPE is SQL_TABLE_STAT.

If the row in the result set corresponds to a table, the implementation sets TYPE to SQL_TABLE_STAT and sets NON_UNIQUE, INDEX_QUALIFIER, INDEX_NAME, ORDINAL_POSITION, COLUMN_NAME, and ASC_OR_DESC to NULL. If CARDINALITY or PAGES are not available from the data source, the implementation sets them to NULL.

SEE ALSO

For information about	**See**
Overview of catalog functions	Chapter 7
Binding a buffer to a column in a result set	*SQLBindCol*()
Canceling statement processing	*SQLCancel*()

19688 Fetching a block of data or scrolling through a result set	*SQLFetchScroll*()
19689 Fetching a single row or a block of data in a forward-only 19690 direction.	*SQLFetch*()
19691 Returning the columns of foreign keys	*SQLForeignKeys*()
19692 Returning the columns of a primary key	*SQLPrimaryKeys*()

19693 **CHANGE HISTORY**

19694 **Version 2**

19695 Revised generally. See **Alignment with Popular Implementations** on page 2.

19696 **NAME**

19697 SQLTablePrivileges — Return as a result set a list of tables and the privileges associated with
19698 each table.

19699 **SYNOPSIS**

```
19700 SQLRETURN SQLTablePrivileges(
19701   SQLHSTMT StatementHandle,
19702   SQLCHAR * CatalogName,
19703   SQLSMALLINT NameLength1,
19704   SQLCHAR * SchemaName,
19705   SQLSMALLINT NameLength2,
19706   SQLCHAR * TableName,
19707   SQLSMALLINT NameLength3);
```

19708 **ARGUMENTS**

19709 *StatementHandle* [Input]
19710 Statement handle.

19711 *CatalogName* [Input]
19712 Table catalog. If a data source supports catalogs, an empty string denotes those tables that
19713 do not have catalogs.

19714 If the SQL_ATTR_METADATA_ID statement attribute is SQL_TRUE, this argument is
19715 interpreted as specified in **Identifier (ID) Arguments** on page 72. If it is SQL_FALSE, this
19716 argument is interpreted as specified in **Ordinary Arguments (OA)** on page 71.

19717 *NameLength1* [Input]
19718 Length of **CatalogName*.

19719 *SchemaName* [Input]
19720 String search pattern for schema names. If a data source supports schemas, an empty string
19721 denotes those tables that do not have schemas.

19722 If the SQL_ATTR_METADATA_ID statement attribute is SQL_TRUE, this argument is
19723 interpreted as specified in **Identifier (ID) Arguments** on page 72. If it is SQL_FALSE, this
19724 argument is interpreted as specified in **Pattern Value (PV) Arguments** on page 71 and the
19725 application may use a search pattern.

19726 *NameLength2* [Input]
19727 Length of **SchemaName*.

19728 *TableName* [Input]
19729 String search pattern for table names.

19730 If the SQL_ATTR_METADATA_ID statement attribute is SQL_TRUE, this argument is
19731 interpreted as specified in **Identifier (ID) Arguments** on page 72. If it is SQL_FALSE, this
19732 argument is interpreted as specified in **Pattern Value (PV) Arguments** on page 71 and the
19733 application may use a search pattern.

19734 *NameLength3* [Input]
19735 Length of **TableName*.

19736 **RETURN VALUE**

19737 SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or
19738 SQL_INVALID_HANDLE.

19739 **DIAGNOSTICS**

19740 When *SQLTablePrivileges*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated
19741 SQLSTATE value may be obtained by calling *SQLGetDiagRec*() with a *HandleType* of

19742 SQL_HANDLE_STMT and a *Handle* of *StatementHandle*. The following table lists the SQLSTATE
19743 values commonly returned by *SQLTablePrivileges*(). The return code associated with each
19744 SQLSTATEvalue is SQL_ERROR, except that for SQLSTATEvalues in class 01, the return code is
19745 SQL_SUCCESS_WITH_INFO.

19746 01000 — General warning
19747 Implementation-defined informational message.

19748 08S01 — Communication link failure
19749 The communication link to the data source failed before the function completed processing.

19750 24000 — Invalid cursor state
19751 A cursor was open on *StatementHandle*.

19752 HY000 — General error
19753 An error occurred for which there was no specific SQLSTATE and for which no
19754 implementation-specific SQLSTATE was defined. The error message returned by
19755 *SQLGetDiagRec*() in the **MessageText* buffer describes the error and its cause.

19756 HY001 — Memory allocation error
19757 The implementation failed to allocate memory required to support execution or completion
19758 of the function.

19759 HY008 — Operation canceled
19760 Asynchronous processing was enabled for *StatementHandle*. The function was called and
19761 before it completed execution, *SQLCancel*() was called on *StatementHandle*. The function
19762 was then called again on *StatementHandle*.

19763 The function was called and, before it completed execution, *SQLCancel*() was called on
19764 *StatementHandle* from a different thread in a multithread application.

19765 HY009 — Invalid use of null pointer
19766 The SQL_ATTR_METADATA_IDstatement attribute was set to SQL_TRUE, *CatalogName*
19767 was a null pointer, and the SQL_CATALOG_NAME option of *SQLGetInfo*() returns that
19768 catalog names are supported.

19769 The SQL_ATTR_METADATA_ID statement attribute was set to SQL_TRUE, and
19770 *SchemaName* or *TableName* argument was a null pointer.

19771 HY010 — Function sequence error
19772 An asynchronously executing function (not this one) was called for *StatementHandle* and
19773 was still executing when this function was called.

19774 *SQLBulkOperations*(), *SQLExecDirect*(), *SQLExecute*(), or *SQLSetPos*() was called for
19775 *StatementHandle* and returned SQL_NEED_DATA. This function was called before data was
19776 sent for all data-at-execution parameters or columns.

19777 HY090 — Invalid string or buffer length
19778 The value of one of the name length arguments was less than 0, but not equal to SQL_NTS.

19779 The value of one of the name length arguments exceeded the maximum length value for the
19780 corresponding qualifier or name.

19781 HYC00 — Optional feature not implemented
19782 A catalog was specified and the implementation does not support catalogs.

19783 A schema was specified and the implementation does not support schemas.

19784 A string search pattern was specified for the table schema, table name, or column name and
19785 the data source does not support search patterns for one or more of those arguments.

19786 The data source does not support the combination of the current settings of the
19787 SQL_ATTR_CONCURRENCY and SQL_ATTR_CURSOR_TYPE statement attributes.

19788 The SQL_ATTR_USE_BOOKMARKS statement attribute was set to SQL_UB_VARIABLE,
19789 and the SQL_ATTR_CURSOR_TYPE statement attribute was set to a cursor type for which
19790 the data source does not support bookmarks.

19791 HYT00 — Timeout expired
19792 The query timeout period expired before the data source returned the result set. The
19793 timeout period is set through *SQLSetStmtAttr*(), SQL_ATTR_QUERY_TIMEOUT.

19794 HYT01 — Connection timeout expired
19795 The connection timeout period expired before the data source responded to the request. The
19796 connection timeout period is set through *SQLSetConnectAttr*(),
19797 SQL_ATTR_CONNECTION_TIMEOUT.

19798 IM001 — Function not supported
19799 The function is not supported on the current connection to the data source.

19800 **COMMENTS**

19801 *SchemaName* and *TableName* accept search patterns, as defined in **Pattern Value (PV) Arguments**
19802 on page 71 and the application may use a search pattern.

19803 *SQLTablePrivileges*() returns the results as a standard result set, ordered by TABLE_CAT,
19804 TABLE_SCHEM, TABLE_NAME, and PRIVILEGE.

19805 To determine the actual lengths of the TABLE_CAT, TABLE_SCHEM, and TABLE_NAME
19806 columns, an application can call *SQLGetInfo*() with the SQL_MAX_CATALOG_NAME_LEN,
19807 SQL_MAX_SCHEMA_NAME_LEN, and SQL_MAX_TABLE_NAME_LEN options.

19808 The following table lists the columns in the result set. Additional columns beyond column 7
19809 (IS_GRANTABLE) can be defined by the implementation. An application should gain access to
19810 implementation-defined columns by counting down from the end of the result set rather than by
19811 specifying an explicit ordinal position; see Section 7.3 on page 68.

	Column Name	Col. No.	Data Type	Comments
19812–19813				
19814–19816	TABLE_CAT	1	Varchar	Catalog identifier; NULL if not applicable to the data source. If a data source supports catalogs, it returns an empty string for those tables that do not have catalogs.
19817–19819	TABLE_SCHEM	2	Varchar	Schema identifier; NULL if not applicable to the data source. If a data source supports schemas, it returns an empty string for those tables that do not have schemas.
19820–19821	TABLE_NAME	3	Varchar not NULL	Table identifier.

GRANTOR	4	Varchar	Identifier of the user who granted the privilege; NULL if not applicable to the data source. For all rows in which the value in the GRANTEE column is the owner of the object, the GRANTOR column is ''_SYSTEM''.
GRANTEE	5	Varchar not NULL	Identifier of the user to whom the privilege was granted.
PRIVILEGE	6	Varchar not NULL	Identifies the table privilege. May be one of the following or a data-source-specific privilege. SELECT: The grantee is permitted to retrieve data for one or more columns of the table. INSERT: The grantee is permitted to insert new rows containing data for one or more columns into to the table. UPDATE: The grantee is permitted to update the data in one or more columns of the table. DELETE: The grantee is permitted to delete rows of data from the table. REFERENCES: The grantee is permitted to refer to one or more columns of the table within a constraint (for example, a unique, referential, or table check constraint). The scope of action permitted the grantee by a given table privilege is data source-dependent. For example, the UPDATE privilege might permit the grantee to update all columns in a table on one data source and only those columns for which the grantor has the UPDATE privilege on another data source.
IS_GRANTABLE	7	Varchar	Indicates whether the grantee is permitted to grant the privilege to other users; ''YES'', ''NO'', or NULL if unknown or not applicable to the data source. A privilege is either grantable or not grantable, but not both. The result set returned by *SQLColumnPrivileges*() does not contain multiple rows for which all columns except the IS_GRANTABLE column contain the same value.

SEE ALSO

For information about	**See**
Overview of catalog functions	Chapter 7
Binding a buffer to a column in a result set	*SQLBindCol*()
Canceling statement processing	*SQLCancel*()
Returning privileges for a column or columns	*SQLColumnPrivileges*()

19862	Returning the columns in a table or tables	*SQLColumns*()
19863	Fetching a block of data or scrolling through a result set	*SQLFetchScroll*()
19864 19865	Fetching a single row or a block of data in a forward-only direction	*SQLFetch*()
19866	Returning table statistics and indexes	*SQLStatistics*()
19867	Returning a list of tables in a data source	*SQLTables*()

19868 **CHANGE HISTORY**

19869 **Version 2**

19870 Function added in this version.

19871 **NAME**

19872 SQLTables — Return as a result set the list of table, catalog, or schema names, and table types,
19873 stored in a specified data source.

19874 **SYNOPSIS**

```
19875 SQLRETURN SQLTables(
19876   SQLHSTMT StatementHandle,
19877   SQLCHAR * CatalogName,
19878   SQLSMALLINT NameLength1,
19879   SQLCHAR * SchemaName,
19880   SQLSMALLINT NameLength2,
19881   SQLCHAR * TableName,
19882   SQLSMALLINT NameLength3,
19883   SQLCHAR * TableType,
19884   SQLSMALLINT NameLength4);
```

19885 **ARGUMENTS**

19886 *StatementHandle* [Input]
19887 Statement handle for retrieved results.

19888 *CatalogName* [Input]
19889 Catalog name. If a data source supports catalogs, an empty string denotes those tables that
19890 do not have catalogs.

19891 If the SQL_ATTR_METADATA_ID statement attribute is SQL_TRUE, this argument is
19892 interpreted as specified in **Identifier (ID) Arguments** on page 72. If it is SQL_FALSE, this
19893 argument is interpreted as specified in **Pattern Value (PV) Arguments** on page 71 and the
19894 application may use a search pattern.

19895 *NameLength1* [Input]
19896 Length of **CatalogName*.

19897 *SchemaName* [Input]
19898 String search pattern for schema names. If a data source supports schemas, an empty string
19899 denotes those tables that do not have schemas.

19900 If the SQL_ATTR_METADATA_ID statement attribute is SQL_TRUE, this argument is
19901 interpreted as specified in **Identifier (ID) Arguments** on page 72. If it is SQL_FALSE, this
19902 argument is interpreted as specified in **Pattern Value (PV) Arguments** on page 71 and the
19903 application may use a search pattern.

19904 *NameLength2* [Input]
19905 Length of **SchemaName*.

19906 *TableName* [Input]
19907 String search pattern for table names.

19908 If the SQL_ATTR_METADATA_ID statement attribute is SQL_TRUE, this argument is
19909 interpreted as specified in **Identifier (ID) Arguments** on page 72. If it is SQL_FALSE, this
19910 argument is interpreted as specified in **Pattern Value (PV) Arguments** on page 71 and the
19911 application may use a search pattern.

19912 *NameLength3* [Input]
19913 Length of **TableName*.

19914 *TableType* [Input]
19915 List of table types to match.

19916 This argument is interpreted as specified in **Value List (VL) Arguments** on page 72,
19917 regardless of the SQL_ATTR_METADATA_ID statement attribute.

19918 *NameLength4* [Input]
19919 Length of **TableType*

19920 **RETURN VALUE**

19921 SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR or
19922 SQL_INVALID_HANDLE.

19923 **DIAGNOSTICS**

19924 When *SQLTables*() returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated
19925 SQLSTATE value can be obtained by calling *SQLGetDiagRec*() with a *HandleType* of
19926 SQL_HANDLE_STMT and a *Handle* of *StatementHandle*. The following table lists the SQLSTATE
19927 values commonly returned by *SQLTables*(). The return code associated with each SQLSTATE
19928 value is SQL_ERROR, except that for SQLSTATE values in class 01, the return code is
19929 SQL_SUCCESS_WITH_INFO.

19930 01000 — General warning
19931 Implementation-defined informational message.

19932 08S01 — Communication link failure
19933 The communication link to the data source failed before the function completed processing.

19934 24000 — Invalid cursor state
19935 A cursor was open on *StatementHandle*.

19936 HY000 — General error
19937 An error occurred for which there was no specific SQLSTATE and for which no
19938 implementation-specific SQLSTATE was defined. The error message returned by
19939 *SQLGetDiagRec*() in the **MessageText* buffer describes the error and its cause.

19940 HY001 — Memory allocation error
19941 The implementation failed to allocate memory required to support execution or completion
19942 of the function.

19943 HY008 — Operation canceled
19944 Asynchronous processing was enabled for *StatementHandle*. The function was called and
19945 before it completed execution, *SQLCancel*() was called on *StatementHandle*. The function
19946 was then called again on *StatementHandle*.

19947 The function was called and, before it completed execution, *SQLCancel*() was called on
19948 *StatementHandle* from a different thread in a multithread application.

19949 HY009 — Invalid use of null pointer
19950 The SQL_ATTR_METADATA_ID statement attribute was set to SQL_TRUE, *CatalogName*
19951 was a null pointer, and the SQL_CATALOG_NAME option of *SQLGetInfo*() returns that
19952 catalog names are supported.

19953 The SQL_ATTR_METADATA_ID statement attribute was set to SQL_TRUE, and
19954 *SchemaName* or *TableName* was a null pointer.

19955 HY010 — Function sequence error
19956 An asynchronously executing function (not this one) was called for *StatementHandle* and
19957 was still executing when this function was called.

19958 *SQLBulkOperations*(), *SQLExecDirect*(), *SQLExecute*(), or *SQLSetPos*() was called for
19959 *StatementHandle* and returned SQL_NEED_DATA. This function was called before data was
19960 sent for all data-at-execution parameters or columns.

19961 HY090 — Invalid string or buffer length
19962 The value of one of the length arguments was less than 0, but not equal to SQL_NTS.

19963 The value of one of the name length arguments exceeded the maximum length value for the
19964 corresponding name.

19965 HYC00 — Optional feature not implemented
19966 A catalog was specified and the implementation does not support catalogs.

19967 A schema was specified and the implementation does not support schemas.

19968 A string search pattern was specified for the table schema or table name and the data source
19969 does not support search patterns for one or more of those arguments.

19970 The data source does not support the combination of the current settings of the
19971 SQL_ATTR_CONCURRENCY and SQL_ATTR_CURSOR_TYPE statement attributes.

19972 The SQL_ATTR_USE_BOOKMARKS statement attribute was set to SQL_UB_VARIABLE,
19973 and the SQL_ATTR_CURSOR_TYPE statement attribute was set to a cursor type for which
19974 the data source does not support bookmarks.

19975 HYT00 — Timeout expired
19976 The query timeout period expired before the data source returned the requested result set.
19977 The timeout period is set through *SQLSetStmtAttr*(), SQL_ATTR_QUERY_TIMEOUT.

19978 HYT01 — Connection timeout expired
19979 The connection timeout period expired before the data source responded to the request. The
19980 connection timeout period is set through *SQLSetConnectAttr*(),
19981 SQL_ATTR_CONNECTION_TIMEOUT.

19982 IM001 — Function not supported
19983 The function is not supported on the current connection to the data source.

19984 **COMMENTS**

19985 *SQLTables*() lists all tables in the requested range. A user may or may not have SELECT
19986 privileges to any of these tables. To check accessibility, an application can:

19987 • Call *SQLGetInfo*() and check the SQL_ACCESSIBLE_TABLES info value.

19988 • Call *SQLTablePrivileges*() to check the privileges for each table.

19989 Otherwise, the application must be able to handle a situation where the user selects a table for
19990 which SELECT privileges are not granted.

19991 *CatalogName*, *SchemaName*, and *TableName* accept search patterns.

19992 To support enumeration of catalogs, schemas, and table types, *SQLTables*() defines the following
19993 special semantics for the *CatalogName*, *SchemaName*, *TableName*, and *TableType* arguments:

19994 • If *CatalogName* is SQL_ALL_CATALOGS and *SchemaName* and *TableName* are empty strings,
19995 then the result set contains a list of valid catalogs for the data source. (All columns except the
19996 TABLE_CAT column contain NULLs.)

19997 • If *SchemaName* is SQL_ALL_SCHEMAS and *CatalogName* and *TableName* are empty strings,
19998 then the result set contains a list of valid schemas for the data source. (All columns except the
19999 TABLE_SCHEM column contain NULLs.)

20000 • If *TableType* is SQL_ALL_TABLE_TYPES and *CatalogName*, *SchemaName*, and *TableName* are
20001 empty strings, then the result set contains a list of valid table types for the data source. (All
20002 columns except the TABLE_TYPE column contain NULLs.)

20003 If *TableType* is not an empty string, it must contain a list of comma-separated values for the types
20004 of interest; each value may be enclosed in single quotation marks or unquoted. For example,

"TABLE", "VIEW" or "TABLE, VIEW". An application should always specify the table type in upper case; the implementation should convert the table type to whatever case the data source needs. If the data source does not support a specified table type, *SQLTables*() does not return any results for that type.

SQLTables() returns the results as a standard result set, ordered by TABLE_TYPE, TABLE_CAT, TABLE_SCHEM, and TABLE_NAME.

To determine the actual lengths of the TABLE_CAT, TABLE_SCHEM, and TABLE_NAME columns, an application can call *SQLGetInfo*() with the SQL_MAX_CATALOG_NAME_LEN, SQL_MAX_SCHEMA_NAME_LEN, and SQL_MAX_TABLE_NAME_LEN options.

The following table lists the columns in the result set. Additional columns beyond column 5 (REMARKS) can be defined by the implementation. An application should gain access to implementation-defined columns by counting down from the end of the result set rather than by specifying an explicit ordinal position; see Section 7.3 on page 68.

Column name	Col. No.	Data type	Comments
TABLE_CAT	1	Varchar	Catalog identifier; NULL if not applicable to the data source. If a data source supports catalogs, it returns an empty string for those tables that do not have catalogs.
TABLE_SCHEM	2	Varchar	Schema identifier; NULL if not applicable to the data source. If a data source supports schemas, it returns an empty string for those tables that do not have schemas.
TABLE_NAME	3	Varchar	Table identifier.
TABLE_TYPE	4	Varchar	Table type identifier; one of the following: ''TABLE'', ''VIEW'', ''SYSTEM TABLE'', ''GLOBAL TEMPORARY'', ''LOCAL TEMPORARY'', ''ALIAS'', ''SYNONYM'' or a data source-specific type identifier.
REMARKS	5	Varchar	A description of the table.

SEE ALSO

For information about	See
Overview of catalog functions	Chapter 7
Binding a buffer to a column in a result set	*SQLBindCol*()
Canceling statement processing	*SQLCancel*()
Returning privileges for a column or columns	*SQLColumnPrivileges*()
Returning the columns in a table or tables	*SQLColumns*()
Fetching a block of data or scrolling through a result set	*SQLFetchScroll*()
Fetching a single row or a block of data in a forward-only direction	*SQLFetch*()
Returning table statistics and indexes	*SQLStatistics*()

20046 Returning privileges for a table or tables *SQLTablePrivileges*()

20047 **CHANGE HISTORY**

20048 **Version 2**

20049 Function added in this version.

Appendix A

Diagnostic Reference Information

This appendix contains reference information on all SQLSTATE codes returned by XDBC implementations.

A.1 Class and Subclass Origin

The SQL_DIAG_CLASS_ORIGIN and SQL_DIAG_SUBCLASS_ORIGIN fields of the diagnostic area indicate, for any returned diagnostic, the document that defines its SQLSTATE class and subclass, respectively. (For more details, see *SQLGetDiagField*().

Class

The SQL_DIAG_CLASS_ORIGIN field is set to

'XDBC' if the SQLSTATE class is 'IM'.

'ISO 9075' for all other SQLSTATE classes of diagnostics defined in this specification, because they also appear in the ISO CLI International Standard.

For implementation-defined SQLSTATEs, SQL_DIAG_CLASS_ORIGIN is an implementation-defined string that uniquely identifies the vendor and product whose documentation defines the class.

Subclass

The SQL_DIAG_SUBCLASS_ORIGIN field is set to

'XDBC' for the following SQLSTATEs defined in this specification:

01S00	08S01	42S01	HY095	HY105	HYT01	IM006
01S01	21S01	42S02	HY097	HY107	IM001	IM007
01S02	21S02	42S11	HY098	HY109	IM002	IM008
01S06	25S01	42S12	HY099	HY110	IM003	IM010
01S07	25S02	42S21	HY100	HY111	IM004	IM011
07S01	25S03	42S22	HY101	HYT00	IM005	IM012

'ISO 9075' for all other diagnostics defined in this specification, because they also appear in the ISO CLI International Standard.

For implementation-defined SQLSTATEs, including implementation-defined subclasses of SQLSTATEs defined in this specification, SQL_DIAG_SUBCLASS_ORIGIN is an implementation-defined string that uniquely identifies the vendor and product whose documentation defines the subclass.

20082 ## A.2 SQLSTATE Cross-reference (Non-normative)

20083 The following tables show each case where an SQLSTATE value is defined in the
20084 **DIAGNOSTICS** section of a function description in Chapter 21. It is meant to be complete, but
20085 any inconsistencies between these tables and the function descriptions are to be resolved in favor
20086 of the function descriptions.

20087 Additional diagnostics can result from a call to *SQLExecDirect*(), *SQLExecute*(), or *SQLPrepare*()
20088 based on the SQL statement text. These diagnostics are listed on the respective reference manual
20089 page and defined in the X/Open **SQL** specification.

20090 01000 — General warning
20091 *SQLAllocHandle*() *SQLBindCol*() *SQLBindParameter*() *SQLBrowseConnect*()
20092 *SQLBulkOperations*() *SQLCancel*() *SQLCloseCursor*() *SQLColAttribute*()
20093 *SQLColumnPrivileges*() *SQLColumns*() *SQLConnect*() *SQLCopyDesc*() *SQLDataSources*()
20094 *SQLDescribeCol*() *SQLDescribeParam*() *SQLDisconnect*() *SQLDriverConnect*() *SQLDrivers*()
20095 *SQLEndTran*() *SQLExecDirect*() *SQLExecute*() *SQLFetch*() *SQLFetchScroll*() *SQLForeignKeys*()
20096 *SQLFreeStmt*() *SQLGetConnectAttr*() *SQLGetCursorName*() *SQLGetData*() *SQLGetDescField*()
20097 *SQLGetDescRec*() *SQLGetEnvAttr*() *SQLGetFunctions*() *SQLGetInfo*() *SQLGetStmtAttr*()
20098 *SQLGetTypeInfo*() *SQLMoreResults*() *SQLNativeSql*() *SQLNumParams*() *SQLNumResultCols*()
20099 *SQLParamData*() *SQLPrepare*() *SQLPrimaryKeys*() *SQLProcedures*() *SQLPutData*()
20100 *SQLRowCount*() *SQLSetConnectAttr*() *SQLSetCursorName*() *SQLSetDescField*()
20101 *SQLSetDescRec*() *SQLSetEnvAttr*() *SQLSetPos*() *SQLSetStmtAttr*() *SQLSpecialColumns*()
20102 *SQLStatistics*() *SQLTablePrivileges*() *SQLTables*()

20103 01001 — Cursor operation conflict
20104 *SQLExecDirect*() *SQLExecute*() *SQLSetPos*()

20105 01002 — Disconnect error
20106 *SQLDisconnect*()

20107 01004 — String data, right truncation
20108 *SQLBrowseConnect*() *SQLBulkOperations*() *SQLColAttribute*() *SQLDataSources*()
20109 *SQLDescribeCol*() *SQLDriverConnect*() *SQLDrivers*() *SQLFetch*() *SQLFetchScroll*()
20110 *SQLGetConnectAttr*() *SQLGetCursorName*() *SQLGetData*() *SQLGetDescField*()
20111 *SQLGetDescRec*() *SQLGetEnvAttr*() *SQLGetInfo*() *SQLGetStmtAttr*() *SQLNativeSql*()
20112 *SQLPutData*() *SQLSetCursorName*() *SQLSetPos*()

20113 01S00 — Invalid connection string attribute
20114 *SQLBrowseConnect*() *SQLDriverConnect*()

20115 01S01 — Error in row
20116 *SQLBulkOperations*() *SQLFetch*() *SQLSetPos*()

20117 01S02 — Attribute value changed
20118 *SQLBrowseConnect*() *SQLConnect*() *SQLDriverConnect*() *SQLExecDirect*() *SQLExecute*()
20119 *SQLGetTypeInfo*() *SQLMoreResults*() *SQLPrepare*() *SQLSetConnectAttr*() *SQLSetDescField*()
20120 *SQLSetEnvAttr*() *SQLSetStmtAttr*()

20121 01S06 — Attempt to fetch before the result set returned the first row-set
20122 *SQLFetchScroll*()

20123 01S07 — Fractional truncation
20124 *SQLBulkOperations*() *SQLExecDirect*() *SQLExecute*() *SQLFetch*() *SQLFetchScroll*()
20125 *SQLGetData*() *SQLSetPos*()

20126 07001 — Wrong number of parameters
20127 *SQLExecDirect*() *SQLExecute*()

20128 07002 — COUNT field incorrect
20129 *SQLExecDirect*() *SQLExecute*()

20130 07005 — Prepared statement not a cursor-specification
20131 *SQLColAttribute*() *SQLDescribeCol*()

20132 07006 — Restricted data type attribute violation
20133 *SQLBindCol*() *SQLBindParameter*() *SQLBulkOperations*() *SQLExecDirect*() *SQLExecute*()
20134 *SQLFetch*() *SQLFetchScroll*() *SQLGetData*() *SQLParamData*() *SQLPutData*()
20135 *SQLSetDescField*() *SQLSetDescRec*() *SQLSetPos*()

20136 07009 — Invalid descriptor index
20137 *SQLBindCol*() *SQLBindParameter*() *SQLBulkOperations*() *SQLColAttribute*() *SQLDescribeCol*()
20138 *SQLDescribeParam*() *SQLGetData*() *SQLGetDescField*() *SQLGetDescRec*() *SQLSetDescField*()
20139 *SQLSetDescRec*()

20140 07S01 — Invalid use of default parameter
20141 *SQLExecDirect*() *SQLExecute*() *SQLPutData*()

20142 08001 — Client unable to establish connection
20143 *SQLBrowseConnect*() *SQLConnect*() *SQLDriverConnect*()

20144 08002 — Connection name in use
20145 *SQLBrowseConnect*() *SQLConnect*() *SQLDriverConnect*()

20146 08003 — Connection does not exist
20147 *SQLAllocHandle*() *SQLDisconnect*() *SQLGetConnectAttr*() *SQLGetInfo*() *SQLNativeSql*()
20148 *SQLSetConnectAttr*()

20149 08003 — Connection not open
20150 *SQLEndTran*()

20151 08004 — Data source rejected the connection
20152 *SQLBrowseConnect*() *SQLConnect*() *SQLDriverConnect*()

20153 08007 — Connection failure during transaction
20154 *SQLEndTran*() *SQLSetConnectAttr*()

20155 08S01 — Communication link failure
20156 *SQLBrowseConnect*() *SQLColumnPrivileges*() *SQLColumns*() *SQLConnect*() *SQLCopyDesc*()
20157 *SQLDescribeCol*() *SQLDescribeParam*() *SQLDriverConnect*() *SQLExecDirect*() *SQLExecute*()
20158 *SQLFetch*() *SQLFetchScroll*() *SQLForeignKeys*() *SQLGetConnectAttr*() *SQLGetData*()
20159 *SQLGetDescField*() *SQLGetDescRec*() *SQLGetFunctions*() *SQLGetInfo*() *SQLGetTypeInfo*()
20160 *SQLMoreResults*() *SQLNativeSql*() *SQLNumParams*() *SQLNumResultCols*() *SQLParamData*()
20161 *SQLPrepare*() *SQLPrimaryKeys*() *SQLProcedures*() *SQLPutData*() *SQLSetConnectAttr*()
20162 *SQLSetDescField*() *SQLSetDescRec*() *SQLSetStmtAttr*() *SQLSpecialColumns*() *SQLStatistics*()
20163 *SQLTablePrivileges*() *SQLTables*()

20164 21S01 — Insert value list does not match column list
20165 *SQLDescribeParam*()

20166 21S02 — Degree of derived table does not match column list
20167 *SQLBulkOperations*() *SQLSetPos*()

20168 22001 — String data, right truncation
20169 *SQLBulkOperations*() *SQLExecDirect*() *SQLExecute*() *SQLFetch*() *SQLFetchScroll*()
20170 *SQLPutData*() *SQLSetPos*()

20171 22002 — Indicator variable required but not supplied
20172 *SQLExecDirect*() *SQLExecute*() *SQLFetch*() *SQLFetchScroll*() *SQLGetData*()

20173 22003 — Numeric value out of range
20174 *SQLBulkOperations*() *SQLFetch*() *SQLFetchScroll*() *SQLGetData*() *SQLPutData*() *SQLSetPos*()

20175 22007 — Invalid date/time format
20176 *SQLBulkOperations*() *SQLFetch*() *SQLFetchScroll*() *SQLGetData*() *SQLNativeSql*()
20177 *SQLPutData*() *SQLSetPos*()

20178 22008 — Date/time field overflow
20179 *SQLBulkOperations*() *SQLPutData*() *SQLSetPos*()

20180 22012 — Division by zero
20181 *SQLFetch*() *SQLFetchScroll*() *SQLGetData*() *SQLPutData*()

20182 22015 — Interval field overflow
20183 *SQLBulkOperations*() *SQLFetch*() *SQLFetchScroll*() *SQLGetData*() *SQLPutData*() *SQLSetPos*()

20184 22018 — Invalid character value for cast specification
20185 *SQLBulkOperations*() *SQLFetch*() *SQLFetchScroll*() *SQLPutData*() *SQLSetPos*()

20186 22018 — Invalid character value
20187 *SQLGetData*() *SQLPrepare*()

20188 22019 — Invalid escape character
20189 *SQLPrepare*()

20190 22025 — Invalid escape sequence
20191 *SQLExecDirect*() *SQLExecute*() *SQLPrepare*()

20192 22026 — String data, length mismatch
20193 *SQLParamData*()

20194 23000 — Integrity constraint violation
20195 *SQLBulkOperations*() *SQLSetPos*()

20196 24000 — Invalid cursor state
20197 *SQLBulkOperations*() *SQLCloseCursor*() *SQLColumnPrivileges*() *SQLColumns*() *SQLFetch*()
20198 *SQLFetchScroll*() *SQLForeignKeys*() *SQLGetData*() *SQLGetDescField*() *SQLGetDescRec*()
20199 *SQLGetStmtAttr*() *SQLGetTypeInfo*() *SQLNativeSql*() *SQLPrepare*() *SQLPrimaryKeys*()
20200 *SQLProcedures*() *SQLSetConnectAttr*() *SQLSetCursorName*() *SQLSetPos*() *SQLSetStmtAttr*()
20201 *SQLSpecialColumns*() *SQLStatistics*() *SQLTablePrivileges*() *SQLTables*()

20202 25000 — Invalid transaction state
20203 *SQLDisconnect*()

20204 25S01 — Transaction state unknown
20205 *SQLEndTran*()

20206 25S02 — Transaction is still active
20207 *SQLEndTran*()

20208 25S03 — Transaction is rolled back
20209 *SQLEndTran*()

20210 28000 — Invalid authorization specification
20211 *SQLBrowseConnect*() *SQLConnect*() *SQLDriverConnect*()

20212 34000 — Invalid cursor name
20213 *SQLExecDirect*() *SQLPrepare*() *SQLSetCursorName*()

20214 40001 — Serialization failure
20215 *SQLFetch*() *SQLFetchScroll*() *SQLGetTypeInfo*() *SQLNativeSql*() *SQLNumParams*()
20216 *SQLNumResultCols*() *SQLParamData*()

20217 42000 — Syntax error or access violation
20218 *SQLBulkOperations*() *SQLExecute*() *SQLSetPos*()

20219 HY000 — General error
20220 *SQLAllocHandle*() *SQLBindCol*() *SQLBindParameter*() *SQLBrowseConnect*()
20221 *SQLBulkOperations*() *SQLCancel*() *SQLCloseCursor*() *SQLColAttribute*()
20222 *SQLColumnPrivileges*() *SQLColumns*() *SQLConnect*() *SQLCopyDesc*() *SQLDataSources*()
20223 *SQLDescribeCol*() *SQLDescribeParam*() *SQLDisconnect*() *SQLDriverConnect*() *SQLDrivers*()
20224 *SQLEndTran*() *SQLExecDirect*() *SQLExecute*() *SQLFetch*() *SQLFetchScroll*() *SQLForeignKeys*()
20225 *SQLFreeHandle*() *SQLFreeStmt*() *SQLGetConnectAttr*() *SQLGetCursorName*() *SQLGetData*()
20226 *SQLGetDescField*() *SQLGetDescRec*() *SQLGetEnvAttr*() *SQLGetFunctions*() *SQLGetInfo*()
20227 *SQLGetStmtAttr*() *SQLGetTypeInfo*() *SQLMoreResults*() *SQLNativeSql*() *SQLNumParams*()
20228 *SQLNumResultCols*() *SQLParamData*() *SQLPrepare*() *SQLPrimaryKeys*() *SQLProcedures*()
20229 *SQLPutData*() *SQLRowCount*() *SQLSetConnectAttr*() *SQLSetCursorName*() *SQLSetDescField*()
20230 *SQLSetDescRec*() *SQLSetEnvAttr*() *SQLSetPos*() *SQLSetStmtAttr*() *SQLSpecialColumns*()
20231 *SQLStatistics*() *SQLTablePrivileges*() *SQLTables*()

