Committee Draft ISO/IEC CD	
Date: **2006-02-18**	Reference number: ISO/JTC 1/SC 32**N1409**
Supersedes document SC 32N1201	

THIS DOCUMENT IS STILL UNDER STUDY AND SUBJECT TO CHANGE. IT SHOULD NOT BE USED FOR REFERENCE PURPOSES.

ISO/IEC JTC 1/SC 32 Data Management and Interchange Secretariat: USA (ANSI)	Circulated to P- and O-members, and to technical committees and organizations in liaison for voting (P-members only) by: **2006-05-18** Please return all votes and comments in electronic form directly to the SC 32 Secretariat by the due date indicated.

ISO/IEC CD 9075-03:200x(E)

Title: Information technology - Information technology - Database Languages - SQL - Part 3: Call-Level Interface (SQL/CLI)

Project: 1.32.03.06.03.00

Introductory note: The attached document is hereby submitted for a three-month letter ballot to the National Bodies of ISO/IEC JTC 1/SC 32. The ballot starts 2006-02-18.

Medium: E

No. of pages: 390

Address Reply to: SC 32 Secretary, ISO/IEC JTC 1/SC 32,
Farance Inc, Island Box 256, New York, NY 10044-0205, United States of America
Telephone: +1 212 486-4700; E-mail: SC32-Sec@JTC1SC32.org

ISO/IEC JTC 1/SC 32

Date: 2006-02-01

CD 9075-3:200x(E)

ISO/IEC JTC 1/SC 32/WG 3

The United States of America (ANSI)

Information technology — Database languages — SQL —

Part 3: Call-Level Interface (SQL/CLI)

Technologies de l'information — Langages de base de données — SQL —

Partie 3: Interface de Niveau d'Appel (SQL/CLI)

Document type: International Standard
Document subtype: Committee Draft (CD)
Document stage: (3) CD under Consideration
Document language: English

Copyright notice

This ISO document is a working draft or a committee draft and is copyright-protected by ISO. While the reproduction of working drafts or committee drafts in any form for use by participants in the ISO standards development process is permitted without prior permission from ISO, neither this document nor any extract from it may be reproduced, stored or transmitted in any form for any other purpose without prior written permission from ISO.

Requests for permission to reproduce for the purpose of selling it should be addressed as shown below or to ISO's member body in the country of the requester.

ANSI Customer Service Department
25 West 43rd Street, 4th Floor
New York, NY 10036
Tele: 1-212-642-4980
Fax: 1-212-302-1286
Email: storemanager@ansi.org
Web: www.ansi.org

Reproduction for sales purposes may be subject to royalty payments or a licensing agreement.

Violaters may be prosecuted.

Contents

Page

Tables

Table   Page

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75% of the national bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this International Standard may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

International Standard ISO/IEC 9075-3 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 32, *Data management and interchange*.

This fourth edition of this part of ISO/IEC 9075 cancels and replaces the third edition, ISO/IEC 9075-3:2003.

ISO/IEC 9075 consists of the following parts, under the general title *Information technology — Database languages — SQL*:

— Part 1: Framework (SQL/Framework)

— Part 2: Foundation (SQL/Foundation)

— Part 3: Call-Level Interface (SQL/CLI)

— Part 4: Persistent Stored Modules (SQL/PSM)

— Part 9: Management of External Data (SQL/MED)

— Part 10: Object Language Bindings (SQL/OLB)

— Part 11: Information and Definition Schema (SQL/Schemata)

— Part 13: SQL Routines and Types Using the Java™ Programming Language (SQL/JRT)

— Part 14: XML-Related Specifications (SQL/XML)

Introduction

The organization of this part of ISO/IEC 9075 is as follows:

1) Clause 1, “Scope”, specifies the scope of this part of ISO/IEC 9075.

2) Clause 2, “Normative references”, identifies additional standards that, through reference in this part of ISO/IEC 9075, constitute provisions of this part of ISO/IEC 9075.

3) Clause 3, “Definitions, notations, and conventions”, defines the notations and conventions used in this part of ISO/IEC 9075.

4) Clause 4, “Concepts”, presents concepts used in the definition of the Call-Level Interface.

5) Clause 5, “Call-Level Interface specifications”, defines facilities for using SQL through a Call-Level Interface.

6) Clause 6, “SQL/CLI routines”, defines each of the routines that comprise the Call-Level Interface.

7) Clause 7, “Definition Schema”, specifies extensions to the Definition Schema required for support of the Call-Level Interface.

8) Clause 8, “Conformance”, defines the criteria for conformance to this part of ISO/IEC 9075.