20232 HY001 — Memory allocation error
20233 *SQLAllocHandle*() *SQLBindCol*() *SQLBindParameter*() *SQLBrowseConnect*()
20234 *SQLBulkOperations*() *SQLCancel*() *SQLCloseCursor*() *SQLColAttribute*()
20235 *SQLColumnPrivileges*() *SQLColumns*() *SQLConnect*() *SQLCopyDesc*() *SQLDataSources*()
20236 *SQLDescribeCol*() *SQLDescribeParam*() *SQLDisconnect*() *SQLDriverConnect*() *SQLDrivers*()
20237 *SQLEndTran*() *SQLExecDirect*() *SQLExecute*() *SQLFetch*() *SQLFetchScroll*() *SQLForeignKeys*()
20238 *SQLFreeHandle*() *SQLFreeStmt*() *SQLGetConnectAttr*() *SQLGetCursorName*() *SQLGetData*()
20239 *SQLGetDescField*() *SQLGetDescRec*() *SQLGetEnvAttr*() *SQLGetFunctions*() *SQLGetInfo*()
20240 *SQLGetStmtAttr*() *SQLGetTypeInfo*() *SQLMoreResults*() *SQLNativeSql*() *SQLNumParams*()
20241 *SQLNumResultCols*() *SQLParamData*() *SQLPrepare*() *SQLPrimaryKeys*() *SQLProcedures*()
20242 *SQLPutData*() *SQLRowCount*() *SQLSetConnectAttr*() *SQLSetCursorName*() *SQLSetDescField*()
20243 *SQLSetDescRec*() *SQLSetEnvAttr*() *SQLSetPos*() *SQLSetStmtAttr*() *SQLSpecialColumns*()
20244 *SQLStatistics*() *SQLTablePrivileges*() *SQLTables*()

20245 HY003 — Invalid application buffer type
20246 *SQLBindCol*() *SQLBindParameter*() *SQLGetData*()

20247 HY004 — Invalid SQL data type
20248 *SQLBindParameter*() *SQLGetTypeInfo*()

20249 HY007 — Associated statement is not prepared
20250 *SQLCopyDesc*() *SQLDescribeCol*() *SQLGetDescField*() *SQLGetDescRec*()

20251 HY008 — Operation canceled
20252 *SQLBulkOperations*() *SQLColAttribute*() *SQLColumnPrivileges*() *SQLColumns*()
20253 *SQLDescribeCol*() *SQLDescribeParam*() *SQLExecDirect*() *SQLExecute*() *SQLFetch*()
20254 *SQLFetchScroll*() *SQLForeignKeys*() *SQLGetData*() *SQLGetTypeInfo*() *SQLMoreResults*()
20255 *SQLNumParams*() *SQLNumResultCols*() *SQLParamData*() *SQLPrepare*() *SQLPrimaryKeys*()
20256 *SQLProcedures*() *SQLPutData*() *SQLSetPos*() *SQLSpecialColumns*() *SQLStatistics*()
20257 *SQLTablePrivileges*() *SQLTables*()

20258 HY009 — Invalid use of null pointer
20259 *SQLAllocHandle*() *SQLBindParameter*() *SQLBulkOperations*() *SQLColumnPrivileges*()
20260 *SQLColumns*() *SQLExecDirect*() *SQLForeignKeys*() *SQLNativeSql*() *SQLPrepare*()
20261 *SQLPrimaryKeys*() *SQLProcedures*() *SQLPutData*() *SQLSetConnectAttr*() *SQLSetCursorName*()
20262 *SQLSetDescField*() *SQLSetEnvAttr*() *SQLSetStmtAttr*() *SQLSpecialColumns*() *SQLStatistics*()
20263 *SQLTablePrivileges*() *SQLTables*()

20264 HY010 — Function sequence error
20265 *SQLBindCol*() *SQLBindParameter*() *SQLBulkOperations*() *SQLCloseCursor*() *SQLColAttribute*()

20266 *SQLColumnPrivileges*() *SQLColumns*() *SQLCopyDesc*() *SQLDescribeCol*() *SQLDescribeParam*()
20267 *SQLDisconnect*() *SQLEndTran*() *SQLExecDirect*() *SQLExecute*() *SQLFetch*() *SQLFetchScroll*()
20268 *SQLForeignKeys*() *SQLFreeHandle*() *SQLFreeStmt*() *SQLGetConnectAttr*()
20269 *SQLGetCursorName*() *SQLGetData*() *SQLGetDescField*() *SQLGetDescRec*() *SQLGetFunctions*()
20270 *SQLGetStmtAttr*() *SQLGetTypeInfo*() *SQLMoreResults*() *SQLNumParams*()
20271 *SQLNumResultCols*() *SQLParamData*() *SQLPrepare*() *SQLPrimaryKeys*() *SQLProcedures*()
20272 *SQLPutData*() *SQLRowCount*() *SQLSetConnectAttr*() *SQLSetCursorName*() *SQLSetDescField*()
20273 *SQLSetDescRec*() *SQLSetPos*() *SQLSetStmtAttr*() *SQLSpecialColumns*() *SQLStatistics*()
20274 *SQLTablePrivileges*() *SQLTables*()

20275 HY011 — Attribute cannot be set now
20276 *SQLPutData*() *SQLSetConnectAttr*() *SQLSetEnvAttr*() *SQLSetStmtAttr*()

20277 HY012 — Invalid transaction operation code
20278 *SQLEndTran*()

20279 HY013 — Memory management error
20280 *SQLAllocHandle*() *SQLFreeHandle*()

20281 HY014 — Limit on the number of handles exceeded
20282 *SQLAllocHandle*()

20283 HY016 — Cannot modify an implementation row descriptor
20284 *SQLCopyDesc*() *SQLSetDescField*() *SQLSetDescRec*()

20285 HY017 — Invalid use of an automatically allocated descriptor handle.
20286 *SQLFreeHandle*() *SQLSetStmtAttr*()

20287 HY018 — Server declined cancel request
20288 *SQLCancel*()

20289 HY019 — Non-character and non-binary data sent in pieces
20290 *SQLPutData*()

20291 HY020 — Attempt to concatenate a null value
20292 *SQLPutData*()

20293 HY021 — Inconsistent descriptor information
20294 *SQLBindCol*() *SQLBindParameter*() *SQLCopyDesc*() *SQLSetDescField*() *SQLSetDescRec*()

20295 HY024 — Invalid attribute value
20296 *SQLSetConnectAttr*() *SQLSetEnvAttr*() *SQLSetStmtAttr*()

20297 HY090 — Invalid string or buffer length
20298 *SQLBindCol*() *SQLBindParameter*() *SQLBrowseConnect*() *SQLBulkOperations*()
20299 *SQLColAttribute*() *SQLColumnPrivileges*() *SQLColumns*() *SQLConnect*() *SQLDataSources*()
20300 *SQLDescribeCol*() *SQLDriverConnect*() *SQLDrivers*() *SQLExecDirect*() *SQLExecute*()
20301 *SQLForeignKeys*() *SQLGetCursorName*() *SQLGetData*() *SQLGetInfo*() *SQLNativeSql*()
20302 *SQLPrepare*() *SQLPrimaryKeys*() *SQLProcedures*() *SQLPutData*() *SQLSetConnectAttr*()
20303 *SQLSetCursorName*() *SQLSetEnvAttr*() *SQLSetPos*() *SQLSetStmtAttr*() *SQLSpecialColumns*()
20304 *SQLStatistics*() *SQLTablePrivileges*() *SQLTables*()

20305 HY091 — Invalid descriptor field identifier
20306 *SQLColAttribute*() *SQLGetDescField*() *SQLGetDescRec*() *SQLSetDescField*() *SQLSetDescRec*()

20307 HY092 — Invalid attribute identifier
20308 *SQLAllocHandle*() *SQLBulkOperations*() *SQLCopyDesc*() *SQLDriverConnect*() *SQLEndTran*()
20309 *SQLFreeStmt*() *SQLGetConnectAttr*() *SQLGetEnvAttr*() *SQLGetStmtAttr*()
20310 *SQLSetConnectAttr*() *SQLSetEnvAttr*() *SQLSetPos*() *SQLSetStmtAttr*()

20311 HY095 — Function type out of range
20312 *SQLGetFunctions*()

20313 HY096 — Information type out of range
20314 *SQLGetInfo*()

20315 HY097 — Column type out of range
20316 *SQLSpecialColumns*()

20317 HY098 — Scope type out of range
20318 *SQLSpecialColumns*()

20319 HY099 — Nullable type out of range
20320 *SQLSpecialColumns*()

20321 HY100 — Uniqueness option type out of range
20322 *SQLStatistics*()

20323 HY101 — Accuracy option type out of range
20324 *SQLStatistics*()

20325 HY103 — Invalid retrieval code
20326 *SQLDataSources*() *SQLDrivers*()

20327 HY104 — Invalid precision value
20328 *SQLBindParameter*() *SQLSetDescRec*()

20329 HY105 — Invalid parameter type
20330 *SQLBindParameter*() *SQLExecDirect*() *SQLExecute*() *SQLSetDescField*()

20331 HY106 — Fetch type out of range
20332 *SQLFetchScroll*()

20333 HY107 — Row value out of range
20334 *SQLFetch*() *SQLFetchScroll*() *SQLSetPos*()

20335 HY109 — Invalid cursor position
20336 *SQLExecDirect*() *SQLExecute*() *SQLGetData*() *SQLGetStmtAttr*() *SQLNativeSql*() *SQLSetPos*()

20337 HY110 — Invalid value of DriverCompletion
20338 *SQLDriverConnect*()

20339 HY111 — Invalid bookmark value
20340 *SQLFetchScroll*()

20341 HYC00 — Optional feature not implemented
20342 *SQLBindCol*() *SQLBindParameter*() *SQLBulkOperations*() *SQLColAttribute*()
20343 *SQLColumnPrivileges*() *SQLColumns*() *SQLDriverConnect*() *SQLEndTran*() *SQLExecDirect*()
20344 *SQLExecute*() *SQLFetch*() *SQLFetchScroll*() *SQLForeignKeys*() *SQLGetConnectAttr*()
20345 *SQLGetData*() *SQLGetEnvAttr*() *SQLGetInfo*() *SQLGetStmtAttr*() *SQLGetTypeInfo*()
20346 *SQLPrepare*() *SQLPrimaryKeys*() *SQLProcedures*() *SQLSetConnectAttr*() *SQLSetEnvAttr*()
20347 *SQLSetPos*() *SQLSetStmtAttr*() *SQLSpecialColumns*() *SQLStatistics*() *SQLTablePrivileges*()
20348 *SQLTables*()

20349 HYT00 — Timeout expired
20350 *SQLBrowseConnect*() *SQLBulkOperations*() *SQLColumnPrivileges*() *SQLColumns*()
20351 *SQLConnect*() *SQLDriverConnect*() *SQLExecDirect*() *SQLExecute*() *SQLForeignKeys*()
20352 *SQLGetTypeInfo*() *SQLPrepare*() *SQLPrimaryKeys*() *SQLProcedures*() *SQLSetPos*()
20353 *SQLSpecialColumns*() *SQLStatistics*() *SQLTablePrivileges*() *SQLTables*()

20354 HYT01 — Connection timeout expired
20355 *SQLAllocHandle*() *SQLBindCol*() *SQLBindParameter*() *SQLBrowseConnect*()

20356 *SQLBulkOperations*() *SQLCancel*() *SQLCloseCursor*() *SQLColAttribute*()
20357 *SQLColumnPrivileges*() *SQLColumns*() *SQLConnect*() *SQLCopyDesc*() *SQLDescribeCol*()
20358 *SQLDescribeParam*() *SQLDisconnect*() *SQLDriverConnect*() *SQLEndTran*() *SQLExecDirect*()
20359 *SQLExecute*() *SQLFetch*() *SQLFetchScroll*() *SQLForeignKeys*() *SQLFreeHandle*() *SQLFreeStmt*()
20360 *SQLGetConnectAttr*() *SQLGetCursorName*() *SQLGetData*() *SQLGetDescField*()
20361 *SQLGetDescRec*() *SQLGetEnvAttr*() *SQLGetFunctions*() *SQLGetInfo*() *SQLGetStmtAttr*()
20362 *SQLGetTypeInfo*() *SQLMoreResults*() *SQLNativeSql*() *SQLNumParams*() *SQLNumResultCols*()
20363 *SQLParamData*() *SQLPrepare*() *SQLPrimaryKeys*() *SQLProcedures*() *SQLPutData*()
20364 *SQLRowCount*() *SQLSetConnectAttr*() *SQLSetCursorName*() *SQLSetDescField*()
20365 *SQLSetDescRec*() *SQLSetEnvAttr*() *SQLSetPos*() *SQLSetStmtAttr*() *SQLSpecialColumns*()
20366 *SQLStatistics*() *SQLTablePrivileges*() *SQLTables*()

20367 IM001 — Function not supported
20368 *SQLAllocHandle*() *SQLBindCol*() *SQLBindParameter*() *SQLBrowseConnect*()
20369 *SQLBulkOperations*() *SQLCancel*() *SQLCloseCursor*() *SQLColAttribute*()
20370 *SQLColumnPrivileges*() *SQLColumns*() *SQLConnect*() *SQLCopyDesc*() *SQLDescribeCol*()
20371 *SQLDescribeParam*() *SQLDisconnect*() *SQLDriverConnect*() *SQLEndTran*() *SQLExecDirect*()
20372 *SQLExecute*() *SQLFetch*() *SQLFetchScroll*() *SQLForeignKeys*() *SQLFreeHandle*() *SQLFreeStmt*()
20373 *SQLGetConnectAttr*() *SQLGetCursorName*() *SQLGetData*() *SQLGetDescField*()
20374 *SQLGetDescRec*() *SQLGetEnvAttr*() *SQLGetInfo*() *SQLGetStmtAttr*() *SQLGetTypeInfo*()
20375 *SQLMoreResults*() *SQLNativeSql*() *SQLNumParams*() *SQLNumResultCols*() *SQLParamData*()
20376 *SQLPrepare*() *SQLPrimaryKeys*() *SQLProcedures*() *SQLPutData*() *SQLRowCount*()
20377 *SQLSetConnectAttr*() *SQLSetCursorName*() *SQLSetDescField*() *SQLSetDescRec*() *SQLSetPos*()
20378 *SQLSetStmtAttr*() *SQLSpecialColumns*() *SQLStatistics*() *SQLTablePrivileges*() *SQLTables*()

20379 IM002 — Data source not found and no default driver specified
20380 *SQLBrowseConnect*() *SQLConnect*() *SQLDriverConnect*()

Appendix B

State Tables

Notes to Reviewers

This section with side shading will not appear in the final copy. - Ed.

For this draft, the CLI Draft 28 state tables are included, with cosmetic changes to adapt it to XDBC. New XDBC functions such as *SQLBrowseConnect*(), *SQLBulkOperations*(), and *SQLSetPos*() are not yet included.

This appendix shows the effect of each XDBC function on the states of the various XDBC handles.

- Section B.1 on page 548 describes valid state transitions for environment handles.
- Table B-1 on page 549 describes valid state transitions for connection handles.
- Table B-2 on page 551 describes legal state transitions for statement handles. The data-at-execute dialogue is an annex of this state table, contained in Section B.3.1 on page 553.
- Section B.4 on page 554 describes separate state transition rules for statement handles and connection handles that also apply when a XDBC function executes asynchronously.
- Section B.5 on page 554 describes valid state transitions for descriptor handles.

Interpretation of the Tables

An entry under a particular state in the table asserts that it is not a sequencing error to call the XDBC function from that state. (Calling the XDBC function may produce some other error, as described on the appropriate reference manual page.) If the call is successful, the state table entry shows the resulting state.

Function Sequence Errors

A blank entry asserts that it is a sequencing error to call the XDBC function in that state. The function sets SQLSTATE to '**HY010**' (**Function sequence error**); unless the reference manual page specifies that another error code applies. In states in which calling a XDBC function is always an error other than a function sequence error, then the applicable SQLSTATE is shown as the state table entry (for example, '**24000**', which is **Invalid cursor state**).

An entry that contains a state symbol (such as C_0) asserts that it is not a sequencing error to call the XDBC function, and specifies the resulting state.

A state table error could be caused by passing to a XDBC function an invalid handle — a null or unallocated handle or a handle of the wrong type. In these cases, the function returns [SQL_INVALID_HANDLE]. Corresponding state table columns (describing attempted operations on an unallocated handle) have the legend INV_H.

20414 **Notation**

20415 The tables describe input to the XDBC function in parentheses, even though that may not be the
20416 exact syntax used. The tables denote output from the routine, including return status, using an
20417 arrow (→) followed by the specific output.

20418 A general state table entry (one that does not show inputs or outputs) describes all remaining
20419 cases of calls to that routine. These general entries assume the routine returns success. Calls that
20420 return failure do not make state transitions, except where described by specific state table
20421 entries.

20422 The boldfaced headings of some state table columns, such as **prepared**, are referenced elsewhere
20423 in this specification; but the wording of these column headings is not normative.

20424 B.1 Environment State Transitions

20425 A XDBC environment can be in one of only two states: allocated and unallocated.

20426 In the unallocated state, the only valid function on the environment is *SQLAllocHandle*() (which
20427 changes the environment's state to allocated).[33]

20428 In the allocated state, the application can call *SQLFreeHandle*(), *SQLGetEnvAttr*() and
20429 *SQLSetEnvAttr*() on the environment. None of these changes the state of the environment except
20430 that calling *SQLFreeHandle*() changes its state to unallocated.

20431 In the allocated state, the application can also allocate connections, as described in Section B.2 on
20432 page 549.

20433 ________________

20434 33. As described on the reference manual page, certain calls to *SQLAllocHandle*() return a restricted handle that the application can
use only to obtain diagnostic information. The restricted handle is not a separate state of the environment handle, since invalid
20435 uses of the restricted handle return [SQL_INVALID_HANDLE] rather than the function sequence error '**HY**010'.

20436 ## B.2 Connection State Transitions

20437 Each connection handle can be in one of the following states:

20438 C_0 Unallocated.

20439 C_1 Allocated.

20440 C_2 Allocated and connected to a database.

20441 The initial state is C_0.

	Connection States		
	C_0	C_1	C_2
Function	**connection unallocated**	**connection allocated**	**allocated and connected**
SQLAllocHandle(Connection)	C_1		
SQLGetConnectAttr(), *SQLGetConnectOption*(), *SQLGetInfo*(), *SQLSetConnectAttr*(), *SQLSetConnectOption*()	INV_H	C_1 [1]	C_2
SQLDataSources(), *SQLGetEnvAttr*(), *SQLGetFunctions*()	INV_H	C_1	C_2
SQLConnect()	INV_H	C_2	'**08**002'
SQLAllocHandle(Descriptor), *SQLAllocHandle*(Statement)	INV_H	'**08**003'	C_2
SQLDisconnect()	INV_H	'**08**003'	C_1
SQLFreeHandle(Connection)	INV_H	C_0	

20460 Table **B-1.** State Table for Connection Handles

20461 **Notes:**

20462 [1] If the operation (getting or setting the specified connection attribute, or getting the specified *SQLGetInfo*() item) requires an existing connection, then a call from this connection state raises '**08**003'.

20465 When a connection is in state C_2, the application can allocate statement handles whose states are governed by Table B-2 on page 551, and can allocate descriptor handles whose states are discussed in Section B.5 on page 554.

20468 ## B.3 Statement Transitions

20469 A statement handle (of type SQLHSTMT) can be in one of the following states:

20470 S_0 Not allocated.

20471 S_1 Allocated.

20472 S_2 Prepared (whether or not the dynamic parameters are set and columns are bound).

20473 S_3 Executed, or cursor open but not positioned on a row.

20474 S_4 Cursor positioned on a row.

20475 S_5 through S_7 refer to the data-at-execute dialogue and jump to the state table in Section B.3.1 on
20476 page 553.

20477 All statement handles are initially in state S_0.

20478 *SQLCancel*() is not included in this table because it does not cause any state transition, although
20479 a state transition may occur when the cancelled function returns. This is subject to special rules
20480 defined in *SQLCancel*(). *SQLCancel*() is included in Table B-3 on page 553 and Table B-4 on page
20481 554 to illustrate specific uses of *SQLCancel*().

The numbers in [] refer to the notes following the table.

	Statement Handle States				
	S_0	S_1	S_2	S_3	S_4
Function	**not allocated**	**allocated**	**prepared**	**executed**	**cursor positioned**
SQLAllocHandle(Statement) [1]	S_1				
SQLSetCursorName()	INV_H	S_1	S_2	**'24000'**	**'24000'**
SQLGetCursorName()	INV_H	S_1	S_2	S_3	S_4
SQLGetStmtAttr(CURRENT_OF_POSITION), *SQLSetStmtAttr*(CURRENT_OF_POSITION)	INV_H	**'24000'**	**'24000'**	**'24000'**	S_4
SQLBindCol(), *SQLBindParam*() *SQLGetStmtAttr*(), *SQLGetStmtOption*(), *SQLSetStmtAttr*(), *SQLSetStmtOption*()	INV_H	S_1	S_2	S_3	S_4
SQLPrepare() [2]	INV_H	S_2	S_2	S_2[3]	**'24000'**
SQLColAttribute(), *SQLDescribeCol*(), *SQLNumResultCols*()	INV_H		S_2	S_3	S_4
SQLExecute() → [SQL_NEED_DATA] [5]	n/a [7]		S_5	S_5[3]	**'24000'**
SQLExecute() [5]	INV_H		S_3	S_3[3]	**'24000'**
SQLExecDirect() → [SQL_NEED_DATA] [5]	n/a [7]	S_5	S_5	S_5[3]	**'24000'**
SQLExecDirect() [5]	INV_H	S_3	S_3	S_3[3]	**'24000'**
SQLGetData()	INV_H			S_3	
SQLRowCount()				S_3	S_4
SQLFetch(), *SQLFetchScroll*()	INV_H			S_4	S_4
SQLGetData()	INV_H				S_4
SQLCloseCursor()	INV_H			S_2	S_2
SQLMoreResults() → [SQL_NO_DATA]	n/a [7]	S_1	S_2	S_3	'S_2
SQLMoreResults()	INV_H	n/a [8]	n/a [8]	n/a [8]	'S_4
Result-set Functions [4]	INV_H	S_3	S_3	S_3[3]	**'24000'**
SQLFreeHandle(Statement)	INV_H	S_0	S_0	S_0	S_0
SQLEndTran() [6]	S_0	S_1	S_1	S_1	S_1
SQLDisconnect() [6]	S_0	S_0	**'25000'**	**'25000'**	**'25000'**

Table **B-2.** State Table for Statement Handles

20514 **Notes:**

20515 [1] The connection handle that *SQLAllocHandle*() references must be in state C_2; see
20516 Table B-1 on page 549.

20517 [2] For WHERE CURRENT OF *cursor*, the separate statement handle that was used to
20518 open *cursor* must be in state S_3 or S_4 for *SQLPrepare*(). After *SQLPrepare*(), that
20519 statement handle remains in the same state.

20520 [3] In state S_3, a statement may be reprepared, and a XDBC function that returns a
20521 result set [4] can be called, only if there are no open cursors (that is, any XDBC
20522 function that returns a result set [4] on the statement handle has been followed by
20523 a *SQLCloseCursor*()).

20524 [4] Result-set functions include the catalog functions (see Chapter 7),
20525 *SQLGetTypeInfo*(), and a function that executes an SQL statement that returns a
20526 result set.

20527 [5] For WHERE CURRENT OF *cursor*, the separate statement handle that was used to
20528 open *cursor* must be in state S_4. After *SQLExecute*() or *SQLExecDirect*(), that
20529 statement handle remains in state S_4.

20530 [6] These functions do not explicitly specify a statement handle. The entry for
20531 *SQLEndTran*() shows state transitions for all statement handles allocated in the
20532 scope (environment or connection) specified in the call. The entry for
20533 *SQLDisconnect*() (which specifies a connection) shows state transitions for all
20534 statement handles allocated on that connection.

20535 [7] Not applicable; the return value assumed by this row of the state table would
20536 never be returned in this state.

20537 [8] Not applicable; a return value covered by a previous row of the state table would
20538 always be returned in this state.

B.3.1 Data-at-execute Dialogue

An application may set an application parameter descriptor to declare that it will pass the actual data for one or more dynamic parameters at execute time. When an application calls *SQLExecDirect*() and *SQLExecute*() and there is at least one dynamic parameter that needs data, the data-at-execute dialogue begins. (See Section 9.4.3 on page 105 for an overview.) Table B-2 on page 551 illustrates these cases by showing the return value → [SQL_NEED_DATA] and the resulting state S_5.

The following states are involved in the data-at-execute dialogue:

S_5 The application is due to call *SQLParamData*() to determine the identity of the first dynamic parameter for which data is needed.

S_6 The application is due to call *SQLPutData*() to supply the first part (or all of) a dynamic argument.

S_7 The application has called *SQLPutData*() at least once for the current dynamic parameter.

The initial state in this table is S_5.

The numbers in [] refer to the notes following the table.

	Statement Handle States		
	S_5	S_6	S_7
Function	**identify parameter needing data**	**provide first piece**	**other calls to *SQLPutData*()**
SQLParamData() → [SQL_NEED_DATA]	S_6	n/a [7]	S_6
SQLParamData()	n/a [8]		S_4
SQLPutData()		S_7	S_7
SQLCancel()	S_1, S_2[10]	S_1, S_2[10]	S_1, S_2[10]
SQLDisconnect() [11]			

Table **B-3.** State Table for Statement Handles (Data-at-Execute Dialogue)

Notes:

[7] Not applicable; the return value assumed by this row of the state table would never be returned in this state.

[8] Not applicable; a return value covered by a previous row of the state table would always be returned in this state.

[10] This line illustrates the use of *SQLCancel*(), typically by the same application, to cancel the data-at-execute dialogue. The statement handle state reverts to the state from which it entered this table: The resulting state is S_1 (**allocated**) if it was *SQLExecDirect*() that originally returned [SQL_NEED_DATA], or S_2 (**prepared**) if it was *SQLExecute*().

[11] This function does not explicitly specify a statement handle. This entry in the table illustrates that it is a state error to disconnect a connection on which there is any statement handle involved in the data-at-execute dialogue.

20578 ## B.4 Asynchrony State Transitions

20579 The asynchrony state is defined on any handle on which a XDBC function (denoted below as
20580 *Fn*()) reports that it is executing asynchronously. If this function takes a connection handle, the
20581 asynchrony state is associated with the connection handle and is independent of the connection
20582 handle state described in Section B.2 on page 549. If it takes a statement handle, the asynchrony
20583 state is associated with the statement handle and is independent of the statement handle state
20584 described in Section B.3 on page 550.

20585 The numbers in [] refer to the notes following the table.

20586–20593

	Asynchrony States		
	A_0	A_1	A_2
	initial call	**still executing**	**asynch. op. cancelled**
Fn() → [SQL_STILL_EXECUTING]	A_1	A_1	A_2
Fn()	A_0[20]	A_0	A_0
SQLCancel()	[21]	A_2	undefined
Certain other XDBC functions [22]	[23]		

20594 Table **B-4.** State Table for Asynchrony

20595 **Notes:**

20596 [20]This situation is permitted, but as it denotes an initial call to a function that does
20597 not report the use of asynchrony, this state table is not relevant.

20598 [21]Calls to *SQLCancel*() are permitted in situations other than asynchrony, subject to
20599 Table B-2 on page 551.

20600 [22]The list of XDBC functions that cannot be called while there is an asynchronous
20601 operation outstanding appears in **Restrictions on Operations during Asynchrony**
20602 on page 122.

20603 [23]Use of these functions is unrestricted in this state, but may be restricted by another
20604 state table.

20605 ## B.5 Descriptor State Transitions

20606 A descriptor can be in one of only two states: allocated and unallocated.

20607 In the unallocated state, the only valid function on the descriptor is *SQLAllocHandle*() (which
20608 changes the descriptor's state to allocated).

20609 In the allocated state, the application can call *SQLCopyDesc*(), *SQLFreeHandle*(),
20610 *SQLGetDescField*(), *SQLGetDescRec*(), *SQLSetDescField*() and *SQLSetDescRec*() on the descriptor.
20611 None of these changes the state of the descriptor except that calling *SQLFreeHandle*() changes its
20612 state to unallocated.

20613 Calling *SQLDisconnect*() frees (changes to the unallocated state) all descriptor handles allocated
20614 on the connection.

Appendix D

Data Types

XDBC defines two sets of data types:

- SQL data types, in which data is stored in the data source (see Section D.1 on page 556)
- C data types, in which data is stored in application buffers (see Section D.2 on page 560).

XDBC defines the C data types and their corresponding XDBC type identifiers. An application specifies the C data type of the buffer that will receive result set data by passing the appropriate C type identifier in *TargetType* in a call to *SQLBindCol*() or *SQLGetData*(). It specifies the C type of the buffer containing a statement parameter by passing the appropriate C type identifier in *ValueType* in a call to *SQLBindParameter*().

Each SQL data type corresponds to an XDBC C data type. Before returning data from the data source, the implementation converts it to the specified C data type. Before sending data to the data source, the implementation converts it from the specified C data type.

Note: The SQL and C data types listed in the sections below are *concise* data types for which each data type is identified by one identifier. Descriptors, however, use a *verbose* data type, in which one identifier can refer to a class of data types; and a type subcode. (For all data types except the date/time and interval types, the concise and verbose data types are the same.) For more information, see **Data Type Identification in Descriptors** on page 574.

This appendix contains the following:

- Typical SQL data types are presented in Section D.1 on page 556.
- C data types are presented in Section D.2 on page 560.
- Attributes of data types — column size, decimal digits, transfer octet length, and display size — are defined in Section D.3 on page 562.
- Detailed information for the interval data types appears not in the preceding sections but in Section D.4 on page 569.
- Pseudo type identifiers and macros are defined in Section D.5 on page 572. This section also describes operations between data types, including considerations for transferring binary data, and the difference between concise and verbose data types.
- Conversion of data from SQL to C data types is specified in Section D.6 on page 576.
- Conversion of data from C to SQL data types is specified in Section D.7 on page 587.

For more information about XDBC data types, see Section 4.4 on page 46.

D.1 SQL Data Types

Each data source defines a set of SQL data types according to the ISO SQL standard. XDBC defines a manifest constant for all standard SQL data types.[34] The application passes this SQL data type identifier as an argument in XDBC functions or retrieves it into a metadata result set. Implementations are responsible for mapping data source-specific SQL data types to XDBC SQL data type identifiers and implementation-defined SQL data type identifiers. The data stored on a data source may be stored in a type specific to that data source.

Each data source defines its own SQL types. The implementation exposes only those SQL data types that the associated data source defines. The application can determine how an implementation maps data source SQL types to the XDBC-defined SQL type identifiers, and any implementation-defined SQL type identifiers, by calling *SQLGetTypeInfo*(). An implementation also returns the SQL data types when describing the data types of columns and parameters through calls to *SQLColAttribute*(), *SQLColumns*(), *SQLDescribeCol*(), *SQLDescribeParam*(), *SQLProcedureColumns*(), and *SQLSpecialColumns*().

Implementations need not support all SQL data types defined in the X/Open **SQL** specification. Furthermore, they may support additional, data-source-specific SQL data types. To determine which data types a data source supports, an application calls *SQLGetTypeInfo*().

The following table lists typical SQL data types. The columns of the table have the following significance:

- **SQL Type Identifier** is a manifest constant by which XDBC refers to the SQL type. This is the value returned in the DATA_TYPEcolumn by a call to *SQLGetTypeInfo*().

- **Typical SQL Data Type** is the equivalent SQL data type from the X/Open **SQL** specification or the ISO SQL standard. In some cases this SQL type specification allows parameters; for example, for some types, precision can be specified. These parameters appear in this column in italics.

  The SQL data type specification is returned in the NAME and CREATE PARAMS column by a call to *SQLGetTypeInfo*(). The NAME column returns the designation, for example, CHAR, while the CREATE PARAMS column returns any parameters.

- **Typical Type Description** describes typical characteristics of the SQL data type and explain the arguments allowed by the SQL type specification.

This table is not normative. It shows commonly used names, ranges, and limits of SQL data types. A given data source may support only some of the listed data types and the characteristics of these may differ from those listed below. Actual characteristics of an SQL data type on any data source may differ from those specified in this table.

---

34. Exceptions are SQL_BIT_VARYING, SQL_TIME_WITH_TIMEZONE, SQL_TIMESTAMP_WITH_TIMEZONE, and SQL_NATIONAL_CHARACTER. Although XDBC defines a manifest constant for SQL_BIT, XDBC defines it with different characteristics from those stated in the ISO SQL standard.

SQL type identifier	Typical SQL data type	Typical type description
SQL_CHAR	CHAR(*n*)	Character string of fixed string length *n*.
SQL_VARCHAR	VARCHAR(*n*)	Variable-length character string with a maximum string length *n*.
SQL_LONGVARCHAR EX	LONG VARCHAR	Variable length character data. Maximum length is data source-dependent.
SQL_DECIMAL	DECIMAL(*p*, *s*)	Signed, exact, numeric value with a precision *p* and scale *s* ($1 \le p \le 15$; $s \le p$). [2,3]
SQL_NUMERIC	NUMERIC(*p*, *s*)	Signed, exact, numeric value with a precision *p* and scale *s* ($1 \le p \le 15$; $s \le p$). [2,3]
SQL_SMALLINT	SMALLINT	Exact numeric value with precision 5 and scale 0 (signed: $-32{,}768 \le n \le 32{,}767$, unsigned: $0 \le n \le 65{,}535$). [1,2]
SQL_INTEGER	INTEGER	Exact numeric value with precision 10 and scale 0 (signed: $2^{31} \le n \le 2^{31} - 1$, unsigned: $0 \le n \le 2^{32} - 1$). [1,2]
SQL_REAL	REAL	Signed, approximate, numeric value with a binary precision 24 (zero or absolute value 10^{-38} to 10^{38}). [2]
SQL_DOUBLE	DOUBLE	Signed, approximate, numeric value with a binary precision 53 (zero or absolute value 10^{-308} to 10^{308}). [2]
SQL_FLOAT	FLOAT	On each implementation, SQL_FLOAT either has the same characteristics as SQL_REAL or it has the same characteristics as SQL_DOUBLE.
SQL_BIT EX	BIT	Single-bit binary data.
SQL_TINYINT EX	TINYINT	Exact numeric value with precision 3 and scale 0 (signed: $-128 \le n \le 127$, unsigned: $0 \le n \le 256$). [1,2]
SQL_BIGINT EX	BIGINT	Exact numeric value with precision 19 (if signed) or 20 (if unsigned) and scale 0 (signed: $-2^{63} \le n \le 2^{63} - 1$, unsigned: $0 \le n \le 2^{64}$). [1,2]
SQL_BINARY EX	BINARY(*n*)	Binary data of fixed length *n*.
SQL_VARBINARY EX	VARBINARY(*n*)	Variable length binary data of maximum length *n*. The maximum is set by the user.
SQL_LONGVARBINARY EX	LONG VARBINARY	Variable length binary data. Maximum length is data source-dependent.

	SQL type identifier	Typical SQL data type	Typical type description
20719–20722	SQL_TYPE_DATE	DATE	Year, month, and day fields, with values constrained as specified in Section D.3.5 on page 568.
20723–20725	SQL_TYPE_TIME	TIME(*p*)	Hour, minute, and second fields, with values constrained as specified in Section D.3.5 on page 568. *p* indicates the seconds precision.
20726–20729	SQL_TYPE_TIMESTAMP	TIMESTAMP(*p*)	Year, month, day, hour, minute, and second fields, with values constrained as specified in Section D.3.5 on page 568. *p* indicates the seconds precision.
20730–20731	SQL_INTERVAL-_MONTH [4]	INTERVAL MONTH(*p*)	Number of months between two dates. *p* is the interval leading precision.
20732–20733	SQL_INTERVAL-_YEAR [4]	INTERVAL YEAR(*p*)	Number of years between two dates. *p* is the interval leading precision.
20734–20735	SQL_INTERVAL-_YEAR_TO_MONTH [4]	INTERVAL YEAR(*p*) TO MONTH	Number of years between two dates. *p* is the interval leading precision.
20736–20737	SQL_INTERVAL_DAY [4]	INTERVAL DAY(*p*)	Number of days between two dates. *p* is the interval leading precision.
20738–20739	SQL_INTERVAL-_HOUR [4]	INTERVAL HOUR(*p*)	Number of hours between two date/times. *p* is the interval leading precision.
20740–20741	SQL_INTERVAL-_MINUTE [4]	INTERVAL MINUTE(*p*)	Number of minutes between two date/times. *p* is the interval leading precision.
20742–20744	SQL_INTERVAL-_SECOND [4]	INTERVAL SECOND(*p*,*q*)	Number of seconds between two date/times. *p* is the interval leading precision and *q* is the interval seconds precision.
20745–20746	SQL_INTERVAL-_DAY_TO_HOUR [4]	INTERVAL DAY(*p*) TO HOUR	Number of days/hours between two date/times. *p* is the interval leading precision.
20747–20748	SQL_INTERVAL-_DAY_TO_MINUTE [4]	INTERVAL DAY(*p*) TO MINUTE	Number of days/hours/minutes between two date/times. *p* is the interval leading precision.
20749–20752	SQL_INTERVAL-_DAY_TO_SECOND [4]	INTERVAL DAY(*p*) TO SECOND(*q*)	Number of days/hours/minutes/seconds between two date/times. *p* is the interval leading precision and *q* is the interval seconds precision.
20753–20754	SQL_INTERVAL-_HOUR_TO_MINUTE [4]	INTERVAL HOUR(*p*) TO MINUTE	Number of hours/minutes between two date/times. *p* is the interval leading precision.

SQL type identifier	Typical SQL data type	Typical type description
SQL_INTERVAL-_HOUR_TO_SECOND [4]	INTERVAL HOUR(*p*) TO SECOND(*q*)	Number of hours/minutes/seconds between two date/times. *p* is the interval leading precision and *q* is the interval seconds precision.
SQL_INTERVAL-_MINUTE_TO_SECOND [4]	INTERVAL MINUTE(*p*) TO SECOND(*q*)	Number of minutes/seconds between two date/times. *p* is the interval leading precision and *q* is the interval seconds precision.

EX This SQL data type is an extension to those defined in the X/Open **SQL** specification.

[1] An application uses *SQLGetTypeInfo*() or *SQLColAttribute*() to determine if a specified data type or result set column is unsigned.

[2] Precision refers to the total number of digits and scale refers to the number of digits to the right of the decimal point.

[3] SQL_DECIMAL and SQL_NUMERIC data types differ only in their precision. The precision of a DECIMAL(*p,s*) is implementation-defined but at least *p*. The precision of a NUMERIC(*p,s*) is exactly *p*.

[4] For more information on the interval SQL data types, see Section D.4 on page 569.

An application calls *SQLGetTypeInfo*() to determine which data types are supported by a data source and the characteristics of those data types.

D.2 C Data Types

Data is stored in the application in XDBC C data types.

The C data type is specified in the *SQLBindCol*() and *SQLGetData*() functions with *TargetType* and in the *SQLBindParameter*() function with *ValueType*. It can also be specified by calling *SQLSetDescField*() to set the SQL_DESC_TYPE field of an ARD or APD, or by calling *SQLSetDescRec*() with *Type*, and with *DescriptorHandle* set to the handle of an ARD or APD).

The following table lists valid type identifiers for the C data types. The table also lists the XDBC C data type that corresponds to each identifier and the definition of this data type.

C type identifier	XDBC C Typedef	C type
SQL_C_CHAR	SQLCHAR *	**unsigned char** *
SQL_C_SSHORT	SQLSMALLINT	**short int**
SQL_C_USHORT	SQLUSMALLINT	**unsigned short int**
SQL_C_SLONG	SQLINTEGER	**long int**
SQL_C_ULONG	SQLUINTEGER	**unsigned long int**
SQL_C_FLOAT	SQLREAL	**float**
SQL_C_DOUBLE	SQLDOUBLE, SQLFLOAT	**double**
SQL_C_BIT	SQLCHAR	**unsigned char**
SQL_C_STINYINT	SQLSCHAR	**signed char**
SQL_C_UTINYINT	SQLCHAR	**unsigned char**
SQL_C_SBIGINT	SQLBIGINT	**int64**
SQL_C_UBIGINT	SQLUBIGINT	**unsigned int64**
SQL_C_BINARY	SQLCHAR *	**unsigned char** *
SQL_C_VAR-BOOKMARK	VARBOOKMARK	**unsigned char** *
SQL_C_TYPE-DATE	SQL_DATE_STRUCT	**struct tagSQL_DATE_STRUCT {** **SQLSMALLINT** *year*; **UWORD** *month*; **UWORD** *day*; **}**
SQL_C_TYPE-TIME	SQL_TIME_STRUCT	**struct tagSQL_TIME_STRUCT {** **UWORD** *hour*; **UWORD** *minute*; **UWORD** *second*; **}**
SQL_C_TYPE-TIMESTAMP	SQL_TIMESTAMP_STRUCT	**struct tagSQL_TIMESTAMP_STRUCT {** **SQLSMALLINT** *year*; **UWORD** *month*; **UWORD** *day*; **UWORD** *hour*;

		UWORD *minute*; **UWORD** *second*; **UDWORD** *fraction*; **}**
SQL_C_NUMERIC	SQL_NUMERIC_STRUCT	**struct tagSQL_NUMERIC_STRUCT {** **BYTE** *precision*; **BYTE** *scale*; **BYTE** *sign*; **BYTE val[**MAXNUMERICLEN**];**[a] **}**
SQL_C_INTERVAL_*	SQL_INTERVAL_STRUCT	See Section D.4 on page 569.

[a] A number is stored in the *val* field of the SQL_NUMERIC_STRUCT structure as a scaled integer, in little-endian mode (the first octet contains the low-order part of the number).

D.2.1 Date/time Structures

The values of the fields of the SQL_DATE_STRUCT, SQL_TIME_STRUCT, and SQL_TIMESTAMP_STRUCT are constrained as specified in Section D.3.5 on page 568. Since the constraint on seconds is to be within the range from 0 up to but not including 62, the *second* field must be in the range from 0 to 61, inclusive. The *fraction* field is a number of nanoseconds. It must be in the range from 0 up to and including 999,999,999.

D.2.2 64-bit Integer Structures

The SQL_C_SBIGINT and SQL_C_UBIGINT data type identifiers denote 64-bit integers. If the C compiler supports 64-bit integers natively, the implementation should define these data types to be the native 64-bit integer type. If not, the implementation should define the following structures to ensure access to data of these types:

```
typedef struct{
  DWORD dwLowWord;
  DWORD dwHighWord;
  } SQLUBIGINT

typedef struct{
  SQLINTEGER sdwLowWord;
  SQLINTEGER sdwHighWord;
  } SQLBIGINT
```

All such structures should be aligned to an 8-octet boundary.

20844 ## D.3 Attributes of Data Types

20845 Data types are characterized by their column (or parameter) size, decimal digits, length, and
20846 display size. The following XDBC functions return these attributes for a parameter in an SQL
20847 statement or an SQL data type on a data source. Each XDBC function returns a different set of
20848 these attributes, as described below.

20849 • *SQLDescribeCol*() returns the column size and decimal digits of the columns it describes.

20850 • *SQLDescribeParam*() returns the parameter size and decimal digits of the parameters it
20851 describes. Note that *SQLBindParameter*() sets the parameter size and decimal digits for a
20852 parameter in an SQL statement.

20853 • The catalog functions *SQLColumns*(), *SQLProcedureColumns*(), and *SQLGetTypeInfo*() return
20854 catalog attributes for a column in a table, result set, or procedure and the catalog attributes of
20855 the data types in the data source. *SQLColumns*() returns the column size, decimal digits, and
20856 length of a column in specified tables. *SQLProcedureColumns*() returns the column size,
20857 decimal digits, and length of a column in a procedure. *SQLGetTypeInfo*() returns the
20858 maximum column size and the minimum and maximum scales of an SQL data type on a data
20859 source.

20860 The transfer octet length does not appear directly in any descriptor field. The transfer octet
20861 length is the length in octets. The SQL_DESC_LENGTH field is the length in characters.

20862 The display size value for all data types corresponds to the value of the
20863 SQL_DESC_DISPLAY_SIZE descriptor field.

20864 The catalog functions *SQLColumns*(), *SQLProcedureColumns*(), and *SQLGetTypeInfo*() return
20865 values from the data source's catalog, not from descriptor fields. They can be called before
20866 statement execution. Descriptor fields do not contain valid values about data before statement
20867 execution. In addition, the values for column size, decimal digits, and display type returned by
20868 *SQLColumns*(), *SQLProcedureColumns*(), and *SQLGetTypeInfo*() are different from the values
20869 contained in the descriptor fields.

20870 A call to *SQLColAttribute*() does not return column size or decimal digits as defined in the
20871 sections below. *SQLColAttribute*() returns the SQL_DESC_PRECISION, SQL_DESC_SCALE, and
20872 SQL_DESC_DISPLAY SIZE fields of the implementation row descriptor, in addition to other
20873 rows. For more information about these descriptor fields, see *SQLSetDescField*().

20874 ### D.3.1 Column Size

20875 The column (or parameter) size of data types is defined as follows:

20876 • For numeric data types, the maximum number of digits used by the data type of the column
20877 or parameter, or the precision of the data.

20878 • For character types, the length in characters of the data.

20879 • For binary data types, the length in octetss of the data.

20880 • For the time, timestamp, and interval data types, the number of characters in the character
20881 representation of the data.

20882 The following table defines the column size for each concise SQL data type. For some types,
20883 column size is defined in terms of the interval leading precision, denoted as *p*; and/or the
20884 seconds precision, denoted as *s*.

20885

SQL type identifier	Column size
SQL_CHAR, SQL_VARCHAR	The defined length in characters of the column or parameter. For example, the length of a column defined as CHAR(10) is 10.
SQL_LONGVARCHAR [a]	The maximum length in characters
SQL_DECIMAL, SQL_NUMERIC	The defined number of digits. For example, the precision of a column defined as NUMERIC(10,3) is 10.
SQL_BIT [b]	1
SQL_TINYINT [b]	3
SQL_SMALLINT [b]	5
SQL_INTEGER [b]	10
SQL_BIGINT [b]	19 (if signed) or 20 (if unsigned)
SQL_REAL [b]	24
SQL_FLOAT [b]	53
SQL_DOUBLE [b]	53
SQL_BINARY, SQL_VARBINARY	The defined length in octets of the column or parameter. For example, the length of a column defined as BINARY(10) is 10.
SQL_LONGVARBINARY [a]	The maximum length in octets.
SQL_TYPE_DATE [b]	10 (the number of characters in the yyyy-mm-dd format)
SQL_TYPE_TIME [b]	8 (the number of characters in the hh:mm:ss format)
SQL_TYPE_TIMESTAMP	The number of characters in the yyyy-mm-dd hh-mm-ss[.f...] format. For example, if a timestamp does not use seconds or fractional seconds, the precision is 16 (the number of characters in the yyyy-mm-dd hh:mm format). If a timestamp uses thousandths of a seconds, the precision is 23 (the number of characters in the yyyy-mm-dd hh:mm:ss.fff format).
SQL_INTERVAL_SECOND	p (if $s = 0$) or $p + s + 1$ (if $s > 0$).
SQL_INTERVAL_DAY_TO_SECOND	$9 + p$ (if $s = 0$) or $10 + p + s$ (if $s > 0$).
SQL_INTERVAL_HOUR_TO_SECOND	$6 + p$ (if $s = 0$) or $7 + p + s$ (if $s > 0$).
SQL_INTERVAL_MINUTE_TO_SECOND	$6 + p$ (if $s = 0$) or $7 + p + s$ (if $s > 0$).
SQL_INTERVAL_YEAR, SQL_INTERVAL_MONTH,	p

SQL_INTERVAL_DAY, SQL_INTERVAL_HOUR, SQL_INTERVAL_MINUTE	
SQL_INTERVAL_YEAR_TO_MONTH, SQL_INTERVAL_DAY_TO_HOUR, SQL_INTERVAL_HOUR_TO_MINUTE	3 + *p*
SQL_INTERVAL_DAY_TO_MINUTE	6 + *p*

[a] If the implementation cannot determine the column or parameter length, it returns SQL_NO_TOTAL.

[b] *ColumnSize* in *SQLBindParameter*() is ignored for this data type.

The values returned for the column (or parameter) size do not correspond to the values in any one descriptor field. The values can come from either the SQL_DESC_PRECISION or SQL_DESC_LENGTH field, depending on the type data, as shown in the following table.

SQL type identifier	**Descriptor field corresponding to Column or parameter size**
SQL_CHAR	LENGTH
SQL_VARCHAR	LENGTH
SQL_LONGVARCHAR	LENGTH
SQL_DECIMAL	PRECISION
SQL_NUMERIC	PRECISION
SQL_BIT	LENGTH
SQL_TINYINT	PRECISION
SQL_SMALLINT	PRECISION
SQL_INTEGER	PRECISION
SQL_BIGINT	PRECISION
SQL_REAL	PRECISION
SQL_FLOAT	PRECISION
SQL_DOUBLE	PRECISION
SQL_BINARY	LENGTH
SQL_VARBINARY	LENGTH
SQL_LONGVARBINARY	LENGTH
SQL_DATE	LENGTH
SQL_TIME	LENGTH
SQL_TIMESTAMP	LENGTH
All interval types	None

D.3.2 Decimal Digits

‘‘Decimal digits’’ is defined as follows:

- For decimal and numeric data types, the maximum number of digits to the right of the decimal point. (For these data types, decimal digits is also called the scale of the data.)
- For approximate floating point number columns or parameters, undefined, since the number of digits to the right of the decimal point is not fixed.
- For date/time or interval data that contains a seconds component, the number of digits to the right of the decimal point in the seconds component of the data.

For SQL_DECIMAL and SQL_NUMERIC, the maximum scale is generally the same as the maximum precision. However, some data sources impose a separate limit on the maximum scale. To determine the minimum and maximum scales allowed for a data type, an application calls *SQLGetTypeInfo*().

In the following table, the concise SQL data types for which decimal digits are applicable are listed in the left-hand column. The center column defines the decimal digits for that data type. The right-hand column contains SCALE if the decimal digits comes from the SQL_DESC_SCALE descriptor field, or PRECISION if it comes from the SQL_DESC_PRECISION field.

SQL type identifier	Decimal digits	Source field in descriptor
SQL_DECIMAL, SQL_NUMERIC	The defined number of digits to the right of the decimal point. For example, the scale of a column defined as NUMERIC(10,3) is 3. This can be a negative number to support storage of very large numbers without using exponential notation, as in storing 12000 as 12 with a scale of –3.	SCALE
SQL_BIT,[a] SQL_TINYINT,[a] SQL_SMALLINT,[a] SQL_INTEGER,[a] SQL_BIGINT[a]	0	SCALE
SQL_TYPE_TIME, SQL_TYPE_TIMESTAMP, SQL_INTERVAL_SECOND, SQL_INTERVAL_DAY_TO_SECOND, SQL_INTERVAL_HOUR_TO_SECOND, SQL_INTERVAL_MINUTE_TO_SECOND	The number of digits to the right of the decimal point in the seconds part of the value (that is, fractional seconds). This number cannot be negative.	PRECISION

[a] *DecimalDigits* in *SQLBindParameter*() is ignored for this data type.

D.3.3 Transfer Octet Length

The transfer octet length of a column is the maximum number of octets returned to the application when data is transferred to its default C data type. For character data, the length does not include the null terminator. Note that the length of a column may be different from the number of octets required to store the data on the data source. (C data types are listed in Section D.2 on page 560.)

The values returned for the transfer octet length do not correspond to the values in SQL_DESC_LENGTH or any other one descriptor field. The SQL_DESC_LENGTH field in the descriptor always indicates the length in characters, while the transfer octet length is defined as the length in octets.

The transfer octet length defined for each XDBC SQL data type is shown in the table below.

21008
SQL type identifier	Transfer octet length
SQL_CHAR	The defined length of the column in octets. For example, the length of a column defined as CHAR(10) is 10. [b]
SQL_VARCHAR, SQL_LONGVARCHAR [a]	The maximum length of the column in octets. [b]
SQL_DECIMAL, SQL_NUMERIC	The number of octets required to hold the character representation of this data (maximum number of digits plus two). Since these data types are returned as character strings, characters are needed for the digits, a sign, and a decimal point. For example, the length of a column defined as NUMERIC(10,3) is 12 if the character set is ANSI.
SQL_TINYINT	1
SQL_SMALLINT	2
SQL_INTEGER	4
SQL_BIGINT	The number of octets required to hold the character representation of this data, since this data type is returned as a character string. The character representation consists of 20 characters: 19 for digits and a sign, if signed, or 20 digits, if unsigned. Thus, the length is 20 if the character set is ANSI.
SQL_REAL	4
SQL_FLOAT	8
SQL_DOUBLE	8
SQL_BIT	1
SQL_BINARY	The defined length of the column in octets. For example, the length of a column defined as BINARY(10) is 10.
SQL_VARBINARY, SQL_LONGVARBINARY [a]	The maximum length of the column in octets.
SQL_TYPE_DATE, SQL_TYPE_TIME	6 (the size of the SQL_DATE_STRUCT or SQL_TIME_STRUCT structure).
SQL_TYPE_TIMESTAMP	16 (the size of the SQL_TIMESTAMP_STRUCT structure).
SQL_INTERVAL_*	34 (the size of the interval structure).

21039 [a] If the implementation cannot determine the column or parameter length, it returns SQL_NO_TOTAL.

21041 [b] This is the same value as the descriptor field SQL_DESC_OCTET_LENGTH.

21042 ### D.3.4 Display Size

21043 The display size of a column is the maximum number of characters needed to display data in
21044 character form. The following table defines the display size for each XDBC SQL data type.

SQL type identifier	Display size
SQL_CHAR, SQL_VARCHAR	The defined length of the column in characters. For example, the display size of a column defined as CHAR(10) is 10.
SQL_LONGVARCHAR	The maximum length of the column in characters.
SQL_DECIMAL, SQL_NUMERIC	The precision of the column plus 1 (for the sign) if the scale is 0. The precision of the column plus 2 (for the sign and decimal point) if the scale is greater than 0. For example, the display size of a column defined as NUMERIC(10,3) is 12.
SQL_BIT	1 (1 digit).
SQL_TINYINT	4 if signed (a sign and 3 digits) or 3 if unsigned (3 digits).
SQL_SMALLINT	6 if signed (a sign and 5 digits) or 5 if unsigned (5 digits).
SQL_INTEGER	11 if signed (a sign and 10 digits) or 10 if unsigned (10 digits).
SQL_BIGINT	20 whether or not signed.
SQL_REAL	13 (a sign, 7 digits, a decimal point, the letter E, a sign, and 2 digits).
SQL_FLOAT, SQL_DOUBLE	22 (a sign, 15 digits, a decimal point, the letter E, a sign, and 3 digits).
SQL_BINARY, SQL_VARBINARY	The defined length of the column times 2 (each octet is represented by a 2 digit hexadecimal number). For example, the display size of a column defined as BINARY(10) is 20.
SQL_LONGVARBINARY	The maximum length of the column times 2.
SQL_TYPE_DATE	10 (a date in the format yyyy-mm-dd).
SQL_TYPE_TIME	8 (a time in the format hh:mm:ss).
SQL_TYPE_TIMESTAMP	19 (if the scale of the timestamp is 0) or 20 plus the precision of the timestamp (if the scale is greater than 0). This is the number of characters in the ‘‘yyyy-mm-dd hh:mm:ss[.f...]’’ format. For example, the display size of a column storing thousandths of a second is 23 (the number of characters in ‘‘yyyy-mm-dd hh:mm:ss.fff’’).
SQL_INTERVAL_*	See Section D.4.

21075 [a] If the implementation cannot determine the column or parameter length, it returns
21076 SQL_NO_TOTAL.

21077 **D.3.5 Constraints on Date/time Values**

21078 Fields in a value of a date/time data type are constrained according to the usual rules imposed
21079 by the Gregorian calendar and the 24-hour clock. (Fields in a value of an interval data type are
21080 similarly constrained, but see also Section D.4 on page 569.) These rules are as follows:

21081 • The year field must be between 1 and 9999, inclusive. Years are measured from the year 0
21082 A.D. Some data sources do not support the entire range of years.

21083 • The month field must be between 1 and 12, inclusive.

21084 • The day field must be between 1 and 28, 29, 30, or 31, inclusive, depending on the month
21085 field, and on whether the year field denotes a leap year.

21086 • The hour field must be between 0 and 23, inclusive.

21087 • The minute field must be between 0 and 59, inclusive.

21088 • The seconds field must be from 0 up to but not including 62.

D.4 Interval Data Types

An interval is defined as the difference between two dates and times. Intervals are expressed in one of two different ways. One is a *year-month* interval that expresses intervals in terms of years and an integral number of months. The other is a *day-time* interval that expresses intervals in terms of days, minutes, and seconds. These two types of intervals are distinct and cannot be mixed, because months may have differing numbers of days.

An interval consists of a set of fields. There is an implied ordering among the fields. For example, in an year-to-month interval, the year comes first, followed by the month. Similarly, in a day-to-minute interval, the fields are in the order day, hour, and minute. The first field in an interval type is called as the *high-order* field, or the *leading* field. The last field is called as the *trailing* field.

In all intervals, the values of the fields are constrained as they are for date/time values (see Section D.3.5 on page 568), except that the value of the high-order field is not thus constrained. For example, in an hour-to-minute interval, the hour field need not be in the range from 0 up to and including 23.

There are 13 interval SQL data types and 13 interval C data types, as listed in the table below. Each of the interval C data types uses the same structure, SQL_INTERVAL_STRUCT, to contain the interval data (for more information, see **C Interval Structure** on page 569). For more information on the SQL data types, see Section D.1 on page 556; for more information on the C data types, see Section D.2 on page 560.

Type identifier	Class	Description
MONTH	Year-Month	Number of months between two dates.
YEAR	Year-Month	Number of years between two dates.
YEAR_TO_MONTH	Year-Month	Number of years and months between two dates.
DAY	Day-Time	Number of days between two dates.
HOUR	Day-Time	Number of hours between two date/times.
MINUTE	Day-Time	Number of minutes between two date/times.
SECOND	Day-Time	Number of seconds between two date/times.
DAY_TO_HOUR	Day-Time	Number of days/hours between two date/times.
DAY_TO_MINUTE	Day-Time	Number of days/hours/minutes between two date/times.
DAY_TO_SECOND	Day-Time	Number of days/hours/minutes/seconds between two date/times.
HOUR_TO_MINUTE	Day-Time	Number of hours/minutes between two date/times.
MINUTE_TO_SECOND	Day-Time	Number of minutes/seconds between two date/times.

C Interval Structure

Each of the C interval data types listed in Section D.2 on page 560 uses the same structure to contain the interval data. When *SQLFetch*(), *SQLFetchScroll*(), or *SQLGetData*() is called, the implementation returns data into the SQL_INTERVAL_STRUCT structure, uses the value that was specified by the application for the C data types (in the call to *SQLBindCol*(), *SQLGetData*(), or *SQLBindParameter*()) to interpret the contents of SQL_INTERVAL_STRUCT, and populates the *interval_type* field of the structure with the *enum* value corresponding to the C type. Note that for applications, the *interval_type* field is read-only. When the structure is used for parameter data, the implementation uses the value specified by the application in the SQL_DESC_CONCISE_TYPE field of the ARD to interpret the contents of SQL_INTERVAL_STRUCT even if the application set the value of the *interval_struct* field to a different value.

This structure is defined as follows:

```
typedef struct tagSQL_INTERVAL_STRUCT
{
 SQLINTERVAL interval_type;
```

```
21137        SQLSMALLINT interval_sign;
21138        union {
21139          SQL_YEAR_MONTH_STRUCT   year_month;
21140          SQL_DAY_SECOND_STRUCT  day_second;
21141          } intval;
21142       } SQL_INTERVAL_STRUCT
21143       typedef enum
21144       {
21145        SQL_IS_YEAR = 1
21146        SQL_IS_MONTH = 2
21147        SQL_IS_DAY = 3
21148        SQL_IS_HOUR = 4
21149        SQL_IS_MINUTE = 5
21150        SQL_IS_SECOND = 6
21151        SQL_IS_YEAR_TO_MONTH = 7
21152        SQL_IS_DAY_TO_HOUR = 8
21153        SQL_IS_DAY_TO_MINUTE = 9
21154        SQL_IS_DAY_TO_SECOND = 10
21155        SQL_IS_HOUR_TO_MINUTE = 11
21156        SQL_IS_HOUR_TO_SECOND = 12
21157        SQL_IS_MINUTE_TO_SECOND = 13
21158       } SQLINTERVAL
21159       typedef struct tagSQL_YEAR_MONTH
21160       {
21161        SQLUINTEGER year;
21162        SQLUINTEGER month;
21163        SQLUINTEGER unused1;
21164        SQLUINTEGER unused2;
21165        SQLUINTEGER unused3;
21166       } SQL_YEAR_MONTH_STRUCT
21167       typedef struct tagSQL_DAY_SECOND
21168       {
21169        SQLUINTEGER day;
21170        SQLUINTEGER hour;
21171        SQLUINTEGER minute;
21172        SQLUINTEGER second;
21173        SQLUINTEGER fraction;
21174       } SQL_DAY_SECOND_STRUCT
```

21175 The *interval_type* field of the SQL_INTERVAL_STRUCT can be any of the SQL interval codes
21176 defined above. This field tells the application what structure is held in the union and also what
21177 members of the structure are relevant. The interval_sign field has the value SQL_FALSE if the
21178 interval leading field in the interval is unsigned; if it is SQL_TRUE, then the leading field is
21179 negative. Note that the value in the leading field itself is always unsigned, regardless of the
21180 value of interval_sign. The interval_sign field acts as a sign bit. The fields of the SQLINTERVAL
21181 *enum* are also defined above.

21182 The unused fields in the SQL_YEAR_MONTH_STRUCT structure give it the same size as the
21183 SQL_DAY_SECOND_STRUCT. When storing a value into an SQL_YEAR_MONTH_STRUCT
21184 structure, the implementation sets these unused fields to 0.

Interval Precision

Interval data types follow different rules for precision from other data types. An interval has three types of precision:

- **Interval precision** is not a numeric value but the list of fields that the interval comprises. For example, the interval precision of the type INTERVAL DAY TO SECOND is the list DAY, HOUR, MINUTE, SECOND. There is no descriptor field that holds this value; the interval precision is determined by the interval data type.
- **Interval leading precision** is the numeric precision of the high-order field of the interval. This field is a signed numeric; its precision is a part of the data type declaration of the interval. For example, the declaration: INTERVAL HOUR(5) TO MINUTE specifies an interval leading precision of 5; the high-order field, which is the HOUR field, can take values from -99999 to 99999. The interval leading precision is contained in the SQL_DESC_DATETIME_INTERVAL_PRECISION field of the descriptor area.
- **Seconds precision** applies to any interval data type that has a SECOND field. This is the scale (the number of decimal digits after the decimal point) of the fractional part of the seconds value. Interval seconds precision is contained in the SQL_DESC_PRECISION field of the descriptor.

When an application calls *SQLSetDescField*() to set the SQL_DESC_TYPE field to SQL_INTERVAL, *SQLSetDescField*() initializes certain fields as defined in **Default Values for Certain Data Types** on page 481. The default interval leading precision is 2, and the default interval seconds precision is 6. The application can override this by subsequent calls to *SQLSetDescField*() to set the SQL_DESC_PRECISION and SQL_DESC_DATETIME_INTERVAL_PRECISION fields.

Interval Data Type Length

The following rules are used to determine the length of an interval data type, expressed as a number of characters. The number of octets depends upon the character set. The length includes the following values added together:

- Two characters for every field in the interval that is not the high-order field.
- For the high-order field, the number of characters that is the express or implicit **Interval leading precision** (see above).
- One character for the separator between the fields.
- 1 plus the express or implied **Seconds precision** (see above).

Format of Interval Literals

When an application inserts a value into a character field in the database that represents an interval, the value must follow the format defined in the X/Open **SQL** specification for interval literals. This indicates that the value is an interval literal, and specifies its type and precision. Only values that follow this format can be retrieved from the database and converted to a C interval data type.

An example of a character string that satisfies this requirement and represents an interval of minus five hours is:

```
INTERVAL - '05:00:00.00' HOUR(2) TO SECOND(2)
```

A common syntax for specifying interval literals (and date/time literals) is the XSQL escape clause defined in Section 8.3.1 on page 84.

21228 ## D.5 Using Data Type Identifiers

21229 Applications use data types identifiers to describe their buffers to the implementation and to
21230 retrieve metadata from the implementation. Applications call the following functions to
21231 perform these tasks:

21232 • *SQLBindParameter*(), *SQLBindCol*(), and *SQLGetData*() to describe the C data type of
21233 application buffers.

21234 • *SQLColAttribute*() and *SQLDescribeCol*() to retrieve the SQL data types of result set columns.

21235 • *SQLDescribeParameter*() to retrieve the SQL data types of parameters.

21236 • *SQLColumns*(), *SQLProcedureColumns*(), and *SQLSpecialColumns*() to retrieve the SQL data
21237 types of various schema information

21238 • *SQLGetTypeInfo*() to retrieve a list of supported data types.

21239 **Pseudo Type Identifiers**

21240 For application programming convenience, XDBC defines a number of pseudo type identifiers.
21241 They do not actually correspond to actual data types, but instead resolve to existing data types
21242 depending on the situation.

21243 **Default C Data Types**

21244 If an application specifies SQL_C_DEFAULT in *SQLBindCol*(), *SQLGetData*(), or
21245 *SQLBindParameter*(), the implementation assumes that the C data type of the output or input
21246 buffer corresponds to the SQL data type of the column or parameter to which the buffer is
21247 bound. For each XDBC SQL data type, the following table shows the corresponding, or *default*, C
21248 data type.

21249 **Important:** Portable applications should not use SQL_C_DEFAULT, but should specify the C
21250 type of all buffers. Implementations cannot always correctly determine the default C type for
21251 the following reasons:

21252 • If the data source promotes an SQL data type of a column or parameter, the implementation
21253 cannot determine the original SQL data type of a column or parameter. Therefore, it cannot
21254 determine the corresponding default C data type.

21255 • If the implementation cannot determine whether a particular column or parameter is signed,
21256 as is often the case when this is handled by the data source, the implementation cannot
21257 determine whether the corresponding default C data type should be signed or unsigned.

21258 Because SQL_C_DEFAULT is provided only as a programming convenience, the application
21259 does not lose any capabilities when it specifies the actual C data type.

21260 For each XDBC SQL data type, the following table shows the default C data type.

	SQL type identifier	Default C type identifier
21262	SQL_CHAR	SQL_C_CHAR
21263	SQL_VARCHAR	SQL_C_CHAR
21264	SQL_LONGVARCHAR	SQL_C_CHAR
21265	SQL_DECIMAL	SQL_C_CHAR
21266	SQL_NUMERIC	SQL_C_CHAR
21267	SQL_BIT	SQL_C_BIT
21268	SQL_TINYINT	SQL_C_STINYINT or SQL_C_UTINYINT [a]
21269	SQL_SMALLINT	SQL_C_SSHORT or SQL_C_USHORT [a]
21270	SQL_INTEGER	SQL_C_SLONG or SQL_C_ULONG [a]

SQL_BIGINT	SQL_C_CHAR
SQL_REAL	SQL_C_FLOAT
SQL_FLOAT	SQL_C_DOUBLE
SQL_DOUBLE	SQL_C_DOUBLE
SQL_BINARY	SQL_C_BINARY
SQL_VARBINARY	SQL_C_BINARY
SQL_LONGVARBINARY	SQL_C_BINARY
SQL_TYPE_DATE	SQL_C_TYPE_DATE
SQL_TYPE_TIME	SQL_C_TYPE_TIME
SQL_TYPE_TIMESTAMP	SQL_C_TYPE_TIMESTAMP
SQL_INTERVAL_MONTH	SQL_C_INTERVAL_MONTH
SQL_INTERVAL_YEAR	SQL_C_INTERVAL_YEAR
SQL_INTERVAL_YEAR_TO_MONTH	SQL_C_INTERVAL_YEAR_TO_MONTH
SQL_INTERVAL_DAY	SQL_C_INTERVAL_DAY
SQL_INTERVAL_HOUR	SQL_C_INTERVAL_HOUR
SQL_INTERVAL_MINUTE	SQL_C_INTERVAL_MINUTE
SQL_INTERVAL_SECOND	SQL_C_INTERVAL_SECOND
SQL_INTERVAL_DAY_TO_HOUR	SQL_C_INTERVAL_DAY_TO_HOUR
SQL_INTERVAL_DAY_TO_MINUTE	SQL_C_INTERVAL_DAY_TO_MINUTE
SQL_INTERVAL_DAY_TO_SECOND	SQL_C_INTERVAL_DAY_TO_SECOND
SQL_INTERVAL_HOUR_TO_MINUTE	SQL_C_INTERVAL_HOUR_TO_MINUTE
SQL_INTERVAL_HOUR_TO_SECOND	SQL_C_INTERVAL_HOUR_TO_SECOND
SQL_INTERVAL_MINUTE_TO_SECOND	SQL_C_INTERVAL_MINUTE_TO_SECOND

[a] If the implementation can determine whether the column is signed or unsigned, such as when it is fetching data from the data source or when the data source supports only a signed type or only an unsigned type, but not both, the implementation uses the corresponding signed or unsigned C data type. If the implementation cannot determine whether the column is signed or unsigned, it passes the data value without attempting to validate it numerically.