9) Annex A, “Typical header files”, is an informative Annex. It provides examples of typical definition files for application programs using the SQL Call-Level Interface.

10) Annex B, “Sample C programs”, is an informative Annex. It provides examples of using the SQL Call-Level Interface in the C programming language.

11) Annex C, “Implementation-defined elements”, is an informative Annex. It lists those features for which the body of this part of ISO/IEC 9075 states that the syntax, the meaning, the returned results, the effect on SQL-data and/or schemas, or any other behavior is partly or wholly implementation-defined.

12) Annex D, “Implementation-dependent elements”, is an informative Annex. It lists those features for which the body of this part of ISO/IEC 9075 states that the syntax, the meaning, the returned results, the effect on SQL-data and/or schemas, or any other behavior is partly or wholly implementation-dependent.

13) Annex E, “Incompatibilities with ISO/IEC 9075:2003”, is an informative Annex. It identifies incompatibilities with ISO/IEC 9075-3:2003.

14) Annex F, “SQL feature taxonomy”, is an informative Annex. It contains a taxonomy of features of the SQL language that are specified in this part of ISO/IEC 9075.

In the text of this part of ISO/IEC 9075, Clauses begin a new odd-numbered page, and in Clause 5, “Call-Level Interface specifications”, through Clause 8, “Conformance”, Subclauses begin a new page. Any resulting blank space is not significant.

Information technology — Database languages — SQL —

Part 3:
Call-Level Interface (SQL/CLI)

1 Scope

This part of ISO/IEC 9075 defines the structures and procedures that may be used to execute statements of the database language SQL from within an application written in a programming language in such a way that procedures used are independent of the SQL statements to be executed.

(Blank page)

2 Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

2.1 JTC1 standards

[ISO1539] ISO/IEC 1539-1:2004, *Information technology — Programming languages — Fortran — Part 1: Base language*.

[ISO1539-2] ISO/IEC 1539-2:2000, *Information technology — Programming languages — Fortran — Part 2: Varying length character strings*.

[ISO1989] ISO 1989:2002, *Information technology — Programming languages — COBOL*.

[ISO6160] ISO 6160:1979, *Programming languages — PL/I*. (Endorsement of ANSI X3.53-1976).

[ISO7185] ISO/IEC 7185:1990, *Information technology — Programming languages — Pascal*.

[ISO8652] ISO/IEC 8652:1995, *Information technology — Programming languages — Ada*.

ISO/IEC 8652:1995/Cor.1:2001.

[Framework] ISO/IEC 9075-1:200*n*, *Information technology — Database languages — SQL — Part 1: Framework (SQL/Framework)*.

[Foundation] ISO/IEC 9075-2:200*n*, *Information technology — Database languages — SQL — Part 2: Foundation (SQL/Foundation)*

[Schemata] ISO/IEC 9075-11:200*n*, *Information technology — Database languages — SQL — Part 11: Information and Definition Schemas (SQL/Schemata)*

[ISO9899] ISO/IEC 9899:1999, *Programming languages — C*.

ISO/IEC 9899:1999/Cor 1:2001, *Technical Corrigendum to ISO/IEC 9899:1999*.

ISO/IEC 9899:1999/Cor 2:2004, *Technical Corrigendum number 2 to ISO/IEC 9899:1999*.

[ISO10206] ISO/IEC 10206:1991, *Information technology — Programming languages — Extended Pascal*.

[ISO11756] ISO/IEC 11756:1999, *Information technology — Programming languages — M*.

(Blank page)

3 Definitions, notations, and conventions

This Clause modifies Clause 3, “Definitions, notations, and conventions”, in ISO/IEC 9075-2.

3.1 Definitions

This Subclause modifies Subclause 3.1, “Definitions”, in ISO/IEC 9075-2.

3.1.1 Definitions provided in Part 3

For the purposes of this document, the following definitions apply.

3.1.1.1 **data source:** A synonym for the SQL-server that is part of the current SQL-connection.

3.1.1.2 **handle:** A CLI object returned by an SQL/CLI implementation when a CLI resource is allocated and used by an SQL/CLI application to reference that CLI resource.

3.1.1.3 **inner table:** The second operand of a left outer join or the first operand of a right outer join.

3.1.1.4 **pseudo-column:** A column that is part of a table but is not part of the descriptor for that table. An example of such a pseudo-column is an implementation-defined row identifier.

3.1.1.5 **rowset:** One or more rows retrieved in a single invocation of the Fetch and FetchScroll routines.

3.1.1.6 **SQL/CLI application:** An application that invokes <CLI routine>s specified in this part of ISO/IEC 9075.

3.2 Conventions

This Subclause modifies Subclause 3.3, “Conventions”, in ISO/IEC 9075-2.