Bookmark C Data Type

The bookmark C data type is a programming convenience that lets an application retrieve a bookmark. This is its only use and it should not be converted to other data types. An application retrieves a bookmark either from column 0 of the result set with *SQLFetch*(), *SQLFetchScroll*(), or *SQLGetData*(), or by calling *SQLGetStmtAttr*(). For more information, see Section 11.2.4 on page 154.

The following table lists the value of *CType* for the bookmark C data type, the XDBC C data type that implements the bookmark C data type, and the definition of this data type:

C type identifier (*CType*)	XDBC C Typedef	C type
SQL_C_VARBOOKMARK	SQLCHAR *	**binary**

SQL_ARD_TYPE

The SQL_ARD_TYPE type identifier is used to indicate that the data in a buffer will be of the type specified in the SQL_DESC_CONCISE_TYPE field of the ARD. SQL_ARD_TYPE is entered in *TargetType* in a call to *SQLGetData*() instead of a specific data type, and lets an application change the data type of the buffer by changing the descriptor field. This value ties the data type of the **TargetValuePtr* buffer to the descriptor field. (SQL_ARD_TYPE is not entered in a call to *SQLBindCol*() or *SQLBindParameter*() because the type of the bound buffer is already tied to the SQL_DESC_TYPE and SQL_DESC_CONCISE_TYPE fields, and can be changed at any time by changing either of those fields.)

Transferring Data in its Binary Form

Among data sources that use the same data source, an application can safely transfer data in the internal form used by that data source on the same data source and hardware platform. For a given piece of data, the SQL data types must be the same in the source and target data sources. The C data type is SQL_C_BINARY.

When the application calls *SQLFetch*(), *SQLFetchScroll*(), or *SQLGetData*() to retrieve the data from the source data source, the implementation retrieves the data from the data source and transfers it, without conversion, to a storage location of type SQL_C_BINARY. When the application calls *SQLExecute*(), *SQLExecDirect*(), or *SQLPutData*() to send the data to the target data source, the implementation retrieves the data from the storage location and transfers it, without conversion, to the target data source.

Applications that transfer any data (except binary data) in this manner are not interoperable among data sources.

Data Type Identification in Descriptors

The SQL data types listed in Section D.1 on page 556 and the C data types listed in Section D.2 on page 560 are *concise* data types: each identifier refers to a single data type. Descriptors, however, do not use a single value to identify data types. Instead, they use a *verbose* data type, and a type subcode. For all data types except date/time and interval data types, the verbose type identifier is the same as the concise type identifier. For date/time and interval data types, however, data type information is stored in the fields SQL_DESC_CONCISE_TYPE, SQL_DESC_TYPE, and SQL_DESC_DATETIME_INTERVAL_CODE. Setting one of these fields affects the others, as described in *SQLSetDescField*().

The following table shows the concise type identifier, verbose type identifier, and type subcode for each SQL type identifier of a date/time or interval data type. (For all other data types, SQL_DESC_CONCISE_TYPE has the same value as SQL_DESC_TYPE, and SQL_DESC_DATETIME_INTERVAL_CODEis 0.)

SQL_DESC_CONCISE_TYPE	SQL_DESC_TYPE	DATETIME_INTERVAL_CODE
SQL_TYPE_DATE	SQL_DATETIME	SQL_CODE_DATE
SQL_TYPE_TIME	SQL_DATETIME	SQL_CODE_TIME
SQL_TYPE_TIMESTAMP	SQL_DATETIME	SQL_CODE_TIMESTAMP
SQL_INTERVAL_MONTH	SQL_INTERVAL	SQL_CODE_MONTH
SQL_INTERVAL_YEAR	SQL_INTERVAL	SQL_CODE_YEAR
SQL_INTERVAL_YEAR_TO_MONTH	SQL_INTERVAL	SQL_CODE_YEAR_TO_MONTH
SQL_INTERVAL_DAY	SQL_INTERVAL	SQL_CODE_DAY
SQL_INTERVAL_HOUR	SQL_INTERVAL	SQL_CODE_HOUR
SQL_INTERVAL_MINUTE	SQL_INTERVAL	SQL_CODE_MINUTE
SQL_INTERVAL_SECOND	SQL_INTERVAL	SQL_CODE_SECOND
SQL_INTERVAL_DAY_TO_HOUR	SQL_INTERVAL	SQL_CODE_DAY_TO_HOUR
SQL_INTERVAL_DAY_TO_MINUTE	SQL_INTERVAL	SQL_CODE_DAY_TO_MINUTE
SQL_INTERVAL_DAY_TO_SECOND	SQL_INTERVAL	SQL_CODE_DAY_TO_SECOND
SQL_INTERVAL_HOUR_TO_MINUTE	SQL_INTERVAL	SQL_CODE_HOUR_TO_MINUTE
SQL_INTERVAL_HOUR_TO_SECOND	SQL_INTERVAL	SQL_CODE_HOUR_TO_SECOND
SQL_INTERVAL_MINUTE_TO_SECOND	SQL_INTERVAL	SQL_CODE_MINUTE_TO_SECOND

The following table shows the concise type identifier, verbose type identifier, and type subcode for each C type identifier of a date/time or interval data type. (For all other data types, SQL_DESC_CONCISE_TYPE has the same value as SQL_DESC_TYPE, and SQL_DESC_DATETIME_INTERVAL_CODEis 0.)

	SQL_DESC_CONCISE_TYPE	SQL_DESC_TYPE	DATETIME_INTERVAL_CODE
21365	SQL_C_TYPE_DATE	SQL_DATETIME	SQL_CODE_DATE
21366	SQL_C_TYPE_TIME	SQL_DATETIME	SQL_CODE_TIME
21367	SQL_C_TYPE_TIME	SQL_DATETIME	SQL_CODE_TIME
21368	SQL_C_TYPE_TIMESTAMP	SQL_DATETIME	SQL_CODE_TIMESTAMP
21369	SQL_C_INTERVAL_MONTH	SQL_INTERVAL	SQL_CODE_MONTH
21370	SQL_C_INTERVAL_YEAR	SQL_INTERVAL	SQL_CODE_YEAR
21371	SQL_C_INTERVAL_YEAR_TO_MONTH	SQL_INTERVAL	SQL_CODE_YEAR_TO_MONTH
21372	SQL_C_INTERVAL_DAY	SQL_INTERVAL	SQL_CODE_DAY
21373	SQL_C_INTERVAL_HOUR	SQL_INTERVAL	SQL_CODE_HOUR
21374	SQL_C_INTERVAL_MINUTE	SQL_INTERVAL	SQL_CODE_MINUTE
21375	SQL_C_INTERVAL_SECOND	SQL_INTERVAL	SQL_CODE_SECOND
21376	SQL_C_INTERVAL_DAY_TO_HOUR	SQL_INTERVAL	SQL_CODE_DAY_TO_HOUR
21377	SQL_C_INTERVAL_DAY_TO_MINUTE	SQL_INTERVAL	SQL_CODE_DAY_TO_MINUTE
21378	SQL_C_INTERVAL_DAY_TO_SECOND	SQL_INTERVAL	SQL_CODE_DAY_TO_SECOND
21379	SQL_C_INTERVAL_HOUR_TO_MINUTE	SQL_INTERVAL	SQL_CODE_HOUR_TO_MINUTE
21380	SQL_C_INTERVAL_HOUR_TO_SECOND	SQL_INTERVAL	SQL_CODE_HOUR_TO_SECOND
21381	SQL_C_INTERVAL_MINUTE_TO_SECOND	SQL_INTERVAL	SQL_CODE_MINUTE_TO_SECOND

21382 ## D.6 Converting Data from SQL to C Data Types

21383 When an application calls *SQLFetch*(), *SQLFetchScroll*(), or *SQLGetData*(), the implementation
21384 retrieves the data from the data source. If necessary, it converts the data from the data type in
21385 which the data source retrieved it to the data type specified by *TargetType* in *SQLBindCol*() or
21386 *SQLGetData*(). Finally, it stores the data in the location pointed to by *TargetValuePtr* in
21387 *SQLBindCol*() or *SQLGetData*().

21388 The following table shows the supported conversions from SQL data types to C data types. A
21389 solid circle indicates the default conversion for an SQL data type (the C data type to which the
21390 data will be converted when *TargetType* is SQL_C_DEFAULT). A hollow circle indicates a
21391 supported conversion.

21392 The format of the converted data is independent of the locale.

21393 **[Requires a table from the sponsors which has not yet been corrected]**

The tables in the following sections describe how the implementation converts data retrieved from the data source; implementations are required to support conversions to all C data types from the SQL data types that they support. For a given SQL data type, the first column of the table lists the legal input values of *TargetType* in *SQLBindCol*() and *SQLGetData*(). The second column lists the outcomes of a test, often using *BufferLength* in *SQLBindCol*() or *SQLGetData*(), which the implementation performs to determine if it can convert the data. For each outcome, the third and fourth columns list the values placed in the buffers specified by *TargetValuePtr* and *StrLen_or_IndPtr* in *SQLBindCol*() or *SQLGetData*() after the implementation has attempted to convert the data. The last column lists the SQLSTATE returned for each outcome by *SQLFetch*(), *SQLFetchScroll*(), or *SQLGetData*().

If *TargetType* in *SQLBindCol*() or *SQLGetData*() contains an identifier for an XDBC C data type not shown in the table for a given SQL data type, *SQLFetch*(), *SQLFetchScroll*(), or *SQLGetData*() returns SQLSTATE 07006 (Restricted data type attribute violation). If *TargetType* contains an identifier that specifies a conversion from a data-source-specific SQL data type to a C data type and the implementation does not support this conversion, *SQLFetch*(), *SQLFetchScroll*(), or *SQLGetData*() returns SQLSTATE HYC00 (Optional feature not implemented).

Though it is not shown in the tables, the implementation returns SQL_NULL_DATA in the buffer specified by *StrLen_or_IndPtr* when the SQL data value is NULL. For an explanation of the use of *StrLen_or_IndPtr* when multiple calls are made to retrieve data, see *SQLGetData*(). When SQL data is converted to character C data, the character count returned in **StrLen_or_IndPtr* does not include the null terminator. If *TargetValuePtr* is a null pointer, *SQLGetData*() returns SQLSTATE HY009 (Invalid use of null pointer); in *SQLBindCol*(), this unbinds the column.

Terms

The following terms and conventions are used in the tables:

- **Length of data** is the number of octets of C data available to return in **TargetValuePtr*, regardless of whether the data will be truncated before it is returned to the application. For string data, this does not include the null terminator.
- **Display size** is the total number of octets needed to display the data in character format.

SQL to C: Character

The identifiers for the character SQL data types are:

SQL_CHAR
SQL_VARCHAR
SQL_LONGVARCHAR

The following table shows the C data types to which character SQL data may be converted. For an explanation of the columns and terms in the table, see the list above.

C type identifier	Test	**TargetValuePtr*	**StrLen_or-_IndPtr*	SQL-STATE
SQL_C_CHAR	Length of data in octets < *BufferLength*	Data	Length of data	N/A
	Length of data in octets ≥ *BufferLength*	Truncated data	Length of data	01004
SQL_C_STINYINT SQL_C_UTINYINT SQL_C_TINYINT	Data converted without truncation	Data data type	Size of the C	N/A

21438 21439 21440	SQL_C_SBIGINT SQL_C_UBIGINT SQL_C_SSHORT	Data converted with truncation of fractional digits	Truncated data	Size of the C data type	01004
21441 21442 21443 21444	SQL_C_USHORT SQL_C_SHORT SQL_C_SLONG SQL_C_ULONG	Conversion of data would result in loss of whole (as opposed to fractional) digits	Undefined	Undefined	22003
21445 21446	SQL_C_LONG SQL_C_NUMERIC	Data is not a numeric literal	Undefined	Undefined	22018
21447 21448 21449	SQL_C_FLOAT SQL_C_DOUBLE	Data is within the range of the data type to which the number is being converted [a]	Data	Size of the C data type	N/A
21450 21451 21452		Data is outside the range of the data type to which the number is being converted [a]	Undefined	Undefined	22003
21453		Data is not a numeric literal	Undefined	Undefined	22018
21454	SQL_C_BIT	Data is 0 or 1 [a]	Data	1 [b]	N/A
21455 21456		Data is greater than 0, less than 2, and not equal to 1 [a]	Truncated data	1 [b]	01004
21457 21458		Data is less than 0 or greater than or equal to 2 [a]	Undefined	Undefined	22003
21459		Data is not a numeric literal	Undefined	Undefined	22018
21460 21461	SQL_C_BINARY	Length of data in octets ≤ *BufferLength*	Data	Length of data	N/A
21462 21463		Length of data in octets > *BufferLength*	Truncated data	Length of data	01004
21464	SQL_C_TYPE_DATE	Data value is a valid *date-value* [a]	Data	6 [b]	N/A
21465 21466		Data value is a valid *timestamp-value*; time portion is zero [a]	Data	6 [b]	N/A
21467 21468		Data value is a valid *timestamp-value*; time portion is non-zero [a,c]	Truncated data	6 [b]	01004
21469 21470		Data value is not a valid *date-value* or *timestamp-value* [a]	Undefined	Undefined	22007
21471	SQL_C_TYPE_TIME	Data value is a valid *time-value* [a]	Data	6 [b]	N/A
21472 21473 21474		Data value is a valid *timestamp-value*; fractional sections portion is zero [a,d]	Data	6 [b]	N/A
21475 21476 21477		Data value is a valid *timestamp-value*; fractional seconds portion is non-zero [a,d,e]	Truncated data	6	
21478 21479		Data value is not a valid *time-value* or *timestamp-value* [a]	Undefined	Undefined	22007
21480 21481 21482	SQL_C_TYPE- _TIMESTAMP	Data value is a valid *timestamp-value*; fractional seconds portion not truncated [a]	Data	16 [b]	N/A
21483 21484 21485		Data value is a valid *timestamp-value*; fractional seconds portion truncated [a]	Truncated data	16 [b]	01004
21486		Data value is a valid *date-value* [a]	Data [f]	16 [b]	N/A
21487		Data value is a valid *time-value* [a]	Data [g]	16 [b]	N/A

	Data value is not a valid *date-value*, *time-value*, or *timestamp-value* [a]	Undefined	Undefined	22007
SQL_C_INTERVAL_*	Data value is a valid interval value fractional sections portion not truncated	Data	Length of data	N/A
	Data value is a valid interval value; fractional seconds portion truncated	Truncated data	Length of data	01S07
	There was no representation of the data in the interval structure	Undefined	Undefined	22015
	The data value is not a valid interval value	Undefined	Undefined	22018

[a] The value of *BufferLength* is ignored for this conversion. The implementation assumes that the size of **TargetValuePtr* is the size of the C data type.

[b] This is the size of the corresponding C data type.

[c] The time portion of the timestamp-value is truncated.

[d] The date portion of the timestamp-value is ignored.

[e] The fractional seconds portion of the timestamp is truncated.

[f] The time fields of the timestamp structure are set to zero.

[g] The date fields of the timestamp structure are set to the current date.

When character SQL data is converted to numeric, date, time, or timestamp C data, leading and trailing spaces are ignored.

All implementations that support date, time, and timestamp data can convert character SQL data to date, time, or timestamp C data as specified in the previous table. Implementations may be able to convert character SQL data from other, implementation-defined formats to date, time, or timestamp C data. Such conversions are not interoperable among data sources.

SQL to C: Numeric

The identifiers for the numeric SQL data types are:

SQL_BIGINT	SQL_FLOAT	SQL_REAL
SQL_DECIMAL	SQL_INTEGER	SQL_SMALLINT
SQL_DOUBLE	SQL_NUMERIC	SQL_TINYINT

The following table shows the C data types to which numeric SQL data may be converted. For an explanation of the columns and terms in the table, see **Terms** on page 577.

C type identifier	Test	**TargetValuePtr*	**StrLen_or-_IndPtr*	SQL-STATE
SQL_C_CHAR	Display size < *BufferLength*	Data	Length of data	N/A
	Number of whole (as opposed to fractional) digits < *BufferLength*	Truncated data	Length of data	01004
	Number of whole (as opposed to fractional) digits ≥ *BufferLength*	Undefined	Undefined	22003
SQL_C_STINYINT SQL_C_UTINYINT SQL_C_TINYINT	Data converted without truncation [a]	Data	Size of the C data type	N/A

C type identifier	Test	**TargetValuePtr*	**StrLen_or_IndPtr*	SQLSTATE
SQL_C_SBIGINT SQL_C_UBIGINT SQL_C_SSHORT SQL_C_USHORT SQL_C_SHORT SQL_C_SLONG SQL_C_ULONG SQL_C_LONG SQL_C_NUMERIC SQL_C_FLOAT SQL_C_DOUBLE	Data converted with truncation of fractional digits [a]	Truncated data	Size of the C data type	01004
	Conversion of data would result in loss of whole (as opposed to fractional) digits [a]	Undefined	Undefined	22003
	Data is within the range of the data type to which the number is being converted [a]	Data	Size of the C data type	N/A
	Data is outside the range of the data type to which the number is being converted [a]	Undefined	Undefined	22003
SQL_C_BIT	Data is 0 or 1 [a]	Data	1 [b]	N/A
	Data is greater than 0, less than 2, and not equal to 1 [a]	Truncated data	1 [b]	01004
	Data is less than 0 or greater than or equal to 2	Undefined	Undefined	22003
SQL_C_BINARY	Length of data ≤ *BufferLength*	Data	Length of data	N/A
	Length of data > *BufferLength*	Undefined	Undefined	22003
SQL_C_INTERVAL_* [e]	Data value is a valid interval value; fractional seconds portion not truncated	Data	Length of data	N/A
	Data value is a valid interval value; fractional seconds portion truncated	Truncated data	Length of data	01S07
	There was no representation of the data in the interval structure	Undefined	Undefined	22015
	The data value is not a valid interval value	Undefined	Undefined	22018

[a] The value of *BufferLength* is ignored for this conversion. The implementation assumes that the size of **TargetValuePtr* is the size of the C data type.

[b] This is the size of the corresponding C data type.

[c] This conversion is supported only if the interval precision of the interval C type (as indicated by the interval_type field of the interval structure) is a single field (i.e., SQL_IS_YEAR, SQL_IS_MONTH, SQL_IS_DAY, SQL_IS_HOUR, SQL_IS_MINUTE, or SQL_IS_SECOND).

[d] This conversion is supported only for the exact numeric data types (SQL_DECIMAL, SQL_NUMERIC, SQL_TINYINT, SQL_SMALLINT, SQL_INTEGER, and SQL_BIGINT). It is not supported for the approximate numeric data types (SQL_REAL, SQL_FLOAT, or SQL_DOUBLE).

[e] This conversion is supported only for the exact numeric data types (SQL_TINYINT, SQL_SMALLINT, SQL_INTEGER, SQL_BIGINT, SQL_DECIMAL, and SQL_NUMERIC), and only if the interval precision of the interval C type (as indicated by the *interval_type* field of the interval structure) is a single field

SQL to C: Bit

The identifier for the bit SQL data type is SQL_BIT.

The following table shows the C data types to which bit SQL data may be converted. For an explanation of the columns and terms in the table, see **Terms** on page 577.

C type identifier	Test	**TargetValuePtr*	**StrLen_or-_IndPtr*	SQL-STATE
SQL_C_CHAR	*BufferLength* > 1	Data	1	N/A
	BufferLength ≤ 1	Undefined	Undefined	22003
SQL_C_STINYINT	None [a]	Data	Size of the C	N/A

SQL_C_UTINYINT			data type	
SQL_C_TINYINT				
SQL_C_SBIGINT				
SQL_C_UBIGINT				
SQL_C_SSHORT				
SQL_C_USHORT				
SQL_C_SHORT				
SQL_C_SLONG				
SQL_C_ULONG				
SQL_C_LONG				
SQL_C_FLOAT				
SQL_C_DOUBLE				
SQL_C_NUMERIC				
SQL_C_BIT		None [a]	Data	1 [b]
SQL_C_BINARY	*BufferLength* ≥ 1	Data	1	N/A
	BufferLength < 1	Undefined	Undefined	22003

[a] The value of *BufferLength* is ignored for this conversion. The implementation assumes that the size of **TargetValuePtr* is the size of the C data type.

[b] This is the size of the corresponding C data type.

When bit SQL data is converted to character C data, the possible values are '0' and '1'.

SQL to C: Binary

The identifiers for the binary SQL data types are:

SQL_BINARY
SQL_VARBINARY
SQL_LONGVARBINARY

The following table shows the C data types to which binary SQL data may be converted. For an explanation of the columns and terms in the table, see **Terms** on page 577.

C type identifier	Test	**TargetValuePtr*	**StrLen_or-_IndPtr*	SQL-STATE
SQL_C_CHAR	Length of data * 2 < *BufferLength*	Data	Length of data	N/A
	Length of data * 2 ≥ *BufferLength*	Truncated data	Length of data	01004
SQL_C_BINARY	Length of data ≤ *BufferLength*	Data	Length of data	N/A
	Length of data > *BufferLength*	Truncated data	Length of data	01004

When binary SQL data is converted to character C data, each octet of source data is represented as two ASCII characters. These characters are the ASCII character representation of the number in its hexadecimal form. For example, a binary 00000001 is converted to '01' and a binary 11111111 is converted to 'FF'.

The implementation always converts individual octets to pairs of hexadecimal digits and terminates the character string with a null octet. Because of this, if *BufferLength* is even and is less than the length of the converted data, the last octet of the **TargetValuePtr* buffer is not used. (The converted data requires an even number of octets, the next-to-last octet is a null octet, and the last octet cannot be used.)

Note: Application developers are discouraged from binding binary SQL data to a character C data type. This conversion is inefficient and slow.

21623 **SQL to C: Date**

21624 The identifier for the date SQL data type is SQL_TYPE_DATE.

21625 The following table shows the C data types to which date SQL data may be converted. For an
21626 explanation of the columns and terms in the table, see **Terms** on page 577.

C type identifier	Test	**TargetValuePtr*	**StrLen_or-_IndPtr*	SQL-STATE
SQL_C_CHAR	*BufferLength* ≥ 11	Data	10	N/A
	BufferLength < 11	Undefined	Undefined	22003
SQL_C_BINARY	Length of data ≤ *BufferLength*	Data	Length of data	N/A
	Length of data > *BufferLength*	Undefined	Undefined	22003
SQL_C_TYPE_DATE	None [a]	Data	6 [c]	N/A
SQL_C_TYPE_TIMESTAMP	None [a]	Data [b]	16 [c]	N/A

21635 [a] The value of *BufferLength* is ignored for this conversion. The implementation assumes that the size of **TargetValuePtr*
21636 is the size of the C data type.

21637 [b] The time fields of the timestamp structure are set to zero.

21638 [c] This is the size of the corresponding C data type.

21639 When date SQL data is converted to character C data, the resulting string is in the 'yyyy-mm-dd'
21640 format. This format is independent of the locale.

21641 **SQL to C: Time**

21642 The identifier for the time SQL data type is SQL_TYPE_TIME.

21643 The following table shows the C data types to which time SQL data may be converted. For an
21644 explanation of the columns and terms in the table, see **Terms** on page 577.

C type identifier	Test	**TargetValuePtr*	**StrLen_or-_IndPtr*	SQL-STATE
SQL_C_CHAR	*BufferLength* ≥ 9	Data	8	N/A
	BufferLength < 9	Undefined	Undefined	22003
SQL_C_BINARY	Length of data ≤ *BufferLength*	Data	Length of data	N/A
	Length of data > *BufferLength*	Undefined	Undefined	22003
SQL_C_TYPE_TIME	None [a]	Data	6 [c]	N/A
SQL_C_TYPE_TIMESTAMP	None [a]	Data [b]	16 [c]	N/A

21653 [a] The value of *BufferLength* is ignored for this conversion. The implementation assumes that the size of **TargetValuePtr*
21654 is the size of the C data type.

21655 [b] The date fields of the timestamp structure are set to the current date and the fractional seconds field of the
21656 timestamp structure is set to zero.

21657 [c] This is the size of the corresponding C data type.

21658 When time SQL data is converted to character C data, the resulting string is in the 'hh:mm:ss'
21659 format. This format is independent of the locale.

21660 **SQL to C: Timestamp**

21661 The identifier for the timestamp SQL data type is SQL_TYPE_TIMESTAMP.

21662 The following table shows the C data types to which timestamp SQL data may be converted. For
21663 an explanation of the columns and terms in the table, see **Terms** on page 577.

21664 21665	C type identifier	Test	**TargetValuePtr*	**StrLen_or-_IndPtr*	SQL-STATE
21666	SQL_C_CHAR	*BufferLength* > Display size	Data	Length of data	N/A
21667		20 ≤ *BufferLength* ≤ Display size	Truncated data [b]	Undefined	01004
21668		*BufferLength* > 20	Undefined	Undefined	22003
21669	SQL_C_BINARY	Length of data ≤ *BufferLength*	Data	Length of data	N/A
21670		Length of data > *BufferLength*	Undefined	Undefined	22003
21671 21672	SQL_C_TYPE_DATE	Time portion of timestamp is zero [a]	Data	6 [f]	N/A
21673 21674		Time portion of timestamp is non-zero [a]	Truncated data [c]	6 [f]	01004
21675 21676	SQL_C_TYPE_TIME	Fractional seconds portion of timestamp is zero [a]	Data	6 [f]	N/A
21677 21678		Fractional seconds portion of timestamp is non-zero [a]	Truncated data [d,e]	6 [f]	01004
21679 21680	SQL_C_TYPE-_TIMESTAMP	Fractional seconds portion of timestamp is not truncated [a]	Data [e]	16 [f]	N/A
21681 21682		Fractional seconds portion of timestamp is truncated [a]	Truncated data [e]	16 [f]	01004

21683 [a] The value of *BufferLength* is ignored for this conversion. The implementation assumes that the size of **TargetValuePtr*
21684 is the size of the C data type.

21685 [b] The fractional seconds of the timestamp are truncated.

21686 [c] The time portion of the timestamp is truncated.

21687 [d] The date portion of the timestamp is ignored.

21688 [e] The fractional seconds portion of the timestamp is truncated.

21689 [f] This is the size of the corresponding C data type.

21690 When timestamp SQL data is converted to character C data, the resulting string is in the 'yyyy-
21691 mm-dd hh:mm:ss[.f...]' format, where up to nine digits may be used for fractional seconds. This
21692 format is independent of the locale. (Except for the decimal point and fractional seconds, the
21693 entire format must be used, regardless of the precision of the timestamp SQL data type.)

21694 **SQL to C: Interval**

21695 The identifiers for the interval SQL data types are:

21696 SQL_INTERVAL_SECOND SQL_INTERVAL_HOUR
21697 SQL_INTERVAL_DAY_TO_SECOND SQL_INTERVAL_MINUTE
21698 SQL_INTERVAL_HOUR_TO_SECOND SQL_INTERVAL_YEAR_TO_MONTH
21699 SQL_INTERVAL_MINUTE_TO_SECOND SQL_INTERVAL_DAY_TO_HOUR
21700 SQL_INTERVAL_YEAR SQL_INTERVAL_DAY_TO_MINUTE
21701 SQL_INTERVAL_MONTH SQL_INTERVAL_HOUR_TO_MINUTE

21702 SQL_INTERVAL_DAY

21703 The following table shows the C data types to which interval SQL data may be converted. For an
21704 explanation of the columns and terms in the table, see **Terms** on page 577.

C type identifier	Test	**TargetValuePtr*	**StrLen_or-_IndPtr*	SQL-STATE
SQL_C_INTERVAL_* [a]	Data value is a valid interval value; fractional seconds portion not truncated	Data	Length of data	N/A
	Data value is a valid interval value; fractional seconds portion truncated	Truncated data	Length of data	01S07
SQL_C_STINYINT [b] SQL_C_UTINYINT [b] SQL_C_USHORT [b] SQL_C_SHORT [b] SQL_C_SLONG [b] SQL_C_ULONG [b] SQL_C_NUMERIC [b] SQL_C_BIGINT [b]	Interval precision was a single field and the data was converted without truncation.	Data	Size of the C data type	N/A
	Interval precision was a single field and truncated fractional.	Truncated data	Length of data	01004
	Interval precision was a single field and truncated whole.	Truncated data	Length of data	22003
	Interval precision was not a single field.	Undefined	Size of the C data type	
SQL_C_BINARY	Length of data ≤ *BufferLength*	Data	Length of data	N/A
	Length of data > *BufferLength*	Undefined	Undefined	22003
SQL_C_CHAR	Display size < *BufferLength*	Data	Size of the C data type	N/A
	Number of whole (as opposed to fractional) digits < *BufferLength*	Truncated data	Size of the C data type	01004
	Number of whole (as opposed to fractional) digits ≥ *BufferLength*	Undefined	Undefined	22003

21733 [a] A year-month interval SQL type can be converted to any year-month interval C type, and a day-time interval SQL
21734 type can be converted to any day-time interval C type.

21735 [b] If the interval precision is a single field (i.e., one of YEAR, MONTH, DAY, HOUR, MINUTE, or SECOND), then the
21736 interval SQL type can be converted to any exact numeric (i.e., SQL_C_STINYINT, SQL_C_UTINYINT,
21737 SQL_C_USHORT, SQL_C_SHORT, SQL_C_SLONG, SQL_C_ULONG, or SQL_C_NUMERIC).

21738 The default conversion of an interval SQL type is to an interval C type with the same interval
21739 subtype.

21740 The implementation does not inspect the **interval_type** field within the
21741 SQL_INTERVAL_STRUCT interval structure to determine what interval subtype to convert to.
21742 The implementation relies solely on the SQL_DESC_CONCISE_TYPE field of the ARD.
21743 However, the implementation updates the **interval_type** field if the conversion changes the
21744 interval subtype.

21745 To achieve a conversion, the application sets the SQL_DESC_CONCISE_TYPE field in the
21746 appropriate record of the ARD to the appropriate concise type. The application then sets the
21747 SQL_DESC_DATA_PTR field in this ARD record to point to the initialized
21748 SQL_C_INTERVAL_STRUCT structure (or passes a pointer to this structure as *TargetValuePtr* in
21749 *SQLGetData*()).

21750 The following example demonstrates how to transfer data from a column of type
21751 SQL_INTERVAL_DAY_TO_MINUTE into the SQL_C_INTERVAL_STRUCT structure such that
21752 it comes back as a DAY_TO_HOUR interval.

```
21753 SQL_INTERVAL_STRUCT is;
21754 SQLINTEGER          cbValue;
21755 SQLUINTEGER         days, hours;
21756 // Execute a select statement; 'interval_column' is a column
21757 // whose data type is SQL_INTERVAL_DAY_TO_MINUTE.
21758 SQLExecDirect(hstmt, 'SELECT interval_column FROM table', SQL_NTS);
21759 // bind
21760 SQLBindCol(hstmt, 1, SQL_C_INTERVAL_DAY_TO_MINUTE, &is,
21761            sizeof(SQL_INTERVAL_STRUCT), &cbValue);
21762 //fetch
21763 SQLFetch(hstmt);
21764 // process data
21765 days = is.intval.day_second.day;
21766 hours = is.intval.day_second.hour;
```

21767 D.6.1 SQL to C Data Conversion Examples

21768 The following examples illustrate how the implementation converts SQL data to C data:

	SQL type identifier	SQL Data Value	C type identifier	*Buffer Length*	**TargetValuePtr*	SQL-STATE
21769 21770						
21771	SQL_CHAR	abcdef	SQL_C_CHAR	7	abcdef\0 a	N/A
21772	SQL_CHAR	abcdef	SQL_C_CHAR	6	abcde\0 a	01004
21773	SQL_DECIMAL	1234.56	SQL_C_CHAR	8	1234.56\0 a	N/A
21774	SQL_DECIMAL	1234.56	SQL_C_CHAR	5	1234\0 a	01004
21775	SQL_DECIMAL	1234.56	SQL_C_CHAR	4	----	22003
21776	SQL_DECIMAL	1234.56	SQL_C_FLOAT	ignored	1234.56	N/A
21777	SQL_DECIMAL	1234.56	SQL_C_SSHORT	ignored	1234	01004
21778	SQL_DECIMAL	1234.56	SQL_C_STINYINT	ignored	----	22003
21779	SQL_DOUBLE	1.2345678	SQL_C_DOUBLE	ignored	1.2345678	N/A
21780	SQL_DOUBLE	1.2345678	SQL_C_FLOAT	ignored	1.234567	N/A
21781	SQL_DOUBLE	1.2345678	SQL_C_STINYINT	ignored	1	N/A
21782	SQL_TYPE_DATE	1992-12-31	SQL_C_CHAR	11	1992-12-31\0 a	N/A
21783	SQL_TYPE_DATE	1992-12-31	SQL_C_CHAR	10	-----	22003
21784 21785	SQL_TYPE_DATE	1992-12-31	SQL_C_TIMESTAMP	ignored	1992,12,31, 0,0,0,0 b	N/A
21786 21787	SQL_TYPE_TIMESTAMP	1992-12-31 23:45:55.12	SQL_C_CHAR	23	1992-12-31 23:45:55.12\0 a	N/A
21788 21789	SQL_TYPE_TIMESTAMP	1992-12-31 23:45:55.12	SQL_C_CHAR	22	1992-12-31 23:45:55.1\0 a	01004
21790 21791	SQL_TYPE_TIMESTAMP	1992-12-31 23:45:55.12	SQL_C_CHAR	18	----	22003

21792 [a] ‘‘\0’’ represents a null terminator. The implementation always null-terminates SQL_C_CHAR data.

21793 [b] The numbers in this list are the numbers stored in the fields of the TIMESTAMP_STRUCT structure.

21794 ## D.7 Converting Data from C to SQL Data Types

21795 When an application calls *SQLExecute*() or *SQLExecDirect*(), the implementation retrieves the
21796 data for any parameters bound with *SQLBindParameter*() from storage locations in the
21797 application. When an application calls *SQLSetPos*(), the implementation retrieves the data for an
21798 update or add operation from columns bound with *SQLBindCol*(). For data-at-execution
21799 parameters, the application sends the parameter data with *SQLPutData*(). If necessary, the
21800 implementation converts the data from the data type specified by *ValueType* in
21801 *SQLBindParameter*() to the data type specified by *ParameterType* in *SQLBindParameter*(). Finally,
21802 the implementation sends the data to the data source.

21803 The following table shows the supported conversions from C data types to SQL data types. A
21804 solid circle indicates the default conversion for an SQL data type (the C data type from which the
21805 data will be converted when the value of ValueType is SQL_C_DEFAULT). A hollow circle
21806 indicates a supported conversion.

21807 The format of the converted data is independent of the locale.

21808 **[Requires a table from the sponsors which has not yet been corrected]**

The tables in the following sections describe how the implementation converts data sent to the data source; implementations are required to support conversions from all C data types to the SQL data types that they support. For a given C data type, the first column of the table lists the legal input values of *ParameterType* in *SQLBindParameter*(). The second column lists the outcomes of a test that the implementation performs to determine if it can convert the data. The third column lists the SQLSTATE returned for each outcome by *SQLExecDirect*(), *SQLExecute*(), *SQLSetPos*(), or *SQLPutData*(). Data is sent to the data source only if SQL_SUCCESS is returned.