3.2.1 Specification of routine definitions

The routines in this document are specified in terms of:

— **Function**: A short statement of the purpose of the routine.

— **Definition**: The name of the routine and the name, mode, and data type of each of its parameters.

— **General Rules**: A specification of the run-time effect of the routine. Where more than one General Rule is used to specify the effect of a routine, the required effect is that which would be obtained by beginning with the first General Rule and applying the Rules in numeric sequence until a Rule is applied that specifies or implies a change in sequence or termination of the application of the Rules. Unless otherwise specified or implied by a specific Rule that is applied, application of General Rules terminates when the last in the sequence has been applied.

4 Concepts

This Clause modifies Clause 4, "Concepts", in ISO/IEC 9075-2.

4.1 Introduction to SQL/CLI

This Subclause is modified by Subclause 4.18, "Introduction to SQL/CLI", in ISO/IEC 9075-9.

The Call-Level Interface (SQL/CLI) is a binding style for executing SQL statements. This part of ISO/IEC 9075 provides specifications for routines that:

— Allocate and deallocate resources.

— Control connections to SQL-servers.

— Execute SQL statements using mechanisms similar to dynamic SQL.

— Obtain diagnostic information.

— Control transaction termination.

— Obtain information about the SQL/CLI implementation and the SQL-implementation.

A *handle* is a CLI object returned by an SQL/CLI implementation when a CLI resource is allocated; the handle is used by an SQL/CLI application to reference that CLI resource. The AllocHandle routine allocates the resources to manage an SQL-environment, an SQL-connection, a CLI descriptor area, or SQL-statement processing; when invoked, it returns an environment handle, a connection handle, a descriptor handle, or a statement handle, respectively. An SQL-connection is allocated in the context of an allocated SQL-environment. CLI descriptor areas and SQL-statements are allocated in the context of an allocated SQL-connection. The FreeHandle routine deallocates a specified resource. The AllocConnect, AllocEnv, and AllocStmt routines can be used to allocate the resources to manage an SQL-connection, an SQL-environment, and SQL-statement processing, respectively, instead of using the AllocHandle routine. The FreeConnect, FreeEnv, and FreeStmt routines can be used to deallocate the specific resource instead of using FreeHandle.

Each allocated SQL-environment has an attribute that determines whether output character strings are null terminated by the SQL/CLI implementation. The SQL/CLI application can set the value of this attribute by using the routine SetEnvAttr and can retrieve the current value of the attribute by using the routine GetEnvAttr.

The Connect routine establishes an SQL-connection, which becomes the *current SQL-connection* . The Disconnect routine terminates an established SQL-connection. Switching between established SQL-connections occurs automatically whenever the SQL/CLI application switches processing to a dormant SQL-connection, which then becomes the *current SQL-connection*.

The ExecDirect routine is used for a one-time execution of an SQL-statement. The Prepare routine is used to prepare an SQL-statement for subsequent execution using the Execute routine. In all three cases, the executed SQL-statement can contain dynamic parameters.

The interface for a description of dynamic parameters, dynamic parameter values, the result columns of a <dynamic select statement> or <dynamic single row select statement>, and the target specifications for the result columns is a CLI descriptor area. A CLI descriptor area for each type of interface is automatically allocated when an SQL-statement is allocated. The SQL/CLI application may allocate additional CLI descriptor areas and nominate them for use as the interface for the description of dynamic parameter values or the description of target specifications by using the routine SetStmtAttr. The SQL/CLI application can determine the handle value of the CLI descriptor area currently being used for a specific interface by using the routine GetStmtAttr. The GetDescField and GetDescRec routines enable information to be retrieved from a CLI descriptor area. The CopyDesc routine enables the contents of a CLI descriptor area to be copied to another CLI descriptor area.

When a <dynamic select statement> or <dynamic single row select statement> is prepared or executed immediately, a description of the result columns is automatically provided in the applicable CLI implementation descriptor area. In this case, the SQL/CLI application may additionally retrieve information by using the DescribeCol and/or the ColAttribute routine to obtain a description of a single result column and by using the NumResultCols routine to obtain a count of the number of result columns. The SQL/CLI application sets values in the CLI application descriptor area for the description of the corresponding target specifications either explicitly, by using the routines SetDescField and SetDescRec, or implicitly, by using the routine BindCol.

When an SQL-statement is prepared or executed immediately, a description of the dynamic parameters is automatically provided in the applicable CLI implementation descriptor area if this facility is supported by the current SQL-connection. An attribute associated with the allocated SQL-connection indicates whether this facility is supported. The value of the attribute may be retrieved using the routine GetConnectAttr. Regardless of whether automatic description is supported, all dynamic input and input/output parameters shall be defined in the application descriptor area before SQL-statement execution. This can be done either explicitly, by using the routines SetDescField and SetDescRec, or implicitly, by using the routine BindParameter. The value of a dynamic input or input/output parameter may be established before SQL-statement execution (immediate parameter value) or may be provided during SQL-statement execution (deferred parameter value). Its description in the CLI descriptor area determines which method is in use. The ParamData routine is used to cycle through and process deferred input and input/output parameter values. The PutData routine is used to provide the deferred values. The PutData routine also enables the values of character string input and input/output parameters to be provided piece by piece.

Before a <call statement> is prepared or executed immediately, the SQL/CLI application may choose whether or not to bind any dynamic output parameters in the CLI application descriptor area. This can be done either explicitly, by using the routines SetDescField and SetDescRec, or implicitly, by using the routine BindParameter. After execution of the statement, values of unbound output and input/output parameters can be individually retrieved using the GetParamData routine. The GetParamData routine also enables the retrieval of the values of character and binary string output and input/output parameters to be accomplished piece by piece.

When a <dynamic select statement> or <dynamic single row select statement> is executed, a cursor is implicitly declared and opened. The cursor name can be supplied by the SQL/CLI application by using the routine SetCursorName. If a cursor name is not supplied by the SQL/CLI application, an implementation-dependent cursor name is generated. The cursor name can be retrieved by using the GetCursorName routine.

The Fetch and FetchScroll routines are used to position an open cursor on a row and to retrieve the values of bound columns for that row. A bound column is one whose target specification in the specified CLI descriptor area defines a location for the target value. The Fetch routine always positions the open cursor on the next row, whereas the FetchScroll routine may be used to position the open cursor on any of its rows. At the time that the cursor is implicitly declared, the value of the CURSOR SCROLLABLE statement attribute shall be SCROLLABLE, allowing the use of FetchScroll with a FetchOrientation other than NEXT. The SQL/CLI application can set the value of this attribute by using the SetStmtAttr routine and can retrieve the current value of the attribute by using the GetStmtAttr routine. The Fetch and FetchScroll routines can also retrieve multiple rows in a single call; the set of rows thus retrieved is called a *rowset*. This is accomplished by setting the

ARRAY_SIZE field of the applicable application row descriptor to the desired number of rows. Note that the single row fetch is just a special case of multi-row fetch, where the rowset size is 1 (one).

Values for unbound columns can be individually retrieved by using the GetData routine. The GetData routine also enables the retrieval of the values of character and binary string columns to be accomplished piece by piece. The current row of a cursor can be deleted or updated by executing a <preparable dynamic delete statement: positioned> or a <preparable dynamic update statement: positioned>, respectively, for that cursor under a different allocated SQL-statement to the one under which the cursor was opened. The CloseCursor routine enables a cursor to be closed.

Result sets can be returned to the SQL/CLI application as a result of invoking the Execute or ExecDirect routine, supplying a statement handle whose current statement is a <call statement>. Such result sets are described and processed in the same way as outlined above for the result sets produced by the execution of a <dynamic select statement>. Multiple result sets may result from the execution of a single <call statement>. These result sets are returned as an ordered set of result sets that can be processed one at a time or in parallel. To process the result sets one at a time, once the processing of a given result set is complete, the MoreResults routine is used to determine whether there are any additional result sets and, if there are, to position the cursor before the first row in the next result set. To process the result sets in parallel, the NextResult routine is used to determine whether there are any additional result sets and, if there are, to position a cursor before the first row in the next result set.

Special routines, called *catalog routines* are available to return result sets from the Information Schema. These routines are:

— ColumnPrivileges: Returns a list of the privileges held on the columns whose names adhere to the requested pattern(s) within a single specified table. Most of this information can also be obtained by using the ExecDirect routine to issue an appropriate query on the COLUMN_PRIVILEGES view of the Information Schema.

— Columns: Returns the column names and attributes for all columns whose names adhere to the requested pattern(s). Most of this information can also be obtained by using the ExecDirect routine to issue an appropriate query on the COLUMNS view of the Information Schema.

— ForeignKeys: Returns either the primary key of a single specified table together with the foreign keys in all other tables that reference that primary key or the foreign keys of a single specified table together with all the primary and unique keys in all other tables that are referenced by those foreign keys. Most of this information can also be obtained by using the ExecDirect routine to issue an appropriate query on the TABLE_CONSTRAINTS view and the REFERENTIAL_CONSTRAINTS view of the Information Schema.