If *ParameterType* in *SQLBindParameter*() contains the identifier of an XDBC SQL data type that is not shown in the table for a given C data type, *SQLBindParameter*() returns SQLSTATE 07006 (Restricted data type attribute violation). If *ParameterType* contains an implementation-defined identifier and the implementation does not support the conversion from the specific C data type to that SQL data type, *SQLBindParameter*() returns SQLSTATE HYC00 (Optional feature not implemented).

If *ParameterValuePtr* and *StrLen_or_IndPtr* in *SQLBindParameter*() are both null pointers, that function returns SQLSTATE HY009 (Invalid use of null pointer). Though it is not shown in the tables, an application sets the value of the length/indicator buffer pointed to by *StrLen_or_IndPtr* in *SQLBindParameter*() or the value of *StrLen_or_IndPtr* in *SQLPutData*() to SQL_NULL_DATA to specify a NULL SQL data value. The application sets these values to SQL_NTS to specify that the value in **ParameterValuePtr* in *SQLBindParameter*() or **DataPtr* in *SQLPutData*() is a null-terminated string.

Terms

The following terms are used in the tables:

- **Length of data** is the number of octets of SQL data available to send to the data source, regardless of whether the data will be truncated before it is sent to the data source. For string data, this does not include the null terminator.
- **Column length** is the number of octets required to store the data at the data source.
- **Display size** is defined for each SQL data type in Section D.3 on page 562.
- **Number of digits** is the number of characters used to represent a number, including the minus sign, decimal point, and exponent (if needed).

C to SQL: Character

The identifier for the character C data type is SQL_C_CHAR.

The following table shows the SQL data types to which C character data may be converted. For an explanation of the columns and terms in the table, see **Terms** on page 588.

SQL type identifier	Test	SQL-STATE
SQL_CHAR SQL_VARCHAR SQL_LONGVARCHAR	Length of data ≤ Column length	N/A
	Length of data > Column length	22001
SQL_DECIMAL SQL_NUMERIC SQL_TINYINT SQL_SMALLINT	Data converted without truncation	N/A
	Data converted with truncation of fractional digits	22001

SQL type	Test	SQLSTATE
SQL_INTEGER SQL_BIGINT	Conversion of data would result in loss of whole (as opposed to fractional) digits	22003
	Data value is not a numeric literal	22018
SQL_REAL SQL_FLOAT SQL_DOUBLE	Data is within the range of the data type to which the number is being converted	N/A
	Data is outside the range of the data type to which the number is being converted	22003
	Data value is not a numeric literal	22018
SQL_BIT	Data is 0 or 1	N/A
	Data is greater than 0, less than 2, and not equal to 1	22001
	Data is less than 0 or greater than or equal to 2	22003
	Data is not a numeric literal	
SQL_BINARY SQL_VARBINARY SQL_LONGVARBINARY	Length of data / 2 ≤ Column length	N/A
	Length of data / 2 > Column length	22001
	Data value is not a hexadecimal value	22018
SQL_TYPE_DATE	Data value is a valid date literal	N/A
	Data value is a valid timestamp literal; time portion is zero	N/A
	Data value is a valid timestamp literal; time portion is non-zero [a]	22001
	Data value is not a valid date literal or timestamp literal	22007
SQL_TYPE_TIME	Data value is a valid time literal	N/A
	Data value is a valid timestamp literal; fractional seconds portion is zero [b]	N/A
	Data value is a valid timestamp literal; fractional seconds portion is non-zero [b,c]	22001
	Data value is not a valid time literal or timestamp literal	22007
SQL_TYPE_TIMESTAMP	Data value is a valid timestamp literal; fractional seconds portion not truncated	N/A

	Data value is a valid timestamp literal; fractional seconds portion truncated	22001
	Data value is a valid date literal [d]	N/A
	Data value is a valid time literal [e]	N/A
	Data value is not a valid date literal, time literal, or timestamp literal	22007
SQL_INTERVAL_*	Data value is a valid interval value; fractional seconds portion not truncated	N/A
	Data value is a valid interval value; fractional seconds portion truncated	22001
	There was no representation of the data in the interval structure	22015
	The data value is not a valid interval value	22018

[a] The time portion of the timestamp is truncated.

[b] The date portion of the timestamp is ignored.

[c] The fractional seconds portion of the timestamp is truncated.

[d] The time portion of the timestamp is set to zero.

[e] The date portion of the timestamp is set to the current date.

When character C data is converted to numeric, date, time, or timestamp SQL data, leading and trailing blanks are ignored.

When character C data is converted to binary SQL data, each two characters are converted to a single octet of binary data. Each two characters represent a number in hexadecimal form. For example, '01' is converted to binary 00000001 and 'FF' is converted to binary 11111111.

The implementation always converts pairs of hexadecimal digits to individual octets and ignores the null terminator. Because of this, if the length of the character string is odd, the last octet of the string (excluding any null terminator) is not converted.

All implementations that support date, time, and timestamp data can convert character C data to date, time, or timestamp SQL data as specified in the previous table. Implementations may be able to convert character C data from other, implementation-defined formats to date, time, or timestamp SQL data. Such conversions are not interoperable among data sources.

Note: Application developers are discouraged from binding character C data to a binary SQL data type. This conversion is inefficient and slow.

C to SQL: Numeric

The identifiers for the numeric XDBC C data types are:

SQL_C_STINYINT SQL_C_SLONG

21921 SQL_C_UTINYINT SQL_C_ULONG
21922 SQL_C_TINYINT SQL_C_LONG
21923 SQL_C_SSHORT SQL_C_FLOAT
21924 SQL_C_USHORT SQL_C_DOUBLE
21925 SQL_C_SHORT SQL_C_NUMERIC
21926 SQL_C_SBIGINT SQL_C_UBIGINT

21927 The following table shows the SQL data types to which numeric C data may be converted. For
21928 an explanation of the columns and terms in the table, see **Terms** on page 588.

SQL type identifier	Test	SQL-STATE
SQL_CHAR SQL_VARCHAR SQL_LONGVARCHAR	Number of digits ≤ Column length	N/A
	Number of whole (as opposed to fractional) digits ≤ Column length	22001
	Number of whole (as opposed to fractional) digits > Column length	22003
SQL_DECIMAL SQL_NUMERIC SQL_TINYINT SQL_SMALLINT SQL_INTEGER SQL_BIGINT	Data converted without truncation	N/A
	Data converted with truncation of fractional or	22003
SQL_REAL SQL_FLOAT SQL_DOUBLE	Data is within the range of the data type to which the number is being converted	N/A
	Data is outside the range of the data type to which the number is being converted	22003
SQL_BIT	Data is 0 or 1	N/A
	Data is greater than 0, less than 2, and not equal to 1	22001
	Data is less than 0 or greater than or equal to 2	22003
SQL_INTERVAL_YEAR [a] SQL_INTERVAL_MONTH [a] SQL_INTERVAL_DAY [a] SQL_INTERVAL_HOUR [a] SQL_INTERVAL_MINUTE [a] SQL_INTERVAL_SECOND [a]	Data value is a valid interval value; fractional seconds portion not truncated	N/A
	Data value is a valid interval value; fractional seconds portion truncated	22001
	There was no representation of the data in the interval structure	22015
	The data value is not a valid interval value	22018

21961 a These conversions are supported only for the exact numeric data types (SQL_C_STINYINT,
21962 SQL_C_UTINYINT, SQL_C_SSHORT, SQL_C_USHORT, SQL_C_SLONG, SQL_C_ULONG,
21963 or SQL_C_NUMERIC). They are not supported for the approximate numeric data types
21964 (SQL_C_FLOAT or SQL_C_DOUBLE). Exact numeric C data types cannot be converted to
21965 an interval SQL type whose interval precision is not a single field.

The implementation ignores the length/indicator value when converting data from the numeric C data types and assumes that the size of the data buffer is the size of the numeric C data type. The length/indicator value is passed in *StrLen_or_Ind* in *SQLPutData*() and in the buffer specified with *StrLen_or_IndPtr* in *SQLBindParameter*(). The data buffer is specified with *DataPtr* in *SQLPutData*() and *ParameterValuePtr* in *SQLBindParameter*().

C to SQL: Bit

The identifier for the bit C data type is SQL_C_BIT.

Bit C data may be converted to the data types listed below. The conversion unconditionally succeeds and the conversion produces no SQLSTATE value.

SQL_BIGINT	SQL_FLOAT	SQL_SMALLINT
SQL_BIT	SQL_INTEGER	SQL_TINYINT
SQL_CHAR	SQL_LONGVARCHAR	SQL_VARCHAR
SQL_DECIMAL	SQL_NUMERIC	
SQL_DOUBLE	SQL_REAL	

The implementation ignores the length/indicator value when converting data from the bit C data type and assumes that the size of the data buffer is the size of the bit C data type. The length/indicator value is passed in *StrLen_or_Ind* in *SQLPutData*() and in the buffer specified with *StrLen_or_IndPtr* in *SQLBindParameter*(). The data buffer is specified with *DataPtr* in *SQLPutData*() and *ParameterValuePtr* in *SQLBindParameter*().

C to SQL: Binary

The identifier for the binary C data type is SQL_C_BINARY.

The following table shows the SQL data types to which binary C data may be converted. For an explanation of the columns and terms in the table, see **Terms** on page 588.

SQL type identifier	Test	SQL-STATE
SQL_CHAR SQL_VARCHAR SQL_LONGVARCHAR	Length of data ≤ Column length	N/A
	Length of data > Column length	22001
SQL_DECIMAL SQL_NUMERIC SQL_TINYINT SQL_SMALLINT SQL_INTEGER SQL_BIGINT SQL_REAL SQL_FLOAT SQL_DOUBLE SQL_BIT SQL_TYPE_DATE SQL_TYPE_TIME SQL_TYPE_TIMESTAMP	Length of data = SQL data length [a]	N/A
	Length of data ≠ SQL data length [a]	22003
SQL_BINARY	Length of data ≤ Column length	N/A

SQL_VARBINARY		
SQL_LONGVARBINARY	Length of data > Column length	N/A

[a] The SQL data length is the number of octets needed to store the data on the data source. (This may be different from the column length, as defined in **Terms** on page 588.)

C to SQL: Date

The identifier for the date C data type is SQL_C_TYPE_DATE.

The following table shows the SQL data types to which date C data may be converted. For an explanation of the columns and terms in the table, see **Terms** on page 588.

SQL type identifier	**Test**	**SQL-STATE**
SQL_CHAR SQL_VARCHAR SQL_LONGVARCHAR	Column length ≥ 10	N/A
	Column length < 10	22003
	Data value is not a valid date	22007
SQL_TYPE_DATE	Data value is a valid date	N/A
	Data value is not a valid date	22007
SQL_TYPE_TIMESTAMP	Data value is a valid date [a]	N/A
	Data value is not a valid date	22007

[a] The time portion of the timestamp is set to zero.

For information about what values are valid in a SQL_C_TYPE_DATE structure, see ‘‘Extended C Data Types,’’ earlier in this appendix.

When date C data is converted to character SQL data, the resulting character data is in the ’yyyy-mm-dd’ format.

The implementation ignores the length/indicator value when converting data from the date C data type and assumes that the size of the data buffer is the size of the date C data type. The length/indicator value is passed in *StrLen_or_Ind* in *SQLPutData*() and in the buffer specified with *StrLen_or_IndPtr* in *SQLBindParameter*(). The data buffer is specified with *DataPtr* in *SQLPutData*() and *ParameterValuePtr* in *SQLBindParameter*().

C to SQL: Time

The identifier for the time C data type is SQL_C_TYPE_TIME.

The following table shows the SQL data types to which time C data may be converted. For an explanation of the columns and terms in the table, see **Terms** on page 588.

SQL type identifier	**Test**	**SQL-STATE**
SQL_CHAR SQL_VARCHAR	Column length ≥ 8	N/A

SQL_LONGVARCHAR	Column length < 8	22003
	Data value is not a valid time	22007
SQL_TYPE_DATE	Data value is a valid time	N/A
	Data value is not a valid time	22007
SQL_TYPE_TIMESTAMP	Data value is a valid time [a]	N/A
	Data value is not a valid time	22007

[a] The date portion of the timestamp is set to the current date and the fractional seconds portion of the timestamp is set to zero.

For information about what values are valid in a SQL_C_TYPE_TIME structure, see ‘‘Extended C Data Types,’’ earlier in this appendix.

When time C data is converted to character SQL data, the resulting character data is in the ’hh:mm:ss’ format.

The implementation ignores the length/indicator value when converting data from the time C data type and assumes that the size of the data buffer is the size of the time C data type. The length/indicator value is passed in *StrLen_or_Ind* in *SQLPutData*() and in the buffer specified with *StrLen_or_IndPtr* in *SQLBindParameter*(). The data buffer is specified with *DataPtr* in *SQLPutData*() and *ParameterValuePtr*in *SQLBindParameter*().

C to SQL: Timestamp

The identifier for the timestamp C data type is SQL_C_TYPE_TIMESTAMP.

The following table shows the SQL data types to which timestamp C data may be converted. For an explanation of the columns and terms in the table, see **Terms** on page 588.

SQL type identifier	**Test**	**SQL-STATE**
SQL_CHAR SQL_VARCHAR SQL_LONGVARCHAR	Column length ≥ Display size	N/A
	19 ≤ Column length < Display size [a]	22001
	Column length < 19	22003
	Data value is not a valid date	22007
SQL_TYPE_DATE	Time fields are zero	N/A
	Time fields are non-zero [b]	22001
	Data value does not contain a valid date	22007
SQL_TYPE_TIME	Fractional seconds fields are zero [c]	N/A
	Fractional seconds fields are non-zero [c,d]	22001
	Data value does not contain a valid time	22007
SQL_TYPE_TIMESTAMP	Fractional seconds fields are not truncated	N/A

22079 | | Fractional seconds fields are truncated [d] | 22001
22080 | | Data value is not a valid timestamp | 22007

22081 [a] The fractional seconds of the timestamp are truncated.
22082 [b] The time fields of the timestamp structure are truncated.
22083 [c] The date fields of the timestamp structure are ignored.
22084 [d] The fractional seconds fields of the timestamp structure are truncated.

22085 For information about what values are valid in a SQL_C_TIMESTAMP structure, see ''Extended
22086 C Data Types,'' earlier in this appendix.

22087 When timestamp C data is converted to character SQL data, the resulting character data is in the
22088 'yyyy-mm-dd hh:mm:ss[.f...]' format.

22089 The implementation ignores the length/indicator value when converting data from the
22090 timestamp C data type and assumes that the size of the data buffer is the size of the timestamp C
22091 data type. The length/indicator value is passed in *StrLen_or_Ind* in *SQLPutData*() and in the
22092 buffer specified with *StrLen_or_IndPtr* in *SQLBindParameter*(). The data buffer is specified with
22093 *DataPtr* in *SQLPutData*() and *ParameterValuePtr* in *SQLBindParameter*().

22094 **C to SQL: Interval**

22095 The identifiers for the interval C data types are:

22096 SQL_C_INTERVAL_MONTH SQL_C_INTERVAL_DAY_TO_HOUR
22097 SQL_C_INTERVAL_YEAR SQL_C_INTERVAL_DAY_TO_MINUTE
22098 SQL_C_INTERVAL_YEAR_TO_MONTH SQL_C_INTERVAL_DAY_TO_SECOND
22099 SQL_C_INTERVAL_DAY SQL_C_INTERVAL_HOUR_TO_MINUTE
22100 SQL_C_INTERVAL_HOUR SQL_C_INTERVAL_HOUR_TO_SECOND
22101 SQL_C_INTERVAL_MINUTE SQL_C_INTERVAL_MINUTE_TO_SECOND
22102 SQL_C_INTERVAL_SECOND

22103 The following table shows the SQL data types to which interval C data may be converted. For an
22104 explanation of the columns and terms in the table, see **Terms** on page 588.

SQL type identifier	Test	SQL-STATE
SQL_CHAR SQL_VARCHAR SQL_LONGVARCHAR	Column length ≥ Display size	N/A
	19 ≤ Column length < Display size	22001
	Column length < 19	22003
	Data value is not a valid date	22007
SQL_TINYINT [b] SQL_SMALLINT [b] SQL_INTEGER [b]	The *type* field in the interval structure is such that the interval is a single field	TBD

SQL type	Test	SQLSTATE
SQL_BIGINT [b] SQL_NUMERIC [b] SQL_DECIMAL [b]	The *type* field in the interval structure is not such that the interval is a single field	
SQL_INTERVAL_* [c]	Data value is a valid interval value; fractional seconds portion not truncated	N/A
	Data value is a valid interval value; fractional seconds portion truncated	22001
	There was no representation of the data in the interval structure	22015
	The data value is not a valid interval value	22018

[a] All C interval data types can be converted to a character data type.

[b] If the type field in the interval structure is such that the interval is a single field, (i.e., SQL_YEAR, SQL_MONTH, SQL_DAY, SQL_HOUR, SQL_MINUTE, or SQL_SECOND), then the interval C type can be converted to any exact numeric (i.e., SQL_TINYINT, SQL_SMALLINT, SQL_INTEGER, SQL_BIGINT, SQL_DECIMAL, or SQL_NUMERIC).

[c] If the type field of the interval structure represents a year-month interval, it can be converted to any year-month SQL interval type. If the type field of the interval structure represents a day-time interval, it can be converted to any day-time SQL interval type.

The default conversion of an interval C type is to an interval SQL type with the same interval subtype.

The implementation ignores the length/indicator value when converting data from the interval C data type and assumes that the size of the data buffer is the size of the interval C data type. The length/indicator value is passed in *StrLen_or_Ind* in *SQLPutData*() and in the buffer specified with *StrLen_or_IndPtr* in *SQLBindParameter*(). The data buffer is specified with *DataPtr* in *SQLPutData*() and *ParameterValuePtr*in *SQLBindParameter*().

The following example demonstrates how to send interval C data stored in the SQL_INTERVAL_STRUCT structure into a database column. The interval structure contains a DAY_TO_SECOND interval; it will be stored in a database column of type SQL_INTERVAL_DAY_TO_MINUTE.

```
SQL_INTERVAL_STRUCT is;
SQLINTEGER          cbValue;

// Initialize the interval struct to contain the DAY_TO_MINUTE
// interval '154 days, 22 hours, and 44 minutes'
// This is for illustration; it is not read by the implementation.
is.interval_type            = SQL_DAY_TO_MINUTE;
is.intval.day_second.day    = 154;
is.intval.day_second.hour   = 22;
is.intval.day_second.minute = 44;
is.interval_sign            = SQL_FALSE;

// Bind the dynamic parameter
SQLBindParameter(hstmt, 1, SQL_PARAM_INPUT, SQL_C_INTERVAL_DAY_TO_MINUTE,
       SQL_INTERVAL_DAY_TO_MINUTE, 0, 0, &is,
       sizeof(SQL_INTERVAL_STRUCT), &cbValue);

// Execute an insert statement; 'interval_column' is a column
// whose data type is SQL_INTERVAL_DAY_TO_HOUR.
```

22160
```
SQLExecDirect(hstmt,'INSERT INTO table(interval_column) VALUES (?)',SQL_NTS);
```

22161
D.7.1 C to SQL Data Conversion Examples

22162 The following examples illustrate how the implementation converts C data to SQL data:

22163
22164

C type identifier	C data value	SQL type identifier	*Column length*	*SQL Data Value*	SQL-State
SQL_C_CHAR	abcdef [a]	SQL_CHAR	6	abcdef	N/A
SQL_C_CHAR	abcdef [a]	SQL_CHAR	5	abcde	22001
SQL_C_CHAR	1234.56 [a]	SQL_DECIMAL	8 [b]	1234.56	N/A
SQL_C_CHAR	1234.56 [a]	SQL_DECIMAL	7 [b]	1234.5	22001
SQL_C_CHAR	1234.56 [a]	SQL_DECIMAL	4	----	22003
SQL_C_FLOAT	1234.56	SQL_FLOAT	not applicable	1234.56	N/A
SQL_C_FLOAT	1234.56	SQL_INTEGER	not applicable	1234	22001
SQL_C_FLOAT	1234.56	SQL_TINYINT	not applicable	----	22003
SQL_C_TYPE_DATE	1992,12,31 [c]	SQL_CHAR	10	1992-12-31	N/A
SQL_C_TYPE_DATE	1992,12,31 [c]	SQL_CHAR	9	----	22003
SQL_C_TYPE_DATE	1992,12,31 [c]	SQL_TIMESTAMP	not applicable	1992-12-31 00:00:00.0	N/A
SQL_C_TYPE_TIMESTAMP	1992,12,31, 23,45,55, 120000000 [d]	SQL_CHAR	22	1992-12-31 23:45:55.12	N/A
SQL_C_TYPE_TIMESTAMP	1992,12,31, 23,45,55, 120000000 [d]	SQL_CHAR	21	1992-12-31 23:45:55.1	22001
SQL_C_TYPE_TIMESTAMP	1992,12,31, 23,45,55, 120000000 [d]	SQL_CHAR	18	----	22003

22165–22184

22185 [a] ''\0'' represents a null terminator. It is required only if the length of the data is SQL_NTS.

22186 [b] In addition to octets for numbers, one octet is required for a sign and another octet is required for the decimal point.

22187 [c] The numbers in this list are the numbers stored in the fields of the SQL_DATE_STRUCT structure.

22188 [d] The numbers in this list are the numbers stored in the fields of the SQL_TIMESTAMP_STRUCT structure.

22189 *Appendix F*

22190
Scalar Functions

22191 Scalar functions are syntactic components of SQL that obtain information and perform
22192 conversions.

22193 **Functions in the X/Open SQL specification**

22194 A data source that complies with the X/Open **SQL** specification provides the following scalar
22195 functions:

22196	CHAR_LENGTH	LOWER	TRANSLATE
22197	CHARACTER_LENGTH	OCTET_LENGTH	TRIM
22198	CONVERT	POSITION	UPPER
22199	EXTRACT	SUBSTRING	

22200 The X/Open **SQL** specification refers to most of these as string operations. The X/Open **SQL**
22201 specification also supports concatenation through the ‖ operator and defines a CAST function
22202 comparable to the CONVERT function defined in Section F.5 on page 609.

22203 **Functions in This Appendix**

22204 The functions in this appendix are optional. An application can call *SQLGetInfo*() to determine
22205 which functions a given data source supports. The details of the call are specified at the start of
22206 each section of this appendix.

22207 If a data source asserts that it supports a given scalar function, the function must be
22208 implemented, syntactically and semantically, as specified in this appendix.

22209 **Using Scalar Functions**

22210 A portable application using scalar functions must account for the possibility that some are not
22211 implemented on a given data source. The application should do both of the following:

22212 • Query the data source using *SQLGetInfo*(), and make its use of the scalar functions
22213 conditional on determining that the data source supports them.

22214 • Code calls to the scalar functions using the XDBC escape clause (see Section 8.3 on page 84)
22215 so that the XDBC implementation passes a syntactic form acceptable to the data source.

22216 In any application algorithm that relies on scalar functions beyond those defined in the X/Open
22217 **SQL** specification, it is possible that the application cannot use the scalar functions on some data
22218 sources, and it is possible that the only indication the application has that a function is
22219 unavailable is the failure of an SQL statement in which the function occurs. The algorithm must
22220 be written to adapt to this possibility.

22221 **Organization of This Appendix**

22222 The scalar functions are organized in terms of the general category of operation:

22223 • **String functions**
22224 Functions that manipulate character strings (including character strings that contain sound
22225 expressions) are listed in Section F.1 on page 601.

22226 • **Numeric functions**
22227 Functions that perform numeric operations, such as trigonometric and transcendental
22228 functions, are listed in Section F.2 on page 603.

22229 • **Time, date, and interval functions**
22230 Functions that extract fields from, and perform arithmetic on, date/time and interval values |
22231 are listed in Section F.3 on page 605.

22232 • **System functions**
22233 Functions that retrieve information from the database are listed in Section F.4 on page 608.

22234 • **CONVERT**
22235 The **CONVERT**() function, which converts a value from one data type to another, is
22236 presented in Section F.5 on page 609.

22237 Within each of these sections, the functions are presented in alphabetic order.

22238 ## F.1 String Functions

22239 This section lists the string manipulation functions. An application can determine which string
22240 functions a data source supports by calling *SQLGetInfo*() with the SQL_STRING_FUNCTIONS
22241 option.

22242 Character string literals used as arguments to scalar functions must be bounded by single
22243 quotes.

22244 Arguments denoted as *string_exp* can be the name of a column, a string literal, or the result of
22245 another scalar function, where the underlying data type can be represented as SQL_CHAR,
22246 SQL_VARCHAR, or SQL_LONGVARCHAR.

22247 Arguments denoted as *start*, *length*, or *count* can be a numeric literal or the result of another
22248 scalar function, where the underlying data type can be represented as SQL_TINYINT,
22249 SQL_SMALLINT, or SQL_INTEGER.

22250 The string functions listed here are 1-based, that is, the first character in the string is character 1.

22251 **ASCII**(*string_exp*)
22252 Returns the ASCII code value of the leftmost character of *string_exp* as an integer.

22253 **CHAR**(*code*)
22254 Returns the character that has the ASCII code value specified by *code*. The value of *code*
22255 should be between 0 and 255; otherwise, the return value is data-source-dependent.

22256 **CONCAT**(*string_exp1*, *string_exp2*)
22257 Returns a character string that is the result of concatenating *string_exp2* to *string_exp1*. The
22258 resulting string is data-source-dependent. For example, if the column represented by
22259 *string_exp1* contained a NULL value, DB2 would return NULL, but SQL Server would return
22260 the non-NULL string.

22261 **DIFFERENCE**(*string_exp1*, *string_exp2*)
22262 Returns an integer value that indicates the difference between the values returned by the
22263 SOUNDEX function for *string_exp1* and *string_exp2*.

22264 **INSERT**(*string_exp1*, *start*, *length*, *string_exp2*)
22265 Returns a character string where *length* characters have been deleted from *string_exp1*
22266 beginning at *start* and where *string_exp2* has been inserted into *string_exp*, beginning at *start*.

22267 **LCASE**(*string_exp*)
22268 Returns a string consisting of *string_exp* in which all upper-case characters have been
22269 converted to lower case.

22270 **LEFT**(*string_exp*, *count*)
22271 Returns the leftmost *count* characters of *string_exp*.

22272 **LENGTH**(*string_exp*)
22273 Returns the number of characters in *string_exp*, excluding trailing blanks.

22274 **LOCATE**(*string_exp1*, *string_exp2*[, *start*])
22275 Returns the starting position of the first occurrence of *string_exp1* within *string_exp2*, or 0 if
22276 there is no occurrence. The search begins at character position *start* (or at the first character
22277 position in *string_exp2*, if *start* is omitted).

22278 **LOCATE_2**(*string_exp1*, *string_exp2*)
22279 Returns the starting position of the first occurrence of *string_exp1* within *string_exp2*, or 0 if
22280 there is no occurrence. The search begins at the first character position in *string_exp2*.

22281 **LTRIM**(*string_exp*)
22282 Returns the characters of *string_exp*, with leading blanks removed.

22283 **REPEAT**(*string_exp*, *count*)

22284 Returns a character string composed of *string_exp* repeated *count* times.

22285 **REPLACE**(*string_exp1*, *string_exp2*, *string_exp3*)

22286 Scan *string_exp1*, replacing all occurrences of *string_exp2* with *string_exp3*.

22287 **RIGHT**(*string_exp*, *count*)

22288 Returns the rightmost *count* characters of *string_exp*.

22289 **RTRIM**(*string_exp*)

22290 Returns the characters of *string_exp* with trailing blanks removed.

22291 **SOUNDEX**(*string_exp*)

22292 Returns a data-source-dependent character string representing the sound of the words in
22293 *string_exp*. For example, SQL Server returns a four digit SOUNDEX code; Oracle returns a
22294 phonetic representation of each word.

22295 **SPACE**(*count*)

22296 Returns a character string consisting of *count* spaces.

22297 **SUBSTRING**(*string_exp*, *start*, *length*)

22298 Returns a character string that is derived from *string_exp* beginning at the character position
22299 specified by *start* for *length* characters.

22300 **UCASE**(*string_exp*)

22301 Returns a string consisting of *string_exp* in which all lower-case characters have been
22302 converted to upper case.

22303 ## F.2 Numeric Functions

22304 This section lists the numeric scalar functions. An application can determine which functions a
22305 data source supports by calling *SQLGetInfo*() with the SQL_NUMERIC_FUNCTIONS option.

22306 Arguments denoted as *numeric_exp* can be the name of a column, the result of another scalar
22307 function, or a numeric literal, where the underlying data type could be represented as
22308 SQL_NUMERIC, SQL_DECIMAL, SQL_TINYINT, SQL_SMALLINT, SQL_INTEGER,
22309 SQL_BIGINT, SQL_FLOAT,SQL_REAL, or SQL_DOUBLE.

22310 Arguments denoted as *float_exp* can be the name of a column, the result of another scalar
22311 function, or a numeric literal, where the underlying data type can be represented as SQL_FLOAT.

22312 Arguments denoted as *integer_exp* can be the name of a column, the result of another scalar
22313 function, or a numeric literal, where the underlying data type can be represented as
22314 SQL_TINYINT, SQL_SMALLINT, SQL_INTEGER, or SQL_BIGINT.

22315 **ABS**(*numeric_exp*)
22316 Returns the absolute value of *numeric_exp*.

22317 **ACOS**(*float_exp*)
22318 Returns the arccosine of *float_exp* as an angle, expressed in radians.

22319 **ASIN**(*float_exp*)
22320 Returns the arcsine of *float_exp* as an angle, expressed in radians.

22321 **ATAN**(*float_exp*)
22322 Returns the arctangent of *float_exp* as an angle, expressed in radians.

22323 **ATAN2**(*float_exp1*, *float_exp2*)
22324 Returns the arctangent of the *x* and *y* coordinates, specified by *float_exp1* and *float_exp2*,
22325 respectively, as an angle, expressed in radians.

22326 **CEILING**(*numeric_exp*)
22327 Returns the smallest integer greater than or equal to *numeric_exp*.

22328 **COS**(*float_exp*)
22329 Returns the cosine of *float_exp*, where *float_exp* is an angle expressed in radians.

22330 **COT**(*float_exp*)
22331 Returns the cotangent of *float_exp*, where *float_exp* is an angle expressed in radians.

22332 **DEGREES**(*numeric_exp*)
22333 Returns the number of degrees converted from *numeric_exp* radians.

22334 **EXP**(*float_exp*)
22335 Returns the exponential value of *float_exp*.

22336 **FLOOR**(*numeric_exp*)
22337 Returns the largest integer less than or equal to *numeric_exp*.

22338 **LOG**(*float_exp*)
22339 Returns the natural logarithm of *float_exp*.

22340 **LOG10**(*float_exp*)
22341 Returns the base-10 logarithm of *float_exp*.

22342 **MOD**(*integer_exp1*, *integer_exp2*)
22343 Returns the remainder (modulus) of *integer_exp1* divided by *integer_exp2*.

22344 **PI**()
22345 Returns the constant value of pi as a floating point value.

22346 **POWER**(*numeric_exp*, *integer_exp*)
22347 Returns the value of *numeric_exp* to the power of *integer_exp*.

22348 **RADIANS**(*numeric_exp*)
22349 Returns the number of radians converted from *numeric_exp* degrees.

22350 **RAND**([*integer_exp*])
22351 Returns a random floating point value, in the range from 0.0 up to but not including 1.0,
22352 using *integer_exp* as the optional seed value.

22353 **ROUND**(*numeric_exp*, *integer_exp*)
22354 Returns *numeric_exp* rounded to *integer_exp* places right of the decimal point. If *integer_exp* is
22355 negative, *numeric_exp* is rounded to | *integer_exp* | places to the left of the decimal point.

22356 **SIGN**(*numeric_exp*)
22357 Returns an indicator of the sign of *numeric_exp*. If *numeric_exp* is less than zero, −1 is
22358 returned. If *numeric_exp* equals zero, 0 is returned. If *numeric_exp* is greater than zero, 1 is
22359 returned.

22360 **SIN**(*float_exp*)
22361 Returns the sine of *float_exp*, where *float_exp* is an angle expressed in radians.

22362 **SQRT**(*float_exp*)
22363 Returns the square root of *float_exp.*

22364 **TAN**(*float_exp*)
22365 Returns the tangent of *float_exp*, where *float_exp* is an angle expressed in radians.

22366 **TRUNCATE**(*numeric_exp*, *integer_exp*)
22367 Returns *numeric_exp*, truncated to *integer_exp* places right of the decimal point. If *integer_exp*
22368 is negative, *numeric_exp* is truncated to | *integer_exp* | places to the left of the decimal point.

22369 ## F.3 Time, Date, and Interval Functions

22370 This section lists the time, date, and interval functions. An application can determine which
22371 time and date functions a data source supports by calling *SQLGetInfo*() with the
22372 SQL_TIMEDATE_FUNCTIONS option. (Aspects of the **TIMESTAMPADD** and
22373 **TIMESTAMPDIFF** scalar functions are individually optional. The extent of support for each
22374 option can also be determined by calling *SQLGetInfo*(), as noted below.)

22375 Arguments denoted as *timestamp_exp* can be the name of a column, the result of another scalar
22376 function, or a time, date, or timestamp literal, where the underlying data type could be
22377 represented as SQL_CHAR, SQL_VARCHAR, SQL_TIME, SQL_DATE, or SQL_TIMESTAMP.

22378 Arguments denoted as *date_exp* can be the name of a column, the result of another scalar
22379 function, or a date or timestamp literal, where the underlying data type could be represented as
22380 SQL_CHAR, SQL_VARCHAR, SQL_DATE, or SQL_TIMESTAMP.

22381 Arguments denoted as *time_exp* can be the name of a column, the result of another scalar
22382 function, or a time or timestamp literal, where the underlying data type could be represented as
22383 SQL_CHAR, SQL_VARCHAR, SQL_TIME, or SQL_TIMESTAMP.

22384 **CURDATE**()
22385 Returns the current date.

22386 **CURTIME**()
22387 Returns the current local time.

22388 **CURTIMESTAMP**(*time_precision*)
22389 Returns the current local date and local time as a timestamp value. The *time_precision*
22390 argument determines the seconds precision of the returned timestamp.

22391 **DAYNAME**(*date_exp*)
22392 Returns a character string containing the data source-specific name of the day (for example,
22393 Sunday, through Saturday or Sun. through Sat. for a data source that uses English, or
22394 Sonntag through Samstag for a data source that uses German) for the day portion of
22395 *date_exp*.

22396 **DAYOFMONTH**(*date_exp*)
22397 Returns the day of the month in *date_exp* as an integer value in the range of 1-31.

22398 **DAYOFWEEK**(*date_exp*)
22399 Returns the day to the week in *date_exp* as an integer value in the range of 1-7, where 1
22400 represents Sunday.

22401 **DAYOFYEAR**(*date_exp*)
22402 Returns the day of the year in *date_exp* as an integer value in the range of 1-366.

22403 **EXTRACT**(*extract_field*, *extract_source*)
22404 Returns the *extract_field* portion of the *extract_source*. The *extract_source* argument is a
22405 date/time or interval expression. The *extract_field* argument can be one of the following
22406 keywords:

22407 SQL_TSI_YEAR
22408 SQL_TSI_MONTH
22409 SQL_TSI_DAY
22410 SQL_TSI_HOUR
22411 SQL_TSI_MINUTE
22412 SQL_TSI_SECOND

22413 **HOUR**(*time_exp*)
22414 Returns the hour in *time_exp* as an integer value in the range of 0-23.

22415 **MINUTE**(*time_exp*)
22416 Returns the minute in *time_exp* as an integer value in the range of 0-59.

22417 **MONTH**(*date_exp*)
22418 Returns the month in *date_exp* as an integer value in the range of 1-12.

22419 **MONTHNAME**(*date_exp*)
22420 Returns a character string containing the data source-specific name of the month (for
22421 example, January through December or Jan. through Dec. for a data source that uses
22422 English, or Januar through Dezember for a data source that uses German) for the month
22423 portion of *date_exp*.

22424 **NOW**()
22425 Returns current date and time as a timestamp value.

22426 **QUARTER**(*date_exp*)
22427 Returns the quarter in *date_exp* as an integer value in the range of 1-4, where 1 represents
22428 January 1 through March 31.

22429 **SECOND**(*time_exp*)
22430 Returns the second in *time_exp* as an integer value in the range of 0 up to but not including
22431 62.

22432 **TIMESTAMPADD**(*interval*, *integer_exp*, *timestamp_exp*)
22433 Returns the timestamp calculated by adding *integer_exp* intervals of type *interval* to
22434 *timestamp_exp*. Valid values of interval are the following keywords:

22435 SQL_TSI_FRAC_SECOND
22436 SQL_TSI_SECOND
22437 SQL_TSI_MINUTE
22438 SQL_TSI_HOUR
22439 SQL_TSI_DAY
22440 SQL_TSI_WEEK
22441 SQL_TSI_MONTH
22442 SQL_TSI_QUARTER
22443 SQL_TSI_YEAR

22444 where fractional seconds are expressed in billionths of a second. For example, the following
22445 SQL statement returns the name of each employee and their one-year anniversary dates:

```
22446 SELECT NAME,
22447   {fn TIMESTAMPADD(SQL_TSI_YEAR, 1, HIRE_DATE)}
22448   FROM EMPLOYEES
```

22449 If *timestamp_exp* is a time value and *interval* specifies days, weeks, months, quarters, or
22450 years, the date portion of *timestamp_exp* is set to the current date before calculating the
22451 resulting timestamp.

22452 If *timestamp_exp* is a date value and *interval* specifies fractional seconds, seconds, minutes, or
22453 hours, the time portion of *timestamp_exp* is set to 0 before calculating the resulting
22454 timestamp.

22455 An application determines which intervals a data source supports by calling *SQLGetInfo*()
22456 with the SQL_TIMEDATE_ADD_INTERVALSoption.

22457 **TIMESTAMPDIFF**(*interval*, *timestamp_exp1*, *timestamp_exp2*)
22458 Returns the integer number of intervals of type interval by which *timestamp_exp2* is greater
22459 than *timestamp_exp1*. Valid values of interval are the following keywords:

22460 SQL_TSI_FRAC_SECOND
22461 SQL_TSI_SECOND
22462 SQL_TSI_MINUTE
22463 SQL_TSI_HOUR
22464 SQL_TSI_DAY
22465 SQL_TSI_WEEK
22466 SQL_TSI_MONTH
22467 SQL_TSI_QUARTER
22468 SQL_TSI_YEAR

22469 where fractional seconds are expressed in billionths of a second. For example, the following
22470 SQL statement returns the name of each employee and the number of years they have been
22471 employed.

```
22472 SELECT NAME,
22473   {fn TIMESTAMPDIFF(SQL_TSI_YEAR, {fn CURDATE()}, HIRE_DATE)}
22474   FROM EMPLOYEES
```

22475 If either timestamp expression is a time value and *interval* specifies days, weeks, months,
22476 quarters, or years, the date portion of that timestamp is set to the current date before
22477 calculating the difference between the timestamps.

22478 If either timestamp expression is a date value and interval specifies fractional seconds,
22479 seconds, minutes, or hours, the time portion of of that timestamp is set to 0 before
22480 calculating the difference between the timestamps.

22481 An application determines which intervals a data source supports by calling *SQLGetInfo*()
22482 with the SQL_TIMEDATE_DIFF_INTERVALSoption.

22483 **WEEK**(*date_exp*) |
22484 Returns the week of the year in *date_exp* as an integer value in the range of 1-53.

22485 **YEAR**(*date_exp*) |
22486 Returns the year in *date_exp* as an integer value. The range is data source-dependent.

22487 ## F.4 System Functions

22488 This section lists the system functions. An application can determine which system functions a
22489 data source supports by calling *SQLGetInfo*() with the SQL_SYSTEM_FUNCTIONS option.

22490 Arguments denoted as *exp* can be the name of a column, the result of another scalar function, or
22491 a literal, where the underlying data type could be represented as SQL_NUMERIC,
22492 SQL_DECIMAL, SQL_TINYINT, SQL_SMALLINT, SQL_INTEGER, SQL_BIGINT, SQL_FLOAT,
22493 SQL_REAL, SQL_DOUBLE, SQL_DATE, SQL_TIME, or SQL_TIMESTAMP.

22494 Arguments denoted as *value* can be a literal constant, where the underlying data type can be
22495 represented as SQL_NUMERIC, SQL_DECIMAL, SQL_TINYINT, SQL_SMALLINT,
22496 SQL_INTEGER, SQL_BIGINT, SQL_FLOAT, SQL_REAL, SQL_DOUBLE, SQL_DATE,
22497 SQL_TIME, or SQL_TIMESTAMP.

22498 Values returned are represented as XDBC data types.

22499 **DATABASE**()
22500 Returns the name of the database corresponding to the connection handle. (The name of the
22501 database is also available by calling *SQLGetConnectAttr*() with the
22502 SQL_CURRENT_QUALIFIER connection attribute.)

22503 **IFNULL**(*exp*, *value*)
22504 If *exp* is null, *value* is returned. If *exp* is not null, *exp* is returned. The possible data type(s) of
22505 value must be compatible with the data type of exp.

22506 **USER**()
22507 Returns the user name in the data source. (The user name is also available as the
22508 SQL_USER_NAME option in *SQLGetInfo*().) The user name may be different from the login
22509 name.

F.5 Explicit Data Type Conversion

The CAST and CONVERT functions both provide explicit data type conversion at the data source. An application can determine whether the data source supports these functions by calling *SQLGetInfo*() with the SQL_CONVERT_FUNCTIONS option. This returns a bitmask in which a specific bit is set to indicate support for the corresponding function.

CAST Function

Data sources that comply with the X/Open **SQL** specification provide the CAST function to convert a value to a different data type. The syntax of CAST is:

```
CAST({expression| NULL} as data-type)
```

where *data-type* is one of the named data types defined in the X/Open **SQL** specification. The pairs of source and destination data types for which conversion via CAST is supported are defined in the X/Open **SQL** specification.

CONVERT Function

This section describes the **CONVERT** scalar function, which converts a value from one data type to another. Support for **CONVERT** is optional; moreover, a data source may support **CONVERT** for only certain combinations of source and target data types, and not support other combinations. An application can determine whether the data source supports conversions between any two data types by calling *SQLGetInfo*() with one of the options beginning with SQL_CONVERT_ listed in **Conversion Information** on page 376. The manifest constant specifies the source data type; *SQLGetInfo*() returns a bitmask specifying the valid target data types for that source. A data source may indicate support for the **CONVERT** scalar function but may be unable to convert between any two data types.

Explicit data type conversion is specified in terms of XDBC SQL data type definitions.

The XDBC syntax for the explicit data type conversion function does not restrict conversions. The validity of specific conversions of one data type to another data type is implementation-defined. The implementation, as it translates the XDBC syntax into the native syntax, rejects conversions that are syntactically valid but not supported by the data source.

The format of the **CONVERT** function is:

CONVERT(*value_exp*, *data_type*)

The function returns the value specified by *value_exp* converted to the specified *data_type*, where *data_type* is one of the following keywords:

SQL_BIGINT
SQL_BINARY
SQL_BIT
SQL_CHAR
SQL_DECIMAL
SQL_DOUBLE
SQL_FLOAT
SQL_INTEGER
SQL_INTERVAL_DAY
SQL_INTERVAL_DAY_TO_HOUR
SQL_INTERVAL_MINUTE_TO_SECOND
SQL_INTERVAL_MONTH
SQL_INTERVAL_SECOND
SQL_INTERVAL_YEAR
SQL_INTERVAL_YEAR_TO_MONTH
SQL_LONGVARBINARY
SQL_LONGVARCHAR
SQL_NUMERIC
SQL_REAL
SQL_SMALLINT

22551 SQL_INTERVAL_DAY_TO_MINUTE SQL_TINYINT
22552 SQL_INTERVAL_DAY_TO_SECOND SQL_TYPE_DATE
22553 SQL_INTERVAL_HOUR SQL_TYPE_TIME
22554 SQL_INTERVAL_HOUR_TO_MINUTE SQL_TYPE_TIMESTAMP
22555 SQL_INTERVAL_HOUR_TO_SECOND SQL_VARBINARY
22556 SQL_INTERVAL_MINUTE SQL_VARCHAR

22557 The XDBC syntax for the explicit data type conversion function does not support specification of
22558 conversion format. If the data source supports specification of explicit formats, the
22559 implementation must either provide a default value or provide an implementation-defined
22560 method of format specification.

22561 The argument *value_exp* can be a column name, the result of another scalar function, or a
22562 numeric or string literal. For example:

```
22563 {fn CONVERT({fn CURDATE()}, SQL_CHAR)}
```

22564 converts the output of the CURDATE scalar function to a character string.

22565 XDBC does not mandate a data type for return values from scalar functions; this is data-source-
22566 specific. Applications should use the CONVERT scalar function whenever possible to force data
22567 type conversion.

22568 **Examples**

22569 The following two examples illustrate the use of the **CONVERT** function. These examples
22570 assume the existence of a table called EMPLOYEES, with an EMPNO column of type
22571 SQL_SMALLINT and an EMPNAME column of type SQL_CHAR.

22572 The examples code the **CONVERT** function using the XDBC escape sequence for scalar function
22573 calls defined in Section 8.3.3 on page 86.

22574 The following SQL statement uses the **CONVERT** function to ensure that the output of the
22575 **CURDATE** function is a date, rather than a timestamp or character data:

```
22576 INSERT INTO Orders (OrderID, CustID, OpenDate, SalesPerson, Status)
22577    VALUES (?, ?, {fn CONVERT({fn CURDATE()}, SQL_DATE)}, ?, ?)
```

22578 If an application specifies the following SQL statement:

```
22579 SELECT EMPNO FROM EMPLOYEES WHERE {fn CONVERT(EMPNO,SQL_CHAR)} LIKE '1%'
```

22580 • Then for an ORACLE data source, the implementation might translate the SQL statement to:

```
22581 SELECT EMPNO FROM EMPLOYEES WHERE to_char(EMPNO) LIKE '1%'
```

22582 For an SQL Server data source, the translation might be:

```
22583 SELECT EMPNO FROM EMPLOYEES WHERE convert(char,EMPNO) LIKE '1%'
```

22584 If an application specifies the following SQL statement:

```
22585 SELECT {fn ABS(EMPNO)}, {fn CONVERT(EMPNAME,SQL_SMALLINT)}
22586    FROM EMPLOYEES WHERE EMPNO <> 0
```

22587 • Then for an ORACLE data source, the implementation might translate the SQL statement to:

```
22588 SELECT abs(EMPNO), to_number(EMPNAME) FROM EMPLOYEES WHERE EMPNO <> 0
```

22589 • For an SQL Server data source, the translation might be:

```
22590 SELECT abs(EMPNO), convert(smallint, EMPNAME) FROM EMPLOYEES
22591    WHERE EMPNO <> 0
```

22592 • For an Ingres data source, the translation might be:

22593
```
SELECT abs(EMPNO), int2(EMPNAME) FROM EMPLOYEES WHERE EMPNO <> 0
```

22594 *Appendix I*

22595 *Driver Manager Implementation (Optional)*

I.1 Introduction

Section 3.2 on page 25 introduced the concept of an XDBC Driver Manager and separate drivers for each type of data source.

The following figure shows how the Driver Manager and drivers connect the application to various data sources:

```
            Application
                 | <-----------XDBC API
          Driver Manager
          /      |       <----XDBC API
     Driver  Driver  Driver
        |       |       |
      Data    Data    Data
     Source  Source  Source
```

There are two things to notice about this diagram. First, multiple drivers and data sources can exist, which gives the application simultaneous access to more than one data source. Second, the XDBC 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 XDBC, 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.

I.1.1 The Driver Manager

The *Driver Manager* is a library that manages communication between applications and drivers. It 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, 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 XDBC 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 (as defined by the state transition tables in Appendix B), and that certain function arguments are valid (as defined in the reference manual pages).

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'. Using this name, the Driver Manager searches the data source information for the driver's file name. It

then loads the driver (assuming it's 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 Chapter 6.

I.1.2 Drivers

Drivers are libraries that implement the functions in the XDBC API. Each is specific to a specific data source (for example, a to a database of a specific vendor) and typically cannot gain direct access to data in a different data source. Drivers expose the capabilities of the underlying data sources; they are not required to implement capabilities not supported by the data source. For example, if the underlying data source does not support outer joins, then neither should the driver. The only major exception to this is that drivers for data sources that do not have standalone database engines 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 XDBC SQL to data-source-specific SQL; this is often limited to replacing escape clauses defined by XDBC with data-source-specific SQL.
- Sending data to and retrieving data from the data source, including converting data types as specified by the application.
- Mapping data-source-specific errors to XDBC SQLSTATEs.

Driver Architecture

Driver architecture falls into two categories, depending on what 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 XDBC calls and SQL statements. For example, a driver may provide access to a file-based data source, or to a data source that has no associated access software. Such a driver must incorporate a database engine capable of processing SQL statements.

- **Data-source-based drivers**

  The driver accesses the physical data through a separate database engine. In this case the driver processes only XDBC calls; it passes SQL statements to the database engine for processing. The database might reside on the same machine as the driver, on a different machine on the network, or through a gateway.

Driver architecture generally matters only to the writers of the driver. However, the architecture can affect whether an application can use data-source-specific SQL. For example, Microsoft Access provides a standalone database engine. If a Microsoft Access driver is data-source-based — that is, it gains access to the data through this engine — the application can pass Microsoft Access-specific SQL statements to the engine for processing.

However, if the driver is file-based — that is, if 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 XDBC SQL.

File-based Drivers

File-based drivers are used with data sources such as dBASE that do not provide a standalone database engine for the driver to use. These drivers access the physical data directly and must implement a database engine to process SQL statements.

In comparing file-based and data-source-based drivers, file-based drivers are harder to write because of the database engine component, less complicated to configure because there are no network pieces, and less powerful because few people have the time to write database engines as powerful as those produced by database companies.

Data-source-based Drivers

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

Because data-source-based drivers use an existing database engine, they are generally easier to write than file-based drivers. Although a data-source-based driver can be easily implemented by translating XDBC calls to native API calls, this results in a slower driver. A better way to implement a data-source-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 XDBC is the native API. For example, Watcom SQL is a standalone engine that resides on the same machine as the application and is loaded directly as the driver.

Data-source-based drivers act as the client in a client-server configuration where the data source acts as the server. Generally, 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 data source 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.

I.2 Choosing a Data Source

The data source 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 wouldn't work with any other data sources. Another example is a vertical application, such as one to do 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 XDBC 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 data-source- or file-based drivers.

Data-source-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 data-source-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.

With file-based drivers, it's possible to use a file paradigm. For data stored on the local machine, users often know that their data is in a particular file. Rather than choosing an unknown data source, it's 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 data-source-based driver. Several common uses of the **DRIVER** keyword for data-source-based drivers are:

- **Not creating data sources**

  A custom application might use a particular driver and database. If the driver name and all information needed to connect to the database are hard-coded in the application, users don't 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 data source a single time**

  For example, a spreadsheet that retrieves data by calling XDBC 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 Section 6.4.5 on page 62.

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 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.The browse result string is:

```
'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 aren't 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,Fran&ccedil;ais};'
```

Since 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:

```
22817 SQLBrowseConnect(hdbc, 'DATABASE=pubs;', SQL_NTS, BrowseResult,
22818                  sizeof(BrowseResult), &BrowseResultLen);
```

22819 Because the **DATABASE** attribute is the final connection attribute required by the driver, the
22820 browsing process is complete, the application is connected to the data source, and
22821 *SQLBrowseConnect*() returns SQL_SUCCESS. *SQLBrowseConnect*() also returns the complete
22822 connection string as the browse result string:

```
22823 'DSN=MySQLServer;SERVER=green;UID=Smith;PWD=Sesame;DATABASE=pubs;'
```

22824 The final connection string returned by the driver doesn't contain the user-friendly names after
22825 each keyword, nor does it contain optional keywords not specified by the application. The
22826 application can use this string with *SQLDriverConnect*() to reconnect to the data source on the
22827 current connection handle (after disconnecting) or to connect to the data source on a different
22828 connection handle. For example:

```
22829 SQLDriverConnect(hdbc, hwnd, BrowseResult, SQL_NTS, ConnStrOut,
22830                  sizeof(ConnStrOut), &ConnStrOutLen, SQL_DRIVER_NOPROMPT);
```

22831 ## I.3 Role of the Driver Manager in the Connection Process

22832 Remember that applications don't call driver functions directly. Instead, they call Driver
22833 Manager functions with the same name and the Driver Manager calls the driver functions.
22834 Usually, this happens almost immediately. For example, the application calls *SQLExecute*() in
22835 Driver Manager and after a few error checks, the Driver Manager calls *SQLExecute*() in the
22836 driver.

22837 The connection process is different. When the application calls *SQLAllocHandle*() with the
22838 SQL_HANDLE_ENV and SQL_HANDLE_DBC options, the function allocates handles only in
22839 the Driver Manager. The Driver Manager doesn't call this function in the driver, because it
22840 doesn't know which driver to call. Similarly, if the application passes the handle of an
22841 unconnected connection to *SQLSetConnectAttr*() or *SQLGetConnectAttr*(), only the Driver
22842 Manager executes the function. It stores or gets the attribute value from its connection handle
22843 and returns SQLSTATE 08003 (Connection not open) when getting a value for an attribute that
22844 hasn't been set and for which XDBC doesn't define a default value.

22845 When the application calls *SQLConnect*(), *SQLDriverConnect*(), or *SQLBrowseConnect*(), the
22846 Driver Manager first determines which driver to use. It then checks if a driver is currently loaded
22847 on the connection:

22848 • If no driver is loaded on the connection, the Driver Manager checks if the specified driver is
22849 loaded on another connection in the same environment. If not, the Driver Manager loads the
22850 driver on the connection and calls *SQLAllocHandle*() in the driver with the
22851 SQL_HANDLE_ENV option.

22852 The Driver Manager then calls *SQLAllocHandle*() in the driver with the SQL_HANDLE_DBC
22853 option, regardless of whether it was just loaded. If the application set any connection
22854 attributes, the Driver Manager calls *SQLSetConnectAttr*() in the driver; if an error occurs, the
22855 Driver Manager's connection function returns SQLSTATE IM006 (Driver's
22856 SQLSetConnectAttr failed). Finally, the Driver Manager calls the connection function in the
22857 driver.

22858 • If the specified driver is loaded on the connection, the Driver Manager only calls the
22859 connection function in the driver. In this case, the driver must make sure that all connection
22860 attributes on the connection maintain their current settings.

22861 • If a different driver is loaded on the connection, the Driver Manager calls *SQLFreeHandle*() in
22862 the driver to free the connection. If there are no other connections that use the driver, the
22863 Driver Manager calls *SQLFreeHandle*() in the driver to free the environment and unloads the
22864 driver. The Driver Manager then performs the same operations as when a driver isn't loaded
22865 on the connection.

22866 When the application calls *SQLDisconnect*(), the Driver Manager calls *SQLDisconnect*() in the
22867 driver. However, it leaves the driver loaded in case the application reconnects to the driver.
22868 When the application calls *SQLFreeHandle*() with the SQL_HANDLE_DBC option, the Driver
22869 Manager calls *SQLFreeHandle*() in the driver. If the driver isn't used by any other connections, the
22870 Driver Manager then calls *SQLFreeHandle*() in the driver with the SQL_HANDLE_ENV option
22871 and unloads the driver.

22872 ## I.4 Other Architectural Issues

22873 **Allocation of Handles**

22874 XDBC has two levels of handles: Driver Manager handles and driver handles. The application
22875 uses Driver Manager handles when calling XDBC functions because it calls those functions in the
22876 Driver Manager. The Driver Manager uses this handle to find the corresponding driver handle
22877 and uses the driver handle when calling the function in the driver. For an example of how driver
22878 and Driver Manager handles are used, see Section I.3 on page 620.

22879 Handles are meaningful only to the XDBC component that created them; that is, only the Driver
22880 Manager can interpret Driver Manager handles and only a driver can interpret its own handles.

22881 That there are two levels of handles is an artifact of the XDBC architecture; it is generally not
22882 relevant to either the application or driver. Although there is generally no reason to do so, it is
22883 possible for the application to determine the driver handles by calling *SQLGetInfo*().

22884 Each piece of code that implements XDBC (the Driver Manager or a driver) contains one or more
22885 environment handles. For example, the Driver Manager maintains a separate environment
22886 handle for each application that is connected to it.

22887 Within a single XDBC environment, multiple connection handles might point to a variety of
22888 drivers and data sources, the same driver and a variety of data sources, or even multiple
22889 connections to the same driver and data source.

22890 **State Transitions**

22891 State transitions are more complex for the Driver Manager and the drivers, as they must track
22892 the state of the environment, each connection, and each statement. Most of this work is done by
22893 the Driver Manager; the majority of the work that must be done by drivers occurs with
22894 statements with pending results.

22895 **Completing Transactions**

22896 Drivers for data sources that support transactions typically implement this function by
22897 executing a COMMIT or ROLLBACK statement. The Driver Manager does not call
22898 *SQLEndTran*() in when the connection is in auto-commit mode; it simply returns SQL_SUCCESS,
22899 even if the application attempts to roll back the transaction. Because drivers for data sources that
22900 do not support transactions are always in auto-commit mode, they can either implement
22901 *SQLEndTran*() to return SQL_SUCCESS without doing anything or not implement it at all.

22902 ## I.5 Implementation of the Diagnostic Area

22903 *SQLGetDiagRec*() and *SQLGetDiagField*() are implemented by the Driver Manager and each
22904 driver. The Driver Manager and each driver maintain diagnostic records for each environment,
22905 connection, statement, and descriptor handle and free those records only when another function
22906 is called with that handle or the handle is freed.

22907 Although both the Driver Manager and each driver must determine the first status record
22908 according the rankings in **Sequence of Status Records** on page 196, the Driver Manager
22909 determines the final sequence of records.

22910 *SQLGetDiagRec*() and *SQLGetDiagField*() do not post diagnostic records about themselves.

22911 **Error Handling Rules**

22912 The following rules govern error handling in *SQLGetDiagRec*() and *SQLGetDiagField*().

22913 All XDBC components:

22914 • Must not replace, alter, or mask errors or warnings received from another XDBC component.

22915 • May add an additional status record when they receive a diagnostic message from another
22916 XDBC component. The added record must add real information value to the original
22917 message.

22918 The XDBC component that directly interfaces a data source:

22919 • Must prefix its vendor identifier, its component identifier, and the data source's identifier to
22920 the diagnostic message it receives from the data source.

22921 • Must preserve the data source's native error code.

22922 • Must preserve the data source's diagnostic message.

22923 Any XDBC component that generates an error or warning independent of the data source:

22924 • Must supply the correct SQLSTATE for the error or warning.

22925 • Must generate the text of the diagnostic message.

22926 • Must prefix its vendor identifier and its component identifier to the diagnostic message.

22927 • Must return a native error code, if one is available and meaningful.

22928 The XDBC component that interfaces the Driver Manager:

22929 • Must initialize the output arguments of *SQLGetDiagRec*() and *SQLGetDiagField*().

22930 • Must format and return the diagnostic information as output arguments of *SQLGetDiagRec*()
22931 and *SQLGetDiagField*() when that function is called.

22932 One XDBC component other than the Driver Manager:

22933 • Must set the SQLSTATE based on the native error. For file-based drivers and data-source-
22934 based drivers that do not use a gateway, the driver must set the SQLSTATE. For data-source-
22935 based drivers that use a gateway, either the driver or a gateway that supports XDBC may set
22936 the SQLSTATE.

22937 ### I.5.1 Role of the Driver Manager

22938 The Driver Manager determines the final order in which to return status records. In particular, it
22939 determines which record has the highest rank and are to be returned first. It does not matter
22940 whether this record was generated by the driver or the Driver Manager. For more information,
22941 see **Sequence of Status Records** on page 196.

22942 The Driver Manager does as much error checking as it can. This saves every driver from
22943 checking for the same errors. For example, if a function argument accepts a discrete number of
22944 values, such as *Operation* in *SQLSetPos*(), the Driver Manager checks that the specified value is
22945 legal.

22946 The following sections describe the types of conditions checked by the Driver Manager. They are
22947 not intended to be exhaustive; for a complete list of the SQLSTATEs the Driver Manager returns,
22948 see the **DIAGNOSTICS** section of each function. Also see the state transition tables in
22949 Appendix B; errors shown in parentheses are detected by the Driver Manager.

22950 **Argument Values**

22951 The Driver Manager checks the following types of arguments. Unless otherwise noted, the
22952 Driver Manager returns SQL_ERROR for errors in argument values.

22953 • Environment, connection, and statement handles usually cannot be null pointers. The Driver
22954 Manager returns SQL_INVALID_HANDLE when it finds a null handle.

22955 • Required pointer arguments, such as OutputHandlePtr in *SQLAllocHandle*() and
22956 CursorName in *SQLSetCursorName*(), cannot be null pointers.

22957 • Option flags that do not support driver-specific values must be a legal value. For example,
22958 *Operation* in *SQLSetPos*() must be SQL_POSITION, SQL_REFRESH, SQL_UPDATE,
22959 SQL_DELETE, or SQL_ADD.

22960 • Option flags must be supported in the version of XDBC supported by the driver. For
22961 example, *InfoType* in *SQLGetInfo*() cannot be SQL_ASYNC_MODE (introduced in ODBC 3.0)
22962 when calling an ODBC 2.0 driver.

22963 • Column and parameter numbers must be greater than 0 or greater than or equal to 0,
22964 depending on the function. The driver must check the upper limit of these argument values
22965 based on the current result set or SQL statement.

22966 • Length/indicator arguments and data buffer length arguments must contain appropriate
22967 values. For example, the argument that specifies the length of a table name in *SQLColumns*()
22968 (*NameLength3*) must be SQL_NTS or a value greater than 0; *BufferLength* in *SQLDescribeCol*()
22969 must be greater than or equal to 0. The driver might also need to check these arguments. For
22970 example, it might check that *NameLength3* is less than or equal to the maximum length of a
22971 table name in the data source.

22972 **State Transitions**

22973 The Driver Manager checks that the state of the environment, connection, or statement is
22974 appropriate for the function being called. For example, a connection must be in an allocated state
22975 when *SQLConnect*() is called and a statement must be in a prepared state when *SQLExecute*() is
22976 called. The Driver Manager returns SQL_ERROR for state transition errors.

22977 **General Errors**

22978 The Driver Manager checks for the following general error and always returns SQL_ERROR
22979 when it encounters it:

22980 - The function must be supported by the driver.

22981 **Driver Manager Errors and Warnings**

22982 The Driver Manager completely or partially implements a number of functions and therefore
22983 checks for all or some of the errors and warnings in those functions.

22984 - The Driver Manager implements *SQLDataSources*() and *SQLDrivers*() and checks for all errors
22985 and warnings in these functions.

22986 - The Driver Manager checks if a driver implements *SQLGetFunctions*(). If the driver does not
22987 implement *SQLGetFunctions*(), the Driver Manager implements and checks for all errors and
22988 warnings in it.

22989 - The Driver Manager partially implements *SQLAllocHandle*(), *SQLConnect*(),
22990 *SQLDriverConnect*(), *SQLBrowseConnect*(), *SQLFreeHandle*(), *SQLGetDiagRec*(), and
22991 *SQLGetDiagField*() and checks for some errors in these functions. It may return the same
22992 errors as the driver for some of these functions, as both perform similar operations. For
22993 example, the Driver Manager or driver may return SQLSTATE IM008 (Dialog failed) if they
22994 are unable to display a login dialog box for *SQLDriverConnect*().

22995 **I.5.2 Role of the Driver**

22996 The driver checks for all errors and warnings not checked by the Driver Manager. This includes
22997 errorsand warnings in data truncation, data conversion, syntax, and some state transitions. The
22998 driver might also check errorsand warnings partially checked by the Driver Manager. For
22999 example, although the Driver Manager checks if the value of Operation in *SQLSetPos*() is legal,
23000 the driver must check whether it is supported.

23001 The driver also maps *native errors*, or errors returned by the data source, to SQLSTATEs. For
23002 example, the driver might map a number of different native errors for illegal SQL syntax to
23003 SQLSTATE 42000 (Syntax error or access violation). The driver returns the native error number
23004 in the SQL_DIAG_NATIVE field of the status record. Driver documentation should show how
23005 errors and warnings are mapped from the data source to arguments in *SQLGetDiagRec*() and
23006 *SQLGetDiagField*().

I.6 Changes to the Reference Manual Pages

The information in this section enhances the information in Chapter 21.

I.6.1 Information on Specific XDBC Functions

SQLAllocHandle()

Limit on number of handles

Drivers may impose a limit on the number of environment, connection, statement, and/or descriptor handles that can be allocated at any one time.

Allocating an environment handle

The Driver Manager doesn't call *SQLAllocHandle*() in the driver at this time, as it doesn't know which driver to call. It delays calling *SQLAllocHandle*() in the driver until the application calls a function to connect to a data source.

Under a Driver Manager's environment handle, if there already exists a driver's environment handle, then *SQLAllocHandle*() with a *HandleType* of SQL_HANDLE_ENV is not called in that driver when a connection is made, only *SQLAllocHandle*() with a *HandleType* of SQL_HANDLE_DBC. If a driver's environment handle does not exist under the Driver Manager's environment handle, then both *SQLAllocHandle*() with a *HandleType* of SQL_HANDLE_ENV and *SQLAllocHandle*() with a *HandleType* of SQL_HANDLE_DBC are called in the driver when the first connection handle of the environment is connected to the driver.

Environment handle allocation errors

Environment allocation occurs both within the Driver Manager and within each driver. The error returned by *SQLAllocHandle*() with an *HandleType* of SQL_HANDLE_ENV depends on which level the error occurred in.

If the implementation cannot allocate memory for **OutputHandlePtr* when *SQLAllocHandle*() with a *HandleType* of SQL_HANDLE_ENV is called, or the application provides a null pointer for *OutputHandlePtr*, *SQLAllocHandle*() returns SQL_ERROR. The implementation sets **OutputHandlePtr* to SQL_NULL_HENV (unless the application provided a null pointer). There is no handle with which to associate additional diagnostic information.

The Driver Manager does not call the driver-level environment handle allocation function until the application calls *SQLConnect*(), *SQLBrowseConnect*(), or *SQLDriverConnect*(). If an error occurs in the driver-level *SQLAllocHandle*() function, then the Driver-Manager-level *SQLConnect*(), *SQLBrowseConnect*(), or *SQLDriverConnect*() function returns SQL_ERROR. The diagnostic data structure contains SQLSTATE IM004 (Driver's *SQLAllocHandle*() failed), followed by a driver-specific SQLSTATE value from the driver. For example, SQLSTATE HY001 (Memory allocation error) indicates that the Driver Manager's call to the driver-level *SQLAllocHandle*() returned SQL_ERROR. The error is returned on a connection handle.

Allocating a connection handle

The Driver Manager doesn't call *SQLAllocHandle*() in the driver at this time, as it doesn't 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 Section I.3 on page 620.

It's important to note that allocating a connection handle isn't the same as loading a driver. The driver isn't loaded until a connection function is called. Thus, after allocating a connection handle and before connecting, the only functions the application can call with the connection handle are *SQLSetConnectAttr*(), *SQLGetConnectAttr*(), or *SQLGetInfo*() with the

23051 SQL_XDBC_VER option. Calling other functions with the connection handle, such as
23052 *SQLEndTran*(), returns SQLSTATE 08003 (Connection not open). For complete details, see
23053 Appendix B.

23054 The Driver Manager processes the *SQLAllocHandle*() function and calls the driver's
23055 *SQLAllocHandle*() function when the application calls *SQLConnect*(), *SQLBrowseConnect*(), or
23056 *SQLDriverConnect*(). (For more information, see *SQLConnect*().)

23057 For additional information about the flow of function calls between the Driver Manager and a
23058 driver, see *SQLConnect*().

23059 **Allocating a statement handle**

23060 When the application calls *SQLAllocHandle*() to allocate a statement handle:

23061 • The Driver Manager allocates a structure in which to store information about the statement
23062 and calls *SQLAllocHandle*() in the driver with the SQL_HANDLE_STMT option.

23063 • The driver allocates its own structure in which to store information about the statement and
23064 returns the driver statement handle to the Driver Manager.

23065 • The Driver Manager returns the Driver Manager statement handle to the application in the
23066 application variable.

23067 **SQLBrowseConnect()**

23068 The Driver Manager loads the driver that was specified in or that corresponds to the data source
23069 name specified in the initial browse request connection string; for information on when this
23070 occurs, see the ''Comments'' section in *SQLConnect*().

23071 The initial browse request connection string may contain the **DRIVER** keyword. If the browse
23072 request connection string contains the **DSN** keyword, the Driver Manager locates a
23073 corresponding data source specification in the system information:

23074 • If the Driver Manager finds the corresponding data source specification, it loads the
23075 associated driver; the driver can retrieve information about the data source from the system
23076 information.

23077 • If the Driver Manager cannot find the corresponding data source specification, it locates the
23078 default data source specification and loads the associated driver; the driver can retrieve
23079 information about the default data source from the system information. 'DEFAULT' is passed
23080 to the driver for the DSN.

23081 • If the Driver Manager cannot find the corresponding data source specification and there is no
23082 default data source specification, it returns SQL_ERROR with SQLSTATEIM002 (Data source
23083 not found and no default driver specified).

23084 If the browse request connection string contains the **DRIVER** keyword, the Driver Manager
23085 loads the specified driver; it does not attempt to locate a data source in the system information.
23086 Because the **DRIVER** keyword does not use information from the system information, the driver
23087 must define enough keywords so that a driver can connect to a data source using only the
23088 information in the browse request connection strings.

23089 **SQLConnect()**

23090 The Driver Manager does not load a driver until the application calls a function (*SQLConnect*(),
23091 *SQLDriverConnect*(), or *SQLBrowseConnect*()) to connect to the driver. Until that point, the Driver
23092 Manager works with its own handles and manages connection information. When the
23093 application calls a connection function, the Driver Manager checks if a driver is currently
23094 connected to for the specified ConnectionHandle:

23095 • If a driver is not connected to, the Driver Manager loads the driver and calls
23096 *SQLAllocHandle*() with a *HandleType* of SQL_HANDLE_ENV, *SQLAllocHandle*() with a
23097 *HandleType* of SQL_HANDLE_DBC, *SQLSetConnectAttr*() (if the application specified any
23098 connection attributes), and the connection function in the driver. The Driver Manager returns
23099 SQLSTATEIM006 (Driver's *SQLSetConnectAttr*() failed) and SQL_SUCCESS_WITH_INFO for
23100 the connection function if the driver returned an error for *SQLSetConnectAttr*(). For more
23101 information, see Chapter 6.