— PrimaryKeys: Returns a list of the columns that constitute the primary key of a single specified table. Most of this information can also be obtained by using the ExecDirect routine to issue an appropriate query on the TABLE_CONSTRAINTS view and the KEY_COLUMN_USAGE view of the Information Schema.

— SpecialColumns: Returns a list of the columns which can uniquely identify any row within a single specified table. Most of this information can also be obtained by using the ExecDirect routine to issue an appropriate query on the COLUMNS view of the Information Schema.

— Tables: Returns information about the tables whose names adhere to the requested pattern(s) and type(s). Most of this information can also be obtained by using the ExecDirect routine to issue an appropriate query on the TABLES view of the Information Schema.

— TablePrivileges: Returns a list of the privileges held on tables whose names adhere to the requested pattern(s). Most of this information can also be obtained by using the ExecDirect routine to issue an appropriate query on the TABLE_PRIVILEGES view of the Information Schema.

These special routines are only available for a small portion of the metadata that is available in the Information Schema. Other metadata (for example, that about SQL-invoked routines, triggers, and user-defined types) can be obtained by executing appropriate queries on the views of the Information Schema.

The GetPosition, GetLength, and GetSubString routines can each be used with its own independent statement handle to access a string value at the server that is represented by a Large Object locator in order to do any of the following:

— The GetPosition routine may be used to determine whether a given substring exists within that string and, if it does, to obtain an integer value that indicates the starting position of the first appearance of the given substring.

— The GetLength routine may be used to obtain the length of that string as an integer.

— The GetSubString routine may be used to retrieve a portion of a string, or alternatively, to create a new Large Object value at the server which is a portion of the string and to return a Large Object locator that represents that value.

The Error, GetDiagField, and GetDiagRec routines obtain diagnostic information about the most recent routine operating on a particular resource. The Error routine always retrieves information from the next status record, whereas the GetDiagField and GetDiagRec routines may be used to retrieve information from any status record.

The number of rows affected by the last executed SQL-statement can be obtained by using the RowCount or GetDiagField routine.

An SQL-transaction is terminated by using the EndTran routine. An SQL-transaction is implicitly initiated whenever a CLI routine is invoked that requires the context of an SQL-transaction and no SQL-transaction is active. An SQL-transaction is explicitly started, and its characteristics set, by using the StartTran routine.

NOTE 1 — Applications are prohibited from using the ExecDirect or Execute routines to execute <start transaction statement>s, <commit statement>s, <rollback statement>s, and <release savepoint statement>s.

The Cancel routine is used to cancel the execution of a concurrently executing SQL/CLI routine; it is also used to terminate the processing of deferred parameter values and the execution of the associated SQL-statement.

The GetFeatureInfo, GetFunctions, GetInfo, GetSessionInfo, and GetTypeInfo routines are used to obtain information about the implementation. The DataSources routine returns a list of names that identify SQL-servers to which the SQL/CLI application may be able to connect and returns a description of each such SQL-server.

4.2 Return codes

The execution of a CLI routine causes one or more conditions to be raised. The status of the execution is indicated by a code that is returned either as the result of invoking a CLI routine that is a CLI function or as the value of the ReturnCode argument of a CLI routine that is a CLI procedure.

The return code values and meanings are described in the following list. If more than one return code is possible, then the one appearing later in the list is the one returned.

— A value of 0 (zero) indicates **Success**. The CLI routine executed successfully.

— A value of 1 (one) indicates **Success with information**. The CLI routine executed successfully but a completion condition was raised: *warning*.

— A value of 100 indicates **No data found**. The CLI routine executed successfully but a completion condition was raised: *no data*.

— A value of 99 indicates **Data needed**. The CLI routine did not complete its execution because additional data is needed. An exception condition was raised: *CLI-specific condition — dynamic parameter value needed*.

— A value of –1 indicates **Error**. The CLI routine did not execute successfully. An exception condition other than *CLI-specific condition — invalid handle* or *CLI-specific condition — dynamic parameter value needed* was raised.

— A value of –2 indicates **Invalid handle**. The CLI routine did not execute successfully because an exception condition was raised: *CLI-specific condition — invalid handle*.

After the execution of a CLI routine, the values of every output argument that corresponds to an output parameter whose value is not explicitly defined by this part of ISO/IEC 9075 is implementation-dependent.

In addition to providing the return code, for all CLI routines other than Error, GetDiagField, and GetDiagRec, the SQL/CLI implementation records information about completion conditions and about exception conditions other than *CLI-specific condition — invalid handle* in the diagnostics area associated with the resource being utilized. The *resource being utilized* by a routine is the resource identified by its input handle. In the case of CopyDesc, which takes two input handles, the resource being utilized is the one identified by TargetDescHandle.