23102 • If the specified driver is already connected to on *ConnectionHandle*, the Driver Manager only
23103 calls the connection function in the driver. In this case, the driver must ensure that all
23104 connection attributes for *ConnectionHandle* maintain their current settings.

23105 • If a different driver is loaded, the Driver Manager calls *SQLFreeHandle*() with a *HandleType* of
23106 SQL_HANDLE_DBC, and then, if no other driver is connected to in that environment, it calls
23107 *SQLFreeHandle*() with a *HandleType* of SQL_HANDLE_ENV in the connected driver and then
23108 disconnects that driver. It then performs the same operations as when a driver is not loaded.

23109 The driver then allocates handles and initializes itself.

23110 When the application calls *SQLDisconnect*(), the Driver Manager calls *SQLDisconnect*() in the
23111 driver. However, it does not disconnect the driver. This keeps the driver in memory for
23112 applications that repeatedly connect to and disconnect from a data source. When the application
23113 calls *SQLFreeHandle*() with a *HandleType* of SQL_HANDLE_DBC, the Driver Manager calls
23114 *SQLFreeHandle*() with a *HandleType* of SQL_HANDLE_DBC and then *SQLFreeHandle*() with a
23115 *HandleType* of SQL_HANDLE_ENV in the driver, and then disconnects the driver.

23116 **Driver Manager Guidelines**

23117 The contents of *ServerName affect how the Driver Manager and a driver work together to
23118 establish a connection to a data source.

23119 • If *ServerName contains a valid data source name, the Driver Manager locates the
23120 corresponding data source specification in the system information and connects to the
23121 associated driver. The Driver Manager passes each *SQLConnect*() argument to the driver.

23122 • If the data source name cannot be found or ServerName is a null pointer, the Driver Manager
23123 locates the default data source specification and connects to the associated driver. The Driver
23124 Manager passes to the driver the *UserName* and *Authentication* arguments unmodified, and
23125 'DEFAULT' for the *ServerName* argument.

23126 • If the ServerName argument is 'DEFAULT', the Driver Manager locates the default data
23127 source specification and connects to the associated driver. The Driver Manager passes each
23128 *SQLConnect*() argument to the driver.

23129 After being connected to by the Driver Manager, a driver can locate its corresponding data
23130 source specification in the system information and use driver-specific information from the
23131 specification to complete its set of required connection information.

SQLCopyDesc()

If the Driver Manager detects that *SourceDescHandle* and *TargetDescHandle* do not belong to the same connection or environment, it implements *SQLCopyDesc*() by performing a field-by-field copy of all XDBC-defined fields using *SQLGetDescField*() and *SQLSetDescField*(). Implementation-defined fields are not copied. The following additional SQLSTATE value is defined:

HY092 — Invalid attribute identifier
SourceDescHandle and *TargetDescHandle* pertain to different servers, and the target server does not support at least one XDBC-defined descriptor field that the source server does support. The error is raised during a call to *SQLCopyDesc*() when the Driver Manager calls *SQLSetDescField*() at the target driver and determines that **ValuePtr* is not valid for the *FieldIdentifier* argument on *TargetDescHandle*.

SQLDataSources()

This function is implemented solely in the Driver Manager. Therefore, it is supported for all drivers regardless of a particular driver's compliance level. The Driver Manager retrieves this information from the system information.

In addition to the values set out in the reference manual page, *Direction* can be SQL_FETCH_FIRST_USER to fetch the first user DSN, or SQL_FETCH_FIRST_SYSTEM to fetch the first system DSN.

When *Direction* is set to SQL_FETCH_FIRST, subsequent calls to *SQLDataSources*() with *Direction* set to SQL_FETCH_NEXT return both user and system DSNs. When *Direction* is set to SQL_FETCH_FIRST_USER, all subsequent calls to *SQLDataSources*() with *Direction* set to SQL_FETCH_NEXT return only user DSNs. When *Direction* is set to SQL_FETCH_FIRST_SYSTEM, all subsequent calls to *SQLDataSources*() with *Direction* set to SQL_FETCH_NEXT return only system DSNs.

SQLDriverConnect()

The *attribute* syntactic element of a connection string is redefined to permit use of the **DRIVER** keyword:

```
attribute ::= attribute-keyword=attribute-value | DRIVER=[{]attribute-value[}]
```

The following additional keyword is defined:

Keyword	Attribute value description
DRIVER	Description of the driver as returned by the *SQLDrivers*() function.

If the **DSN** and **DRIVER** keywords are included in the same connection string, the implementation uses whichever keyword appears first.

Because the **DRIVER** keyword does not use information from the system information, the driver must define enough keywords so that a driver can connect to a data source using only the information in the connection string. (For more information, see **Driver Manager Guidelines** on page 629.) The driver defines which keywords are required in order to connect to the data source.

The **FILEDSN** and **DRIVER** keywords are not mutually exclusive. If any keyword appears in a connection string with **FILEDSN**, then the attribute value of the keyword in the connection string is used rather than the attribute value of the same keyword in the file DSN.

23174 **Driver-Specific Connection Information**

23175 *SQLConnect*() assumes that a data source name, user ID, and password are sufficient to connect
23176 to a data source and that all other connection information can be stored on the system. This is
23177 often not the case. For example, a driver might need one user ID and password to log into a
23178 server and a different user ID and password to log into a data source. Because *SQLConnect*()
23179 accepts a single user ID and password, this means that the other user ID and password must be
23180 stored with the data source information on the system if *SQLConnect*() is to be used. This is a
23181 potential breach of security and should be avoided unless the password is encrypted.

23182 *SQLDriverConnect*() lets the driver define an arbitrary amount of connection information in the
23183 form of keyword-value pairs. For example, suppose a driver requires a data source name, a user
23184 ID and password for the server, and a user ID and password for the data source. A custom
23185 program that always uses the XYZ Corp data source might prompt the user for IDs and
23186 passwords and build the following set of keyword-value pairs, or *connection string*, to pass to
23187 *SQLDriverConnect*():

```
23188 DSN=XYZ Corp;UID=Gomez;PWD=Sesame;UIDDBMS=JGomez;PWDDBMS=Shazam;
```

23189 The **DSN** (Data Source Name) keyword names the data source, the **UID** and **PWD** keywords
23190 specify the user ID and password for the server, and the **UIDDBMS** and **PWDDBMS** keywords
23191 specify the user ID and password for the data source. Note that the final semicolon is optional.
23192 *SQLDriverConnect*() parses this string; uses the XYZ Corp data source name to retrieve
23193 additional connection information from the system, such as the server address; and logs in to the
23194 server and data source using the specified user IDs and passwords.

23195 A FILEDSN keyword can be used in a call to *SQLDriverConnect*() to specify that a file DSN be
23196 established as the data source. A SAVEFILE keyword can be used to specify the name of a file
23197 DSN in which the keyword attributes of a successful connection made by the call to
23198 *SQLDriverConnect*() will be saved. For more information on file DSNs, see *SQLDriverConnect*().

23199 **Driver Manager Guidelines**

23200 The Driver Manager constructs a connection string to pass to the driver in the
23201 InConnectionString argument of the driver's *SQLDriverConnect*() function. Note that the Driver
23202 Manager does not modify the InConnectionString argument passed to it by the application.

23203 If the connection string specified by the application contains the **DSN** keyword or does not
23204 contain either the **DSN** or **DRIVER** keywords, the action of the Driver Manager is based on the
23205 value of the DriverCompletion argument:

23206 • SQL_DRIVER_PROMPT: The Driver Manager displays the Data Sources dialog box. It
23207 constructs a connection string from the data source name returned by the dialog box and any
23208 other keywords passed to it by the application. If the data source name returned by the
23209 dialog box is empty, the Driver Manager specifies the keyword-value pair DSN=Default.

23210 • SQL_DRIVER_COMPLETE or SQL_DRIVER_COMPLETE_REQUIRED: If the connection
23211 string specified by the application includes the **DSN** keyword, the Driver Manager copies the
23212 connection string specified by the application. Otherwise, it takes the same actions as it does
23213 when DriverCompletion is SQL_DRIVER_PROMPT.

23214 • SQL_DRIVER_NOPROMPT: The Driver Manager copies the connection string specified by
23215 the application.

23216 If the connection string specified by the application contains the **DRIVER** keyword, the Driver
23217 Manager copies the connection string specified by the application.

23218 Using the connection string it has constructed, the Driver Manager determines which driver to
23219 use, loads that driver, and passes the connection string it has constructed to the driver; for more

23220 information about the interaction of the Driver Manager and the driver, see the ''Comments''
23221 section in *SQLConnect*(). If the connection string contains the **DSN** keyword or does not contain
23222 either the **DSN** or the **DRIVER** keyword, the Driver Manager determines which driver to use as
23223 follows:

23224 1. If the connection string contains the **DSN** keyword, the Driver Manager retrieves the
23225 driver associated with the data source from the system information.

23226 2. If the connection string does not contain the **DSN** keyword or the data source is not found,
23227 the Driver Manager retrieves the driver associated with the Default data source from the
23228 system information. The Driver Manager changes the value of the **DSN** keyword in the
23229 connection string to ''DEFAULT''.

23230 3. If the **DSN** keyword in the connection string is set to ''DEFAULT'', the Driver Manager
23231 retrieves the driver associated with the Default data source from the system information.

23232 **Driver Guidelines**

23233 The driver checks if the connection string passed to it by the Driver Manager contains the **DSN** •
23234 or **DRIVER** keyword. If the connection string contains the **DRIVER** keyword, the driver cannot
23235 retrieve information about the data source from the system information. If the connection string
23236 contains the **DSN** keyword or does not contain either the **DSN** or the **DRIVER** keyword, the
23237 driver can retrieve information about the data source from the system information as follows:

23238 1. If the connection string contains the **DSN** keyword, the driver retrieves the information for
23239 the specified data source.

23240 2. If the connection string does not contain the **DSN** keyword, the specified data source is not
23241 found, or the **DSN** keyword is set to ''DEFAULT'', the driver retrieves the information for
23242 the Default data source.

23243 The driver uses any information it retrieves from the system information to augment the
23244 information passed to it in the connection string. If the information in the system information
23245 duplicates information in the connection string, the driver uses the information in the connection
23246 string.

23247 Based on the value of DriverCompletion, the driver prompts the user for connection
23248 information, such as the user ID and password, and connects to the data source:

23249 • SQL_DRIVER_PROMPT: The driver displays a dialog box, using the values from the
23250 connection string and system information (if any) as initial values. When the user exits the
23251 dialog box, the driver connects to the data source. It also constructs a connection string from |
23252 the value of the **DSN** or **DRIVER** keyword in **InConnectionString* and the information |
23253 returned from the dialog box. It places this connection string in the **OutConnectionString* |
23254 buffer.

23255 • SQL_DRIVER_COMPLETE or SQL_DRIVER_COMPLETE_REQUIRED: If the connection
23256 string contains enough information, and that information is correct, the driver connects to the
23257 data source and copies **InConnectionString* to **OutConnectionString*. If any information is |
23258 missing or incorrect, the driver takes the same actions as it does when DriverCompletion is
23259 SQL_DRIVER_PROMPT, except that if DriverCompletion is
23260 SQL_DRIVER_COMPLETE_REQUIRED, the driver disables the controls for any information
23261 not required to connect to the data source.

23262 • SQL_DRIVER_NOPROMPT: If the connection string contains enough information, the driver
23263 connects to the data source and copies **InConnectionString* to **OutConnectionString*. |
23264 Otherwise, the driver returns SQL_ERROR for *SQLDriverConnect*().

23265 On successful connection to the data source, the driver also sets **StringLength2Ptr* to the length
23266 of **OutConnectionString*.

23267 If the user cancels a dialog box presented by the Driver Manager or the driver,
23268 *SQLDriverConnect*() returns SQL_NO_DATA.

23269 For information about how the Driver Manager and the driver interact during the connection
23270 process, see *SQLConnect*().

23271 **Connecting Directly to Drivers**

23272 As discussed in Section I.2 on page 617, some applications don't want to use a data source at all.
23273 Instead, they want to connect directly to a driver. *SQLDriverConnect*() provides a way for the
23274 application to connect directly to a driver without specifying a data source. Conceptually, a
23275 temporary data source is created at run time.

23276 To connect directly to a driver, the application specifies the **DRIVER** keyword in the connection
23277 string instead of the **DSN** keyword. The value of the **DRIVER** keyword is the description of the
23278 driver as returned by *SQLDrivers*(). For example, suppose a driver has the description Paradox
23279 Driver and requires the name of a directory containing the data files. To connect to this driver,
23280 the application might use either of the following connection strings:

```
23281 DRIVER={Paradox Driver};Directory=C:PARADOX;
23282 DRIVER={Paradox Driver};
```

23283 With the first string, the driver wouldn't need any additional information. With the second
23284 string, the driver would need to prompt for the name of the directory containing the data files.

23285 **SQLDrivers()**

23286 The accompanying reference manual page for *SQLDrivers*() is mandatory when the Driver
23287 Manager architecture is in use.

23288 **SQLEndTran()**

23289 This function is implemented in the Driver Manager and in each driver. To complete all
23290 transactions in an environment, the Driver Manager calls *SQLEndTran*() once for each driver
23291 with active work in the transaction, except that the Driver Manager never calls *SQLEndTran*() for
23292 a driver that is in auto-commit mode.

23293 If the Driver Manager receives SQL_ERROR on one or more connections, it returns SQL_ERROR
23294 to the application, and the diagnostic information is placed in the diagnostic data structure of the
23295 environment.

23296 **SQLFreeHandle()**

23297 An application should not use a handle after it has been freed; the Driver Manager does not
23298 check the validity of a handle in a function call.

23299 **SQLGetDiagField()**

23300 The driver does not have to implement the SQL_DIAG_RETURNCODE diagnostic field; it is
23301 always implemented by the Driver Manager.

23302 **SQLGetFunctions()**

23303 This function is implemented in the Driver Manager; it can also be implemented in drivers. If a
23304 driver implements *SQLGetFunctions*(), the Driver Manager calls the function in the driver.
23305 Otherwise, it executes the function itself.

23306 *SQLGetFunctions*() always reports that *SQLGetFunctions*(), *SQLDataSources*(), and *SQLDrivers*()
23307 are supported for all valid values of *ConnectionHandle* because these functions are implemented
23308 in the Driver Manager.

23309 **SQLGetEnvAttr()**

23310 There are no driver-specific environment attributes.

23311 **SQLGetInfo()**

23312 If *InfoType* is SQL_DRIVER_HDESC or SQL_DRIVER_HSTMT, *InfoValuePtr* is both input and
23313 output. (See SQL_DRIVER_HDESC or SQL_DRIVER_HSTMT below.)

23314 **SQLGetInfo() Specific Requests**

23315 SQL_DATA_SOURCE_READ_ONLY
23316 This characteristic pertains only to the data source itself; it is not a characteristic of the
23317 driver that enables access to the data source. A driver that is read/write may be used with a
23318 data source that is read-only. If a driver is read-only, all of its data sources must be read-
23319 only, and must return SQL_DATA_SOURCE_READ_ONLY.

23320 SQL_DRIVER_HDBC, SQL_DRIVER_HENV
23321 An SQLINTEGER value, the driver's environment handle or connection handle, determined |
23322 by *InfoType*. |

23323 These options are implemented by the Driver Manager alone.

23324 SQL_DRIVER_HDESC
23325 An SQLINTEGER value, the driver's descriptor handle determined by the Driver Manager's
23326 descriptor handle, which must be passed on input in *InfoValuePtr from the application.
23327 Note that in this case, InfoValuePtr is both an input and output argument. The input
23328 descriptor handle passed in *InfoValuePtr must have been either explicitly or implicitly
23329 allocated on the ConnectionHandle.

23330 This option is implemented by the Driver Manager alone. |

23331 SQL_DRIVER_HLIB
23332 An SQLINTEGER value, a handle that refers to the driver, generated by the software that
23333 loaded the driver. The handle is only valid for the connection handle specified in the call to
23334 *SQLGetInfo*().

23335 This option is implemented by the Driver Manager alone. |

23336 SQL_DRIVER_HSTMT
23337 An SQLINTEGER value, the driver's statement handle determined by the Driver Manager
23338 statement handle, which must be passed on input in *InfoValuePtr from the application.
23339 Note that in this case, InfoValuePtr is both an input and an output argument. The input
23340 statement handle passed in *InfoValuePtr must have been allocated on the argument
23341 ConnectionHandle.

23342 This option is implemented by the Driver Manager alone. |

23343 SQL_DRIVER_NAME
23344 A character string with the filename of the driver used to access the data source.

23345 SQL_DRIVER_ODBC_VER

23346 A character string with the version of ODBC that the driver supports. The version is of the
23347 form ##.##, where the first two digits are the major version and the next two digits are the
23348 minor version. SQL_SPEC_MAJOR and SQL_SPEC_MINOR define the major and minor
23349 version numbers. For the version of ODBC described in this manual, these are 3 and 0, and
23350 the driver should return ''03.00''.

23351 SQL_DRIVER_VER

23352 A character string with the version of the driver and, optionally a description of the driver.
23353 At a minimum, the version is of the form ##.##.####, where the first two digits are the major
23354 version, the next two digits are the minor version, and the last four digits are the release
23355 version.

23356 SQL_XDBC_VER

23357 This is implemented solely in the Driver Manager.

23358 **SQLPrepare()**

23359 The driver may modify the statement to use the form of SQL used by the data source, then
23360 submit it to the data source for preparation. For the driver, a statement handle is similar to a |
23361 statement identifier in SQL. If the data source supports statement identifiers, the driver can send
23362 a statement identifier and parameter values to the data source.

23363 **SQLSetConnectAttr()**

23364 If *SQLSetConnectAttr*() is called before the driver is loaded, the Driver Manager stores the |
23365 attributes in its connection structure and sets them in the driver as part of the connection |
23366 process. |

23367 The Driver Manager returns a SQLSTATE of HY024 only for connection and statement attributes
23368 that accept a discrete set of values, such as SQL_ATTR_ACCESS_MODE or
23369 SQL_ATTR_ASYNC_ENABLE. For all other connection and statement attributes, the driver
23370 must verify the value specified in **ValuePtr*. |

23371 **SQLSetDescField()**

23372 (The Driver Manager returns a SQLSTATE of HY009 (Invalid use of null pointer) only for
23373 descriptor fields that accept a discrete set of values. For descriptor fields for which the *ValuePtr*
23374 argument is a pointer, the driver must verify the value specified in **ValuePtr*.)

23375 **SQLSetDescRec()**

23376 The Driver Manager returns a SQLSTATE of HY009 if *DataPtr* points to an invalid value based
23377 on *Type*, *SubType*, *Length*, *Precision*, or *Scale*. value. For all other descriptor fields, the driver must
23378 verify the value of *DataPtr*.

23379 **SQLSetEnvAttr()**

23380 There are no driver-specific environment attributes.

23381 **SQLSetStmtAttr()**

23382 (The Driver Manager returns SQLSTATE HY090 (Invalid string or buffer length) only for
23383 connection and statement attributes that accept a discrete set of values, such as
23384 SQL_ATTR_ACCESS_MODE or SQL_ ATTR_ASYNC_ENABLE. For all other connection and
23385 statement attributes, the driver must verify the value specified in **ValuePtr*.)

23386 A driver should not emulate the behavior of the SQL_ATTR_MAX_ROWS statement attribute
23387 for *SQLFetch*() or *SQLFetchScroll*() (if result set size limitations cannot be implemented at the
23388 data source) if it cannot guarantee that SQL_ATTR_MAX_ROWS is implemented properly. |

23389 I.6.2 SQLSTATEs of Specific XDBC Functions |

23390 This section explains which diagnostics are the responsibility of the Driver Manager and which |
23391 are the responsibility of the driver. For precise definitions of the cases where diagnostics are |
23392 raised, refer to the reference manual page of each XDBC function (**DIAGNOSTICS**). A cross- |
23393 reference of diagnostics, sorted first by SQLSTATE value and then by XDBC function, appears in |
23394 Appendix A. |

23395 *Notes to Reviewers* |

23396 *This section with side shading will not appear in the final copy. - Ed.* |

23397 I have tried to assemble a description based on the specific cases in the reference manual pages. |
23398 I suspect that this exercise will help disclose errors both here and in the source document. |

23399 In a Driver Manager implementation, the following errors are generated by the Driver Manager: |

23400 01000 — General warning |
23401 This warning is only reported by the Driver Manager in cases in which the entire function is |
23402 implemented exclusively by the Driver Manager: *SQLDrivers*(). In other cases, the driver |
23403 generates the warning. |

23404 01004 — String data, right truncation |
23405 The Driver Manager reports this diagnostic in *SQLDataSources*() and *SQLDrivers*(). The |
23406 driver reports it in all other cases. |

23407 07006 — Restricted data type attribute violation |
23408 The Driver Manager reports a column number of zero when use of bookmarks is disabled in |
23409 *SQLBindCol*(), and a record number of zero when use of bookmarks is disabled in |
23410 *SQLSetDescField*() and *SQLSetDescRec*(). The driver reports all other cases of this error. |

23411 07008 — Invalid descriptor count |
23412 The Driver Manager detects all cases of this error, reported based on the value of |
23413 *FieldIdentifier* in *SQLDescField*(). |

23414 07009 — Invalid descriptor index |
23415 The Driver Manager reports invalid values of the descriptor index in cases of negative |
23416 column number, column number of zero when use of bookmarks is disabled, and all cases |
23417 in *SQLGetData*() where the column number reflects an attempt to obtain columns in an |
23418 incorrect sequence (considering bound columns or the column returned in the previous call |
23419 to *SQLGetData*()). |

23420 The Driver Manager reports 07009 based on a failure of the consistency check in |
23421 *SQLGetData*(). |

23422 The driver reports invalid values of the descriptor index when it exceeds the number of |
23423 columns in the result set or the number of parameters in the associated SQL statement. |

23424 The driver reports 07009 in *DescribeParam*() when the associated SQL statement was not a
23425 DML statement or was part of a SELECT list.

23426 Notes to Reviewers

23427 *This section with side shading will not appear in the final copy. - Ed.*

23428 07009 was not marked as a Driver Manager error in *SQLGetDescField*(), *SQLGetDescRec*(),
23429 *SQLSetDescField*(), and *SQLSetDescRec*(). This doesn't seem consistent. Maybe another
23430 paragraph is required here.

23431 08002 — Connection name in use
23432 The Driver Manager detects all attempts to reuse a connection handle on which a
23433 connection is already open.

23434 08003 — Connection does not exist
23435 Errors regarding the status of a connection are discernible from status information
23436 associated with the connection handle and are detected by the Driver Manager.

23437 24000 — Invalid cursor state
23438 In *SQLGetData*(), the Driver Manager reports the error if *StatementHandle* was in an executed
23439 state but no result set was associated with it.

23440 For other cases of this error, if *SQLFetch*() or *SQLFetchScroll*() had been called, this error is
23441 raised by the Driver Manager; otherwise, it is raised by the driver. Cases where the error is
23442 caused by the cursor position with respect to the result set are raised by the driver.

23443 HY000 — General error
23444 Either the Driver Manager or the driver can raise this error (except in cases where the driver
23445 is not involved: *SQLDataSources*() and *SQLDrivers*().

23446 HY001 — Memory allocation error
23447 The Driver Manager raises HY001 if it fails to allocate required memory. The driver can also
23448 raise HY001.

23449 HY003 — Invalid application buffer type
23450 This error is reported by the Driver Manager in *SQLGetData*(), and by the driver in
23451 *SQLBindCol*() and *SQLBindParameter*()

23452 HY007 — Associated statement is not prepared
23453 The Driver Manager reports this error in *SQLDescribeCol*() if there were no preceding calls
23454 that prepared the statement. Other cases of this error, occurring because the handle was
23455 associated with an IRD, are reported by the driver.

23456 HY009 — Invalid use of null pointer
23457 The Driver Manager detects inappropriate use of null pointers, except in cases where an
23458 argument can be either a pointer or a scalar and the Driver Manager cannot know whether
23459 the argument of a specific call is a pointer. These include calls to *SetConnectAttr*(),
23460 *SetDescField*(), *SetEnvAttr*(), and *SetStmtAttr*() when the attribute or descriptor field is a
23461 driver-defined to be a string.

23462 HY010 — Function sequence error
23463 The Driver Manager maintains state information associated with each handle and detects all
23464 state transition errors except the following:

23465 — *SQLExecute*() reports HY010 if *StatementHandle* was not prepared. The Driver Manager
23466 does not track all cases in which the driver produces a result set. However, the Driver
23467 Manager detects that *SQLExecute*() was called out of sequence and reports HY010 if
23468 either of the following is true: (1) *StatementHandle* is not in an executed state, or (2) a
23469 cursor was open on *StatementHandle* and *SQLFetch*() or *SQLFetchScroll*() had been called.

23470 Other cases of HY010 are reported by the driver.

23471 — *SQLGetData*() reports HY010 if a cursor was open on *StatementHandle* and *SQLFetch*() or
23472 *SQLFetchScroll*() had been called, but the cursor was positioned before the start of the
23473 result set or after the end of the result set. Since the Driver Manager does not track the
23474 cursor position in the result set, the driver issues this report.

23475 HY012 — Invalid transaction operation code
23476 The Driver Manager detects all cases of this error, reported based on the value of
23477 *CompletionType* in *SQLEndTran*().

23478 HY017 — Invalid use of an automatically allocated descriptor handle.
23479 The Driver Manager detects all attempts to modify or free an automatically-allocated
23480 descriptor, such as an implementation descriptor.

23481 HY024 — Invalid attribute value
23482 The Driver Manager raises HY024 in *SQLDriverConnect*() when *WindowHandle* is a null
23483 pointer and a non-null pointer is needed. Other cases of HY024, which involve the values
23484 specified for attributes, are reported by the driver.

23485 HY090 — Invalid string or buffer length
23486 This diagnostic relates to the value of a length argument.

23487 The driver detects all cases of HY090 in *SQLBulkOperations*() and *SQLSetPos*(), and all cases
23488 of HY090 based on the value of a parameter set by *SQLBindParameter*().

23489 The Driver Manager detects all other cases when the value of the argument was negative
23490 (except when it was legally the negative constant SQL_NTS). The Driver Manager also
23491 detects excessive values of *NameLength1* in *SQLConnect*().

23492 The driver detects other cases where the argument exceeds the maximum value for the
23493 corresponding string. The driver detects cases of invalid length for driver-specific
23494 environment attributes, connection attributes, and statement attributes.

23495 ## *Notes to Reviewers*

23496 *This section with side shading will not appear in the final copy. - Ed.*

23497 Is the above really true where it mentions *SQLBulkOperations*() and *SQLSetPos*()? The errors
23498 documented there seem capable of being detected by the DM; perhaps these new manual
23499 pages were not reviewed for this.

23500 HY091 — Invalid descriptor field identifier
23501 The Driver Manager detects inappropriate values of *FieldIdentifier* in *SQLColAttribute*(). The
23502 driver reports this error in all other cases.

23503 HY092 — Invalid attribute identifier
23504 The Driver Manager reports HY092 in calls to *SQLAllocHandle*(), *SQLDriverConnect*(),
23505 *SQLEndTran*(), and *SQLFreeHandle*(), when it indicates an invalid value of an argument.

23506 In *SQLBulkOperations*() and *SQLSetPos*(), the Driver Manager reports HY092 except that the
23507 driver reports HY092 when *Operation* is not consistent with the SQL_CONCURRENCY
23508 statement attribute.

23509 The driver reports HY092 in *SQLGetConnectAttr*() when the value of the argument is not
23510 appropriate as a connection attribute, environment attribute, or statement attribute.

23511 ***Notes to Reviewers***
23512 *This section with side shading will not appear in the final copy. - Ed.*

23513 The ODBC manual documents this differently for different attributes.

23514 The Driver Manager reports HY092 in *SQLCopyDesc*(), and this case is further described in
23515 **SQLCopyDesc()** on page 628. *SQLCopyDesc*()

23516 HY095 — Function type out of range
23517 The Driver Manager detects all cases of this error, reported based on the value of *FunctionId*
23518 in *SQLGetFunctions*().

23519 HY097 — Column type out of range
23520 The Driver Manager detects all cases of this error, reported based on the value of
23521 *IdentifierType* in *SQLSpecialColumns*().

23522 HY098 — Scope type out of range
23523 The Driver Manager detects all cases of this error, reported based on the value of *Scope* in
23524 *SQLSpecialColumns*().

23525 HY099 — Nullable type out of range
23526 The Driver Manager detects all cases of this error, reported based on the value of *Nullable* in
23527 *SQLSpecialColumns*().

23528 HY100 — Uniqueness option type out of range
23529 The Driver Manager detects all cases of this error, reported based on the value of *Unique* in
23530 *SQLStatistics*().

23531 HY101 — Accuracy option type out of range
23532 The Driver Manager detects all cases of this error, reported based on the value of *Reserved* in
23533 *SQLStatistics*().

23534 HY103 — Invalid retrieval code
23535 The Driver Manager detects all cases of this error, reported based on the value of *Direction* in
23536 *SQLDataSources*() and *SQLDrivers*().

23537 HY105 — Invalid parameter type
23538 The Driver Manager reports this error when it is based on the value of *InputOutputType* in
23539 *SQLBindParameter*() or of the SQL_DESC_PARAMETER_TYPE field in *SQLSetDescField*().
23540 The driver reports this error if it occurs in *SQLExecDirect*() or *SQLExecute*().

23541 HY106 — Fetch type out of range
23542 The Driver Manager detects all cases of invalid *FetchOrientation* in *SQLFetchScroll*().

23543 HY109 — Invalid cursor position
23544 The Driver Manager reports this error in *SQLSetPos*() when *RowNumber* was 0 *Operation* was
23545 SQL_POSITION, and *SQLSetPos*() was called after *SQLBulkOperations*() was called, and
23546 before *SQLFetchScroll*() or *SQLFetch*() was called. The driver reports all other cases of this
23547 error.

23548 HY110 — Invalid value of DriverCompletion
23549 The Driver Manager detects all cases of this error, reported based on the value of
23550 *DriverCompletion* in *SQLDriverConnect*().

23551 IM001 — Function not supported
23552 The Driver Manager raises this SQLSTATE for functions that are not supported on the
23553 specified server.

23554 For *SQLAllocHandle*() and *SQLFreeHandle*(), this typically means that *HandleType* is
23555 SQL_HANDLE_STMT and the driver was not an XDBC driver, or *HandleType* was
23556 SQL_HANDLE_DESC and the driver does not support descriptor handles.

23557 In a Driver Manager implementation, additional SQLSTATE values are defined that are not
23558 mentioned in the reference manual pages:

23559 IM002 — Data source not found and no default driver specified
23560 This diagnostic specifically deals with the lack of driver information in the system
23561 information. The diagnostic is documented in the function reference section based on the
23562 value of:

23563 — *InConnectionString* of a call to *SQLBrowseConnect*()

23564 — *ServerName* of a call to *SQLConnect*()

23565 — *InConnectionString* of a call to *SQLDriverConnect*().
23566 In all situations except a call to *SQLBrowseConnect*(), IM002 also applies when information
23567 on the default data source and driver could not be found in the system information.

23568 IM003 — Specified driver could not be loaded
23569 The driver listed in the data source specification in the system information, (or, for
23570 *SQLBrowseConnect*() or *SQLDriverConnect*(), the one specified by the DRIVER keyword) was
23571 not found or could not be loaded for some other reason.

23572 IM004 — Driver's SQLAllocHandle on SQL_HANDLE_ENV failed
23573 During an attempt to connect, the Driver Manager called the driver's *SQLAllocHandle*()
23574 function with a *HandleType* of SQL_HANDLE_ENV and the driver returned an error.

23575 IM005 — Driver's SQLAllocHandle on SQL_HANDLE_DBC failed
23576 During an attempt to connect, the Driver Manager called the driver's *SQLAllocHandle*()
23577 function with a *HandleType* of SQL_HANDLE_DBC and the driver returned an error.

23578 IM006 — Driver's SQLSetConnectAttr failed
23579 During an attempt to connect, the Driver Manager called the driver's *SQLSetConnectAttr*()
23580 function and the driver returned an error. (The function returns
23581 SQL_SUCCESS_WITH_INFO).

23582 IM007 — No data source or driver specified; dialog prohibited
23583 (.Fn SQLDriverConnect) No data source name or driver was specified in the connection
23584 string and DriverCompletion was SQL_DRIVER_NOPROMPT.

23585 IM008 — Dialog failed
23586 (.Fn SQLDriverConnect) The driver attempted to display its login dialog box and failed.

23587 IM010 — Data source name too long
23588 The attribute value for the DSN keyword was longer than SQL_MAX_DSN_LENGTH
23589 characters.

23590 IM011 — Driver name too long
23591 The attribute value for the DRIVER keyword was longer than 255 characters.

23592 IM012 — DRIVER keyword syntax error
23593 The keyword-value pair for the DRIVER keyword contained a syntax error.

Glossary

This Glossary is intended to assist understanding and is not a substantive part of this specification.

Glossary

Access plan
A plan generated by the database engine to execute an SQL statement. Equivalent to executable code compiled from a third-generation language such as C.

Aggregate function
A function that generates a single value from a group of values, often used with GROUP BY and HAVING clauses. Aggregate functions include AVG, COUNT, MAX, MIN, and SUM. Also known as set functions. *See also* scalar function.

ANSI
American National Standards Institute, the national standards organization of the United States. The international (ISO) standards on which X/Open Data Management specifications are based have corresponding ANSI standards.

APD
Application parameter descriptor

API
Application Programming Interface. A set of routines that an application uses to request services from the implementation. XDBC is an API for database access.

API compliance
See *Compliance level*. (The term API compliance may imply that compliance is defined as support for a defined set of functions, which for XDBC is an over-simplification.)

Application
An executable program written by or for the end-user; for purposes of this specification, it is a program that calls XDBC functions.

Application descriptor
A descriptor, either an application parameter descriptor or an application row descriptor. The descriptor contains the application's version of data. For example, specifications of data types in an application descriptor pertain to the C data types of the application's buffers.

Application parameter descriptor (APD)
A descriptor that contains the application's version of a set of dynamic parameters.

Application row descriptor (ARD)
A descriptor that contains the application's version of a row of a table.

ARD
Application row descriptor.

Auto-commit mode
A transaction commit mode in which each database operation is a separate transaction. It takes effect immediately after it is executed. In auto-commit mode there is no need for the application to delimit transactions.

Automatically-allocated descriptor
One of the four descriptors that the XDBC implementation allocates and associates with a statement handle when the application allocates the handle.

Binding

As a verb, the act of associating a column in a result set or a parameter in an SQL statement with an application variable. As a noun, the association.

Block cursor

A cursor capable of fetching more than one row of data at a time. See Section 11.1 on page 140.

Buffer

A piece of application memory used to pass data between the application and the implementation. Buffers often come in pairs: a data buffer and a data length buffer.