4.3 Diagnostics areas in SQL/CLI

Each diagnostics area comprises header information consisting of fields that contain general information relating to the routine that was executed and zero (0) or more status records containing information about individual conditions that occurred during the execution of the CLI routine. A condition that causes a status record to be generated is referred to as a *status condition*.

At the beginning of the execution of any CLI routine other than Error, GetDiagField, and GetDiagRec, the diagnostics area for the resource being utilized is emptied. If the execution of such a routine does not result in the exception condition *CLI-specific condition — invalid handle* or the exception condition *CLI-specific condition — dynamic parameter value needed*, then:

— Header information is generated in the diagnostics area.

— If the routine's return code indicates **Success**, then no status records are generated.

— If the routine's return code indicates **Success with information** or **Error**, then one or more status records are generated.

— If the routine's return code indicates **No data found**, then no status record is generated corresponding to SQLSTATE value '02000' but there may be status records generated corresponding to SQLSTATE value '02*nnn*', where '*nnn*' is an implementation-defined subclass value.

When Fetch or FetchScroll is invoked, the resulting rowset has one or more rows, and exceptions or warnings are generated, then the corresponding records in the diagnostics area have the ROW_NUMBER field set to the row number of the row in the rowset associated with the exceptions or warnings. If a status record does not correspond to any row in the rowset, or the record is generated as a result of calling a routine other than Fetch or FetchScroll, the ROW_NUMBER field is set to zero. The COLUMN_NUMBER field of the status record contains the column number (if any) to which this exception or warning condition applies. If the status record does not apply to any column, then COLUMN_NUMBER is set to zero.

Status records in the diagnostics area are ordered by ROW_NUMBER. If multiple status records are generated for the same ROW_NUMBER value, then the order in which the second and subsequent of those status records appear is implementation-dependent. Which of those status records appears first is also implementation-dependent, except that:

— Status records corresponding to *transaction rollback* have precedence over status records corresponding to other exceptions, which in turn have precedence over status records corresponding to the completion condition *no data*, which in turn have precedence over status records corresponding to the completion condition *warning*.

— Apart from any status records corresponding to an implementation-specified *no data*, any status record corresponding to an implementation-specified condition that duplicates, in whole or in part, a condition defined in this part of ISO/IEC 9075 shall not be the first status record.

The routines GetDiagField and GetDiagRec retrieve information from a diagnostics area. The SQL/CLI application identifies which diagnostics area is to be accessed by providing the handle of the relevant resource as an input argument. The routines return a result code but do not modify the identified diagnostics area.

The Error routine also retrieves information from a diagnostics area. The Error routine retrieves the status records in the identified diagnostics area one at a time but does not permit already processed status records to be retrieved. Error returns a result code but does not modify the identified diagnostics area.

The RowCount routine retrieves the ROW_COUNT field from the diagnostics area for the specified statement handle. RowCount returns a result code and may cause status records to be generated.

A CLI diagnostics area comprises the header fields specified under “Header fields” Table 1, “Header fields in SQL/CLI diagnostics areas”, as well as zero (0) or more status records, each of which comprises the fields specified under “Status record fields” Table 2, “Status record fields in SQL/CLI diagnostics areas”.

Table 1 — Header fields in SQL/CLI diagnostics areas

Field	**Data type**
DYNAMIC_FUNCTION	CHARACTER VARYING (*L1*)†
DYNAMIC_FUNCTION_CODE	INTEGER
MORE	INTEGER
NUMBER	INTEGER
RETURNCODE	SMALLINT
ROW_COUNT	INTEGER
TRANSACTIONS_COMMITTED	INTEGER
TRANSACTIONS_ROLLED_BACK	INTEGER

Field	Data type
TRANSACTION_ACTIVE	INTEGER
Implementation-defined header field	Implementation-defined data type
† **Where** *L* is an implementation-defined integer not less than 128 and *L1* is an implementation-defined integer not less than 254.	

Table 2 — Status record fields in SQL/CLI diagnostics areas

Field	Data type
CATALOG_NAME	CHARACTER VARYING (*L*)†
CLASS_ORIGIN	CHARACTER VARYING (*L1*)†
COLUMN_NAME	CHARACTER VARYING (*L*)†
COLUMN_NUMBER	INTEGER
CONDITION_IDENTIFIER	CHARACTER VARYING (*L*)†
CONDITION_NUMBER	INTEGER
CONNECTION_NAME	CHARACTER VARYING (*L*)†
CONSTRAINT_CATALOG	CHARACTER VARYING (*L*)†
CONSTRAINT_NAME	CHARACTER VARYING (*L*)†
CONSTRAINT_SCHEMA	CHARACTER VARYING (*L*)†
CURSOR_NAME	CHARACTER VARYING (*L*)†
MESSAGE_LENGTH	INTEGER
MESSAGE_OCTET_LENGTH	INTEGER
MESSAGE_TEXT	CHARACTER VARYING (*L1*)†
NATIVE_CODE	INTEGER
PARAMETER_MODE	CHARACTER VARYING (*L*)†
PARAMETER_NAME	CHARACTER VARYING (*L*)†