Byte

This specification assumes a byte contains 8 bits and uses the term *octet* to describe this unit of storage.

C data type

The data type of a variable in a C program, in this case the application.

Catalog

The set of system tables in a database that describe the information in the database, as opposed to the information itself. Also known as a schema or data dictionary.

Catalog function

An XDBC function used to retrieve information from the database's catalog.

CLI

Stands for Call-Level Interface for SQL database access. The term is equivalent to the X/Open-specific term XDBC. The term CLI was used in the March 1995 predecessor to this specification, and is used in the ISO CLI International Standard. A call-level interface is simply an application programming interface (API).

Client/server

A database access strategy in which one or more clients gain access to data through a server. The clients generally implement the user interface while the server controls database access.

Column

The container for a single item of information in a row. Also known as field.

Commit

To make the changes in a transaction permanent.

Complete

To end a transaction, either by commiting the work or by rolling it back.

Compliance level

A value that indicates which of three nested sets of XDBC functions a data source completely supports. There are three XDBC compliance levels: Core, Level 1, and Level 2, defined in Section 1.7 on page 13.

Concurrency

The ability of more than one transaction to gain access to the same data at the same time.

Conformance

This specification uses the term *Compliance* as a synonym for *Conformance*. However, the word CONFORMANCE exists in some manifest constants for conformance with standards.

Connection

A particular instance of a data source plus whatever connection technology is required to gain access to the data source, such as drivers.

23682 **Connection browsing**
23683 Searching the network for data sources to connect to. Connection browsing might involve
23684 several steps. For example, the user might first browse the network for servers, then browse
23685 a particular server for a database.

23686 **Connection handle**
23687 A handle to a data structure that contains information about a connection.

23688 **Current row**
23689 The row currently pointed to by the cursor. Positioned operations act on the current row.

23690 **Cursor**
23691 A movable pointer to a row location within a table.

23692 **Data buffer**
23693 A buffer used to pass data. Often associated with a data buffer is a data length buffer.

23694 **Data dictionary**
23695 See catalog.

23696 **Data length buffer**
23697 A buffer used to pass the length of the value in a corresponding data buffer. The data length
23698 buffer is also used to store indicators, such as whether the data value is null-terminated.

23699 **Data source**
23700 The data the user wants to access and its associated operating system, DBMS, and network
23701 platform (if any).

23702 **Data type**
23703 The type of a piece of data. XDBC defines C and SQL data types. *See also* type indicator.

23704 **Data-at-execution column**
23705 A column for which data is sent after *SQLSetPos*() is called. So named because the data is
23706 sent at execution time rather than being placed in a row-set buffer. Long data is generally
23707 sent in parts at execution time.

23708 **Data-at-execution parameter**
23709 A parameter for which data is sent after *SQLExecute*() or *SQLExecDirect*() is called. So
23710 named because the data is sent when the SQL statement is executed rather than being
23711 placed in a parameter buffer. Long data is generally sent in parts at execution time.

23712 **Database**
23713 A discrete collection of data in a DBMS. Also a DBMS.

23714 **Database engine**
23715 The software in a DBMS that parses and executes SQL statements and accesses the physical
23716 data.

23717 **DBMS**
23718 Database Management System. A layer of software between the physical database and the
23719 user. The DBMS manages all access to the database.

23720 **DDL**
23721 Data Definition Language. Those statements in SQL that define, as opposed to manipulate,
23722 data. For example, CREATE TABLE, CREATE INDEX, GRANT, and REVOKE.

23723 **Descriptor**
23724 A data structure that holds information about either column data or dynamic parameters.
23725 See Chapter 13.

23726 **Desktop database**
23727 A DBMS designed to run on a personal computer. Generally, these DBMSs do not provide a
23728 standalone database engine and must be accessed through a file-based driver. These engines
23729 generally have reduced support for SQL and transactions.

23730 **Diagnostic**
23731 Diagnostics are information about the ability to which an XDBC function was able to
23732 perform the operation requested. Diagnostics comprise errors, warnings, and the success
23733 indication. See Chapter 15.

23734 **DML**
23735 Data Manipulation Language. Those statements in SQL that manipulate, as opposed to
23736 define, data. For example, INSERT, UPDATE, DELETE, and SELECT.

23737 **Driver**
23738 Connection technology, usually specific to a single DBMS or type of data source. See
23739 Appendix I.

23740 **Driver Manager**
23741 A routine library that manages access to drivers for the application. The Driver Manager
23742 loads and unloads drivers and passes calls to XDBC functions to the correct driver. See
23743 Appendix I.

23744 **Dynamic cursor**
23745 A scrollable cursor capable of detecting rows updated, deleted, or inserted in the result set.

23746 **Dynamic SQL**
23747 An environment in which SQL statements are created and compiled at run time. *See also*
23748 static SQL.

23749 **Embedded SQL**
23750 SQL statements that are included directly in a program written in another language, such as
23751 COBOL or C. XDBC does not use embedded SQL. *See also* static SQL and dynamic SQL.

23752 **Environment**
23753 A global context in which to gain access to data; associated with the environment is any
23754 information that is global in nature, such as a list of all connections in that environment.

23755 **Environment handle**
23756 A handle to a data structure that contains information about the environment.

23757 **Escape clause**
23758 Syntax defined by XDBC that an application can include in an SQL statement that it
23759 generates. The implementation converts the escape clause into correct syntax in the SQL
23760 dialect that the data source uses.

23761 **Execute**
23762 Having generated the text of an SQL statement, the application executes it by passing the
23763 text to the *SQLExecDirect*() or *SQLExecute*() function.

23764 **Explicitly-allocated descriptor**
23765 A descriptor that the application allocates by calling *SQLAllocHandle*(); and especially a
23766 descriptor that the application associates with a statement handle, taking the place of one of
23767 the automatically-allocated descriptors.

23768 **Fetch**
23769 To retrieve one or more rows from a result set.

23770 **Field**
23771 See column.

Foreign key
A column or columns in a table that match the primary key in another table.

Forward-only cursor
A cursor that can only move forward through the result set and generally fetching one row at a time. Most relational databases support only forward-only cursors.

Handle
A value that uniquely identifies something such as a file or data structure. Handles are meaningful only to the software that creates and uses them, but are passed by other software to identify things. XDBC defines handles for environments, connections, statements, and descriptors.

Implementation
The XDBC implementation refers to whatever software receives and processes an application's calls to the XDBC functions. A typical XDBC implementation is a run-time library. In a client/server architecture, the implementation primarily refers to the client-side software.

Implementation descriptor
A descriptor, either an implementation parameter descriptor or an implementation row descriptor. The descriptor contains the implementation's version of data. For example, specifications of data types in an implementation descriptor pertain to the SQL data types in which the data are stored in the database.

Implementation parameter descriptor (IPD)
A descriptor that contains the implementation's version of a set of dynamic parameters.

Implementation row descriptor (IRD)
A descriptor that contains the implementation's version of a row of a table.

Integrity Enhancement Facility
Features of the SQL language that let creators of tables specify constraints (interrelationships) between the columns of a table or between columns of separate tables. These constraints are checked at the end of any database operation. Constraints (table constraints and column constraints) are discussed in the CREATE TABLE statement in the X/Open **SQL** specification.

Interoperability
The ability of one application to use the same code when accessing data in different DBMSs.

IPD
Implementation parameter descriptor.

IRD
Implementation row descriptor.

ISO/IEC
International Standards Organization/International Electrotechnical Commission. The XDBC API is based on the ISO/IEC Call-Level Interface.

Isolation
See transaction isolation.

Join
An operation in a relational database that links the rows in two or more tables by matching values in specified columns.

Key
A column or columns whose values identify a row. *See also* primary key and foreign key.

23818 **Keyset**
23819 A set of keys used by a mixed or keyset-driven cursor to refetch rows.

23820 **Keyset-driven cursor**
23821 A scrollable cursor that detects updated and deleted rows by using a keyset.

23822 **Literal**
23823 A character representation of an actual data value in an SQL statement.

23824 **Locking**
23825 The process by which a DBMS restricts access to a row in a multiuser environment. The
23826 DBMS usually sets a bit on a row or the physical page containing a row that indicates the
23827 row or page is locked.

23828 **Long data**
23829 Any binary or character data over a certain length, such as 255 octets or characters. Typically
23830 much longer. Such data is generally sent to and retrieved from the data source in parts.

23831 **Manual-commit mode**
23832 A mode in which the application must explicitly complete each transaction by calling
23833 *SQLTransact*().

23834 **Metadata**
23835 Data that describes a parameter in an SQL statement or a column in a result set. For
23836 example, the data type, octet length, and precision of a parameter.

23837 **NULL value**
23838 Having no explicitly-assigned value. In particular, a NULL value is different from a zero or
23839 a blank.

23840 **Octet**
23841 Eight bits or one byte. Programmers have treated octets and characters interchangeably, but
23842 this assumes the use of character sets such as ASCII in which every character occupies a
23843 single octet. For international character sets and character sets being developed, this
23844 assumption is false.

23845 **Octet length**
23846 The length in octets of a buffer or the data it contains.

23847 **Optimistic concurrency**
23848 A strategy to increase concurrency in which rows are not locked. Instead, before they are
23849 updated or deleted, a cursor checks to see if they have been changed since they were last
23850 read. If so, the update or delete fails. *See also* pessimistic concurrency.

23851 **Option**
23852 One of a set of valid values for the argument of an XDBC function, by which the application
23853 selects the operation to be performed; especially, an option of *SQLGetInfo*(), by which the
23854 application specifies the piece of information on the implementation's capabilities and level
23855 of support to be retrieved.

23856 **Outer join**
23857 A join in which both matching and nonmatching rows are returned. The values of all
23858 columns from the unmatched table in nonmatching rows are set to NULL.

23859 **Owner**
23860 The owner of a table.

23861 **Parameter**
23862 A variable in an SQL statement, marked with a parameter marker or question mark (?).
23863 Parameters are bound to application variables and their values retrieved when the

23864 statement is executed.

23865 **Parameter descriptor**
23866 A descriptor that represents a set of dynamic parameters; either an application parameter
23867 descriptor or an implementation parameter descriptor.

23868 **Parameter operation array**
23869 An array containing values that an application can set to indicate that the corresponding
23870 parameter should be ignored in a *SQLExecDirect*() or *SQLExecute*() operation.

23871 **Parameter status array**
23872 An array containing the status of a parameter after a call to *SQLExecDirect*() or
23873 *SQLExecute*().

23874 **Pessimistic concurrency**
23875 A strategy for implementing serializability in which rows are locked so that other
23876 transactions cannot change them. *See also* optimistic concurrency.

23877 **Positioned operation**
23878 Any operation that acts on the current row. For example, positioned UPDATE and DELETE
23879 statements, *SQLGetData*(), and *SQLSetPos*().

23880 **Positioned UPDATE statement**
23881 An SQL statement used to update the values in the current row.

23882 **Positioned DELETE statement**
23883 An SQL statement used to delete the current row.

23884 **Prepare**
23885 To compile an SQL statement. An access plan is created by preparing an SQL statement.

23886 **Primary key**
23887 A column or columns that uniquely identifies a row in a table.

23888 **Procedure**
23889 A group of one or more precompiled SQL statements that are stored as a named object in a
23890 database.

23891 **Procedure column**
23892 An argument in a procedure call, the value returned by a procedure, or a column in a result
23893 set created by a procedure.

23894 **Qualifier**
23895 A database that contains one or more tables.

23896 **Query**
23897 An SQL statement. Sometimes used to mean a SELECT statement.

23898 **Radix**
23899 The base of a number system. Usually 2 or 10.

23900 **Record**
23901 See row.

23902 **Result set**
23903 The set of rows created by executing a SELECT statement.

23904 **Return code**
23905 The value returned by an XDBC function.

23906 **Roll back**
23907 To return the values changed by a transaction to their original state.

23908 **Row**
23909 A set of related columns that describe a specific entity. Also known as a record.

23910 **Row descriptor**
23911 A descriptor that represents a row of a table in the database; either an application row
23912 descriptor or an implementation row descriptor.

23913 **Row operation array**
23914 An array containing values that an application can set to indicate that the corresponding
23915 row should be ignored in a *SQLSetPos*() operation.

23916 **Row status array**
23917 An array containing the status of a row after a call to *SQLFetch*(), *SQLFetchScroll*(), or
23918 *SQLSetPos*().

23919 **Row-set**
23920 The set of rows returned in a single fetch.

23921 **Row-set buffers**
23922 The buffers bound to the columns of a result set and in which the data for an entire row-set
23923 is returned.

23924 **Scalar function**
23925 A function that generates a single value from a single value. For example, a function that
23926 changes the case of character data. See Appendix F.

23927 **Schema**
23928 See catalog.

23929 **Scrollable cursor**
23930 A cursor that can move forward or backward through the result set.

23931 **Serializability**
23932 Whether two transactions executing simultaneously produce a result that is the same as the
23933 serial (or sequential) execution of those transactions. Transactions must be serializable in
23934 order to maintain database integrity.

23935 **Set function**
23936 See aggregate function.

23937 **SQL**
23938 Structured Query Language. A language used by relational databases to query, update, and
23939 manage data. Standard SQL is defined in the ISO SQL standard. The X/Open definition of
23940 SQL appears in the X/Open **SQL** specification.

23941 **SQL data type**
23942 The data type of a column or parameter as it is stored in the data source.

23943 **SQLSTATE**
23944 A five-character value that indicates a particular error.

23945 **SQL statement**
23946 A complete phrase in SQL that begins with a keyword and completely describes an action to
23947 be taken. For example, SELECT * FROM Orders. SQL statements should not be confused
23948 with statements.

23949 **State**
23950 A well-defined condition of an item. For example, a connection has seven states, including
23951 unallocated, allocated, connected, and needing data. Certain operations can only be done
23952 when an item is in a particular state. For example, a connection can only be freed only when
23953 it is in an allocated state and not, for example, when it is in a connected state.

23954 **State transition**
23955 The movement of an item from one state to another. XDBC defines rigorous state transitions
23956 for environments, connections, and statements.

23957 **Statement**
23958 A container for all the information related to an SQL statement.

23959 **Statement handle**
23960 A handle to a data structure that contains information about a statement.

23961 **Static cursor**
23962 A scrollable cursor that cannot detect updates, deletes, or inserts in the result set. Usually
23963 implemented by making a copy of the result set.

23964 **Static SQL**
23965 A type of embedded SQL in which SQL statements are hard coded and compiled when the
23966 rest of the program is compiled. *See also* dynamic SQL.

23967 **Stored procedure**
23968 See procedure.

23969 **System information**
23970 An implementation-defined method of storing initialization and defaults. See Section 3.6 on
23971 page 30

23972 **Table**
23973 A collection of rows.

23974 **Transaction**
23975 An atomic unit of work. The work in a transaction must be completed as a whole; if any part
23976 of the transaction fails, the entire transaction fails.

23977 **Transaction isolation**
23978 The act of isolating one transaction from the effects of all other transactions.

23979 **Transaction isolation level**
23980 A measure of how well a transaction is isolated. There are five transaction isolation levels:
23981 Read Uncommitted, Read Committed, Repeatable Read, Serializable, and Versioning. See
23982 Section 14.2.2 on page 186.

23983 **Two-phase commit**
23984 A technique for completing transactions that ensures that associated work is either all
23985 committed or all rolled back. See Section 14.1.2 on page 183. XDBC does not require that
23986 implementations use two-phase commit.

23987 **Type indicator**
23988 An integer value passed to or returned from an XDBC function to indicate the data type of
23989 an application variable, a parameter, or a column. XDBC defines type indicators for both C
23990 and SQL data types.

23991 **View**
23992 An alternative way of looking at the data in one or more tables. A view is usually created as
23993 a subset of the columns from one or more tables. In XDBC, the term table includes views.

23994 **XDBC**
23995 This specification, X/Open Database Connectivity, which defines an API with a standard
23996 set of routines an application can use to gain access to data in a data source.

23997

Index

1995 issue, differences from 2
ABS function **603**
Access plan **639**
ACOS function **603**
active table 28
active transactions per connection 1
adaptation of embedded SQL 5
Aggregate function **639**
allocate handle
  in Core level 13
allocating descriptor
  timing 175
allocation
  deferral option 175
  explicit, of descriptor 175
  no need to declare size 7
ANSI **639**
APD **639**
API **1**, **639**
API compliance **639**
Application **639**
  shrink-wrapped 1
Application descriptor **639**
Application parameter descriptor (APD) **639**
Application row descriptor (ARD) **639**
ARD **639**
arrow
  in state tables 548
ASCII function **601**
ASIN function **603**
asynchrony **116**
  determining support for 116
  in Level 1 14
  new in this issue 3
  prerequisites 116
ATAN function **603**
ATAN2 function **603**
Auto-commit mode **639**
automatic data conversion 7
automatic sizing 7
Automatically-allocated descriptor **639**
batch
  multiple results from 156
  new in this issue 2
binary portability 1
Binding **640**
binding client to server 28
binding language 9
binding to an array of parameters
  new in this issue 2
block cursor **140**, **640**
bookmark
  new in this issue 2
bound column **174**
brace character 84
branding
  information on host language support 9
Buffer **640**
Byte **640**
C data type **640**
call
  escape sequence 86
calling procedure
  escape clause 88
cancelling asynchronous function 119
CAST function **609**
catalog **65**, **640**
  that does not have a name 28
Catalog function **640**
catalog-name 28
  syntax 28
CEILING function **603**
change
  temporary, to statement attribute 93
CHAR function **601**
choice of language 9
CLI **640**
CLI, differences from 2
client **28**
client/server **1**, **640**
Column **640**
column-wise binding
  bind offset ignored 113
commit **181**, **640**
compilation of embedded SQL 1
complete **181**, **640**
Compliance **11**
Compliance level **640**
compliance policy 9
compound statement
  new in this issue 2
CONCAT function **601**
concurrency **191**, **640**
concurrent processing 1

24046 limit on number of operations 123
24047 Conformance **640**
24048 conformance policy 9
24049 Connection **640**
24050 level of support for asynchrony 116
24051 number of active transactions 1
24052 state table 549
24053 Connection browsing **641**
24054 connection enhancement 3
24055 Connection handle **641**, 7
24056 connection statement
24057 executed by client 28
24058 connection string **61**
24059 connection-specific state 7
24060 context
24061 for stored routine 1
24062 control flow
24063 basic 52
24064 overview 51
24065 conversion of data 7
24066 convert data type 609
24067 CONVERT function **609**
24068 COS function **603**
24069 COT function **603**
24070 CURDATE function **605**
24071 current row **140**, **641**
24072 cursor **133**, **641**
24073 block **140**
24074 explicit declaration not required 7
24075 in Core level 13
24076 invalid state 547
24077 XDBC model 7
24078 cursor, scrollable
24079 in Level 1 14
24080 cursor-specification **7**
24081 returning multiple rows 7
24082 CURTIME function **605**
24083 CURTIMESTAMP function **605**
24084 data **28**
24085 global 1
24086 Data buffer **641**
24087 data dictionary **65**, **641**
24088 Data length buffer **641**
24089 data source **27**, **641**
24090 default **61**
24091 data structure
24092 automatic sizing 7
24093 referenced by handle 7
24094 Data type **641**
24095 conversion 609
24096 data-at-execute dialogue
24097 and asynchrony 122
24098 Data-at-execution column **641**

Data-at-execution parameter **641**
database **28**, **641**
concurrent operation 1
defined in X/Open SQL 1
Database engine **641**
DATABASEfunction **608**
date
literal escape clause 84
date value
constraints on 568
DAYNAME function **605**
DAYOFMONTH function **605**
DAYOFWEEK function **605**
DAYOFYEAR function **605**
DBMS **641**
DDL **641**
DE
in SQLBindParam() 220
in SQLGetInfo() 386, 395, 406
default
asynchrony mode 118
default data source **61**
deferred allocation 175
DEGREES function **603**
DELETE
footnote in state table 552
DELETE, positioned
new in this issue 2
delimiting transaction 10
Deprecated features **11**
DESCRIBE INPUT
optional feature 11
descriptor **170**, **641**
relation to handle 7
state transition 554
when allocated 175
descriptor handle
obtaining 175
obtaining as statement attribute 175
Desktop database **642**
development of XDBC 1
Diagnostic **642**
diagnostic record 195
diagnostics area
contents during asynchrony 122
relation to handle 7
diagnostics statement
executed by client 28
DIFFERENCE function " 1 601
differences from March 1995 issue 2
direct invocation
XDBC cursor based on 7
dirty read **186**

24099 discrepancy versus ISO
24100 resolving 8
24101 distributed transaction processing 1
24102 DML **642**
24103 Driver **642**
24104 Driver Manager **642**
24105 DTP 1
24106 Dynamic cursor **642**
24107 dynamic library 1
24108 Dynamic SQL **642**, 1
24109 XDBC cursor based on 7
24110 Embedded SQL **642**
24111 compliance 9
24112 XDBC not implemented on 5
24113 enable asynchrony 118
24114 entry in state tables
24115 specific overriding general 548
24116 Environment **642**
24117 state transition 548
24118 Environment handle **642**, 7
24119 equivalence to embedded SQL 1
24120 error
24121 inhibiting state transition 548
24122 escape character (LIKE)
24123 escape clause 86
24124 Escape clause **642**, 84
24125 date, time, timestamp 84
24126 interval 86
24127 LIKE escape character 86
24128 of X/Open SQL 84
24129 outer join 87
24130 procedure call 88
24131 escape sequence
24132 scalar function 86
24133 Execute **642**
24134 as opposed to OPEN 7
24135 execution model 7
24136 EXP function **603**
24137 expanded features in this issue 2
24138 explicit allocation/freeing 175
24139 explicit declaration of cursor 7
24140 Explicitly-allocated descriptor **642**
24141 extensibility
24142 of GetDescField() 175
24143 EXTRACT function **605**
24144 Fetch **642**
24145 fetch, multi-row
24146 new in this issue 3
24147 Field **642**
24148 file-based data source 27
24149 flexibility 1
24150 FLOOR function **603**
24151 flowchart of XDBC use 52

footnotes, significance of **12**
Foreign key **643**
forward-only cursor **133**, **643**
fourth-generation language 1
free handle
in Core level 13
freeing handle
descriptors freed when statement freed 175
explicit, of descriptor 175
function **97**
escape sequence 86
function call facility 1
function, table of 203
GET DIAGNOSTICS
executed by client 28
GetDescField()
overview 175
GetInfo()
determine asynchrony support 116
global data 1
global variable 7
Gregorian calendar 568
Handle **643**, 1
in Core level 13
statement 7
unallocated or null 547
header record 195
host language
required support for 9
HOUR function **605**
IFNULL function **608**
Implementation **643**
Implementation descriptor **643**
implementation method, asynchrony 117
Implementation parameter descriptor (IPD) **643**
Implementation row descriptor (IRD) **643**
Implementation-defined **12**
column attributes 175
initial call **118**
initial state
asynchrony 118
input parameter
notation in state tables 548
variable-length 44
INSERT function **601**
Integrity Enhancement Facility **643**
Interoperability **643**, 1
interpretation of state table 547
interpreting dynamic SQL 1
interval
literal escape clause 86
invalid cursor state 547
invocation technique 1

24152 IPD **643**
24153 IRD **643**
24154 ISO CLI
24155 relation to 8
24156 resolving discrepancy 8
24157 ISO/IEC **643**
24158 Isolation **643**, 181
24159 Join **643**
24160 Key **643**
24161 Keyset **644**
24162 Keyset-driven cursor **644**
24163 language binding 9
24164 language support 9
24165 layering XDBC on embedded SQL 5
24166 LCASE function **601**
24167 LENGTH function **601**
24168 length parameter 44
24169 level of support for asynchrony 116
24170 library, dynamic 1
24171 LIKE escape character
24172 escape clause 86
24173 limit
24174 concurrency 123
24175 Literal **644**
24176 escape clause for 84
24177 LOCATE function **601**
24178 LOCATE_2 function **601**
24179 Locking **644**, 181
24180 LOG function **603**
24181 LOG10 function **603**
24182 long data **105**, **644**
24183 LTRIM function **601**
24184 Manual-commit mode **644**
24185 margin notation
24186 OP 11
24187 market
24188 to measure asynchrony implementation 118
24189 metadata **28**, **127**, **644**
24190 method
24191 asynchrony implementation 117
24192 MINUTE function **606**
24193 MOD function **603**
24194 MONTH function **606**
24195 MONTHNAME function **606**
24196 MoreResults()
24197 new in this issue 4
24198 multi-row fetch
24199 new in this issue 3
24200 multiple association of descriptor 175
24201 multiple attempts to cancel 119
24202 multiple results **156**
24203 multiple rows
24204 cursor-specification 7

naming 28
new features in this issue 2
no effect 117
nonrepeatable read **186**
notation
in state tables 548
NOW function **606**
null handle 547
NULL value **644**
null-terminated string
input argument 44
object naming 28
obtaining descriptor handle 175
Octet **644**
Octet length **644**
OP 3, 3, 11, 14, 25
in SQLDrivers() 291
OP margin notation 11
OPEN
as opposed to EXECUTE 7
Optimistic concurrency **644**
Option **644**
Optional features **11**
order
of asynchronous operations 123
original function **118**
other X/Open documents, relation to 7
Outer join **644**
escape clause 87
output parameter
notation in state tables 548
overview of control flow 51
Owner **644**
parallel activity 117
Parameter **644**
notation in state tables 548
Parameter descriptor **645**
Parameter operation array **645**
Parameter status array **645**
performance
of asynchrony implementations 118
Pessimistic concurrency **645**
phantom 186
PI function **603**
policy, compliance 9
polling asynchronous function 116
portability 1
Positioned DELETE statement **645**
Positioned operation **645**
Positioned UPDATE statement **645**
positioned UPDATE/DELETE
new in this issue 2
POWER function **604**

24205 Prepare **645**
24206 preprocessor, XDBC independent of 1
24207 prerequisites for asynchrony 116
24208 previous issue, differences from 2
24209 Primary key **645**
24210 procedure **97**, **645**
24211 escape clause 88
24212 Procedure column **645**
24213 ProcedureColumns()
24214 new in this issue 4
24215 Procedures()
24216 new in this issue 4
24217 programming language 9
24218 proprietary source code 1
24219 PSM **1**
24220 qualification
24221 by catalog/schema name 28
24222 Qualifier **645**
24223 quality of implementation 118
24224 QUARTER function **606**
24225 Query **645**
24226 RADIANS function **604**
24227 Radix **645**
24228 RAND function **604**
24229 Record **645**
24230 diagnostic 195
24231 relation to standards 8
24232 relation to the X/Open **SQL** specification 7
24233 REPEAT function **602**
24234 REPLACE function **602**
24235 restricted handle
24236 effect on state tables 548
24237 restriction
24238 during asynchrony 122
24239 Result set **645**
24240 new in this issue 4
24241 results
24242 multiple **156**
24243 Return code **645**
24244 RIGHT function **602**
24245 roll back **181**, **645**
24246 ROUND function **604**
24247 Row **646**
24248 current **140**
24249 cursor-spec returning multiple 7
24250 describing with descriptor 171
24251 Row descriptor **646**
24252 Row operation array **646**
24253 Row status array **646**
24254 row-set **3**, **140**, **646**
24255 Row-set buffers **646**
24256 run-time library, dynamic 1
24257 Scalar function **646**

escape sequence 86
schema **28**, **646**
in Level 1 14
schema-name 28
syntax 28
scope of changes to global data 1
Scrollable cursor **646**
in Level 1 14
SECOND function **606**
sequence
of calls for asynchrony 119
overview of XDBC use 51
sequence error 547
sequencing error 547
Serializability **646**
server **28**
benefits of XDBC architecture 1
send asynchronous request to 117
SERVER_INFO
use to determine naming system 28
Set function **646**
SetDescField()
overview 175
shrink-wrapped application 1
SIGN function **604**
SIN function **604**
skeleton environment
effect on state tables 548
SOUNDEX function **602**
source code 1
SPACE function **602**
SQL **646**
compliance 9
SQL (embedded) 1
XDBC not implemented on 5
SQL data type **646**
SQL descriptor **170**
relation to handle 7
SQL statement **646**
SQL-based data source 27
SQL_ASYNC_MODE 116
SQL_ATTR_ASYNC_ENABLE **118**, 118
as connection attribute 458
SQL_ATTR_CONCURRENCY
in Level 2 15
SQL_ATTR_METADATA_ID
not defined as connection attribute 5
SQL_DEFAULT
as initial value of TYPE 175
SQL_DIAG_RETURNCODE
contents during asynchrony 122
SQL_INVALID_HANDLE
in state tables 547

24258 SQL_IS_NOT_POINTER **38**
24259 SQL_IS_POINTER **38**
24260 SQL_NTS 44, 44
24261 SQL_NULL_DATA 44
24262 SQL_STILL_EXECUTING **116**
24263 SQL_XDBC_KEYWORDS 390
24264 SQLAllocHandle() **208**
24265 in Core level 13
24266 SQLBindCol() **212**
24267 in Core level 13
24268 SQLBindParam() **220**
24269 dropped in this issue 5
24270 SQLBindParameter() **221**
24271 in Core level 13
24272 SQLBrowseConnect() **234**
24273 in Level 1 15
24274 new in this issue 3
24275 SQLBulkOperations() **239**
24276 in Level 1 15
24277 in Level 2 15
24278 SQLCancel() **247**
24279 asynchrony overview 116
24280 in Core level 14
24281 overview 107
24282 SQLCloseCursor() **250**
24283 in Core level 13
24284 SQLColAttribute() **252**
24285 in Core level 13
24286 SQLColumnPrivileges() **256**
24287 in Level 2 15
24288 SQLColumns() **261**
24289 in Core level 13
24290 SQLConnect() **269**
24291 default data source 61
24292 in Core level 14
24293 SQLCopyDesc() **272**
24294 in Core level 14
24295 SQLDataSources() **275**
24296 in Core level 14
24297 SQLDescribeCol() **277**
24298 in Core level 13
24299 SQLDescribeParam() **281**
24300 in Level 2 15
24301 SQLDisconnect() **284**
24302 in Core level 14
24303 SQLDriverConnect() **286**
24304 default data source 61
24305 in Core level 14
24306 new in this issue 3
24307 SQLDrivers() **291**
24308 in Core level 14
24309 SQLEndTran() **294**
24310 in Core level 14

SQLExecDirect() **297**
in Core level 14
SQLExecute() **302**
in Core level 14
SQLFetch() **307**
in Core level 14
SQLFetchScroll() **316**
in Core level 14
SQLForeignKeys() **326**
in Level 2 15
SQLFreeHandle() **333**
in Core level 13
SQLFreeStmt() **336**
in Core level 13
SQLGetConnectAttr() **338**
in Core level 14
SQLGetCursorName() **341**
in Core level 13
SQLGetData() **343**
in Core level 14
SQLGetDescField() **350**
in Core level 14
SQLGetDescRec() **354**
in Core level 14
SQLGetDiagField() **358**
in Core level 14
SQLGetDiagRec() **364**
in Core level 14
SQLGetEnvAttr() **367**
in Core level 14
SQLGetFunctions() **369**
in Core level 14
SQLGetInfo()
new items in this issue 4
SQLGetInfo() **372**
in Core level 14
SQLGetStmtAttr() **407**
in Core level 14
SQLGetTypeInfo() **410**
in Core level 13
SQLHDBC
state table 549
SQLHDESC **170**
state transition 554
SQLHENV
state transition 548
SQLHSTMT
state table 550
SQLMoreResults() **417**
in Level 1 15
SQLNativeSql() **420**
in Core level 14
new in this issue 4

24311 SQLNumParams() **423**
24312 in Core level 13
24313 new in this issue 4
24314 SQLNumResultCols() **425**
24315 in Core level 13
24316 SQLParamData() **427**
24317 in Core level 13
24318 SQLPrepare() **430**
24319 in Core level 14
24320 SQLPrimaryKeys() **434**
24321 in Level 1 15
24322 SQLProcedureColumns() **438**
24323 in Level 1 15
24324 SQLProcedures() **445**
24325 in Level 1 15
24326 SQLPutData() **449**
24327 in Core level 13
24328 SQLRowCount() **454**
24329 in Core level 13
24330 SQLSetConnectAttr() **456**
24331 in Core level 14
24332 SQLSetCursorName() **462**
24333 in Core level 13
24334 SQLSetDescField() **464**
24335 in Core level 14
24336 SQLSetDescRec() **484**
24337 in Core level 14
24338 SQLSetEnvAttr() **488**
24339 in Core level 14
24340 SQLSetPos() **491**
24341 in Level 1 15
24342 in Level 2 15
24343 SQLSetStmtAttr() **503**
24344 in Core level 14
24345 SQLSpecialColumns() **516**
24346 in Core level 14
24347 SQLSTATE **646**
24348 relation to handle 7
24349 SQLStatistics() **522**
24350 in Core level 13
24351 SQLTablePrivileges() **528**
24352 in Level 2 15
24353 SQLTables() **533**
24354 in Core level 13
24355 SQRT function **604**
24356 standard
24357 relation to 8
24358 State **646**
24359 state specific to connection 7
24360 state table **547**
24361 State transition **647**
24362 Statement **647**
24363 level of support for asynchrony 116

statement attribute
temporary change from execution 93
Statement handle **647**, 7
restrictions during asynchrony 122
state table 550
statement processing 54
statement text
compliance with X/Open SQL 9
statement, compound
new in this issue 2
Static cursor **647**
Static SQL **647**, 1
status record 195
stored modules **1**
Stored procedure **647**
stored routine
new support in this issue 4
string
zero-length 44
structure of database, unknown 1
subsequent call **118**
syntax 119
SUBSTRING function **602**
synchronous execution 116
System information **647**
Table **647**
table of functions 203
table, state **547**
TAN function **604**
temporary change to statement attribute 93
terminology
for asynchrony 118
three-part name
in Level 2 15
three-part naming 28
time
literal escape clause 84
time value
constraints on 568
time-slicing 117
timestamp
literal escape clause 84
TIMESTAMPADD function **606**
TIMESTAMPDIFF function **606**
transaction **181**, **647**
compliance policy 10
in Level 1 15
model 1
Transaction isolation **647**
Transaction isolation level **647**
transition, state **547**
TRIM function **602**
TRUNCATE function **604**

24364 two-part name
24365 in Level 1 14
24366 Two-phase commit **647**
24367 two-phase commit not mandated 184
24368 TYPE
24369 initial value 175
24370 Type indicator **647**
24371 UCASE function **602**
24372 unallocated handle 547
24373 Undefined **12**
24374 unknown database structure 1
24375 UPDATE
24376 footnote in state table 552
24377 UPDATE, positioned
24378 new in this issue 2
24379 usage overview 51
24380 USER function **608**
24381 user-defined-name
24382 as catalog-name and schema-name 28
24383 valid SQL statement **9**
24384 variable-length input parameter 44
24385 vendor escape clause (SQL) 84
24386 View **647**
24387 visibility of changes to global data 1
24388 WEEK function **607**
24389 WHERE CURRENT OF
24390 footnote in state table 552
24391 X/Open SQL **1**
24392 relation of XDBC to 7
24393 XDBC **1**, **647**
24394 development of 1
24395 YEAR function **607**
24396 zero-length string 44
24397 for null catalog name 28