Field	**Data type**
PARAMETER_ORDINAL_POSITION	INTEGER
ROUTINE_CATALOG	CHARACTER VARYING (*L*)†
ROUTINE_NAME	CHARACTER VARYING (*L*)†
ROUTINE_SCHEMA	CHARACTER VARYING (*L*)†
ROW_NUMBER	INTEGER
SCHEMA_NAME	CHARACTER VARYING (*L*)†
SERVER_NAME	CHARACTER VARYING (*L*)†
SQLSTATE	CHARACTER (5)
SPECIFIC_NAME	CHARACTER VARYING (*L*)†
SUBCLASS_ORIGIN	CHARACTER VARYING (*L1*)†
TABLE_NAME	CHARACTER VARYING (*L*)†
TRIGGER_CATALOG	CHARACTER VARYING (*L*)†
TRIGGER_NAME	CHARACTER VARYING (*L*)†
TRIGGER_SCHEMA	CHARACTER VARYING (*L*)†
Implementation-defined status field	Implementation-defined data type
† **Where** *L* is an implementation-defined integer not less than 128 and *L1* is an implementation-defined integer not less than 254.	

All diagnostics area fields specified in other parts of ISO/IEC 9075 that are not included in this table are not applicable to SQL/CLI.

4.3.1 Setting of ROW_NUMBER and COLUMN_NUMBER fields

Except where otherwise specified in this part of ISO/IEC 9075, the ROW_NUMBER and COLUMN_NUMBER fields in a status record are always 0 (zero).

4.4 Miscellaneous characteristics

4.4.1 Handles

The AllocHandle routine returns a handle that uniquely identifies the allocated resource. Although the data type of a handle parameter is INTEGER, its value has no meaning in any other context and should not be used as a numeric operand or modified in any way.

In general, if the related resource cannot be allocated, then a handle value of zero is returned. However, even if a resource has been successfully allocated, processing of that resource can subsequently fail due to memory constraints as follows:

— If additional memory is required but is not available, then an exception condition is raised: *CLI-specific condition — memory allocation error*.

— If previously allocated memory cannot be accessed, then an exception condition is raised: *CLI-specific condition — memory management error*.

NOTE 2 — No diagnostic information is generated in this case.

The validity of a handle in a compilation unit other than the one in which the identified resource was allocated is implementation-defined.

Specifying (the address of) a valid handle as the output handle for an invocation of AllocHandle does not have the effect of reinitializing the identified resource. Instead, a new resource is allocated and a new handle value overwrites the old one.

4.4.2 Null terminated strings

An input character string provided by the SQL/CLI application may be terminated by the implementation-defined null character that terminates C character strings. If this technique is used, the application may set the associated length argument to either the length of the string excluding the null terminator or to –3, indicating NULL TERMINATED.

If the NULL TERMINATION attribute for the SQL-environment is *True*, then all output character strings returned by the SQL/CLI implementation are terminated by the implementation-defined null character that terminates C character strings. If the NULL TERMINATION attribute is *False*, then output character strings are not null terminated.

4.4.3 Null pointers

If the programming language of the invoking SQL/CLI application supports pointers, then the SQL/CLI application may provide a zero-valued pointer, referred to as a null pointer, in the following circumstances:

— In lieu of an output argument that is to receive the length of a returned character string. This indicates that the SQL/CLI application wishes to prohibit the return of this information.

— In lieu of other output arguments where specifically allowed by this part of ISO/IEC 9075. This indicates that the SQL/CLI application wishes to prohibit the return of this information.

— In lieu of input arguments where specifically allowed by this part of ISO/IEC 9075. The semantics of such a specification depend on the context.

If the SQL/CLI application provides a null pointer in any other circumstances, then an exception condition is raised: *CLI-specific condition — invalid use of null pointer*.

If the NULL TERMINATION attribute for the SQL-environment is *False*, then specifying a zero buffer size for an output argument is equivalent to specifying a null pointer for that output argument.

4.4.4 Environment attributes

Environment attributes are associated with each allocated SQL-environment and affect the behavior of CLI functions in that SQL-environment.

The GetEnvAttr routine enables the SQL/CLI application to determine the current value of a specific attribute. For attributes that may be set by the user, the SetEnvAttr routine enables the SQL/CLI application to set the value of a specific attribute. Attribute values may be set by the SQL/CLI application whenever there are no SQL-connections allocated within the SQL-environment.

Table 16, “Codes used for environment attributes”, and Table 20, “Data types of attributes”, in Subclause 5.14, “Other tables associated with CLI”, indicate for each attribute its name, code value, data type, possible values, and whether the attribute may be set using SetEnvAttr.

The NULL TERMINATION attribute determines whether output character strings are null terminated by the SQL/CLI implementation. The attribute is set to *True* when an SQL-environment is allocated.

4.4.5 Connection attributes

Connection attributes are associated with each allocated SQL-connection and affect the behavior of CLI functions operating in the context of that allocated SQL-connection.

The GetConnectAttr routine enables the SQL/CLI application to determine the current value of a specific connection attribute. For connection attributes that may be set by the user, the SetConnectAttr routine enables the SQL/CLI application to set the value of a specific connection attribute.

Table 17, “Codes used for connection attributes”, and Table 20, “Data types of attributes”, in Subclause 5.14, “Other tables associated with CLI”, indicate for each connection attribute its name, code value, data type, possible values and whether the connection attribute may be set using SetConnectAttr.

The POPULATE IPD attribute determines whether the SQL/CLI implementation will populate the implementation parameter descriptor with an item descriptor area for each <dynamic parameter specification> when an SQL-statement is prepared or executed immediately. The POPULATE IPD attribute is automatically set each time an SQL-connection is established for the allocated SQL-connection.

The SAVEPOINT NAME connection attribute specifies the savepoint to be referenced in an invocation of the EndTran routine that uses the SAVEPOINT NAME ROLLBACK or SAVEPOINT NAME RELEASE CompletionType, respectively. The SAVEPOINT NAME attribute is set to a zero-length string when the SQL-connection is allocated.

4.4.6 Statement attributes

Statement attributes are associated with each allocated SQL-statement and affect the processing of SQL-statements under that allocated SQL-statement.

The GetStmtAttr routine enables the SQL/CLI application to determine the current value of a specific statement attribute. For statement attributes that may be set by the user, the SetStmtAttr routine enables the SQL/CLI application to set the value of a specific statement attribute.

Table 18, "Codes used for statement attributes", and Table 20, "Data types of attributes", in Subclause 5.14, "Other tables associated with CLI", indicate for each statement attribute its name, code value, data type, possible values, and whether the statement attribute may be set by using SetStmtAttr.

The APD HANDLE statement attribute is the value of the handle of the current application parameter descriptor for the allocated SQL-statement. The statement attribute is set to the value of the handle of the automatically allocated application parameter descriptor when the SQL-statement is allocated.

The ARD HANDLE statement attribute is the value of the handle of the current application row descriptor for the allocated SQL-statement. The statement attribute is set to the value of the handle of the automatically allocated application row descriptor when the SQL-statement is allocated.

The IPD HANDLE statement attribute is the value of the handle of the implementation parameter descriptor associated with the allocated SQL-statement. The statement attribute is set to the value of the handle of the automatically allocated implementation parameter descriptor when the SQL-statement is allocated.

The IRD HANDLE statement attribute is the value of the handle of the implementation row descriptor associated with the allocated SQL-statement. The statement attribute is set to the value of the handle of the automatically allocated implementation row descriptor when the SQL-statement is allocated.

The CURSOR SCROLLABLE statement attribute determines the *scrollability* implicitly declared when Execute or ExecDirect are invoked. The statement attribute is set to NONSCROLLABLE when the SQL-statement is allocated. The CURSOR SENSITIVITY statement attribute determines the *sensitivity* to changes of the cursor implicitly declared when Execute or ExecDirect are invoked. The statement attribute is set to ASENSITIVE when the SQL-statement is allocated.

The CURSOR HOLDABLE statement attribute determines the *holdability* of the cursor implicitly declared when Execute or ExecDirect are invoked. The statement attribute is set to HOLDABLE or NONHOLDABLE when the statement is allocated, depending on the values of the CURSOR COMMIT BEHAVIOR item used by the GetInfo routine.

The statement attribute CURRENT OF POSITION identifies the row in the rowset to which a positioned update or delete operation applies. This is set to 1 (one) when an SQL-statement is initially allocated. It is reset to 1 (one) whenever Fetch or FetchScroll are successfully executed.

The NEST DESCRIPTOR statement attribute determines whether nested descriptor items are permitted in a CLI descriptor. Nested descriptor items are used to describe ROW, ARRAY, and MULTISET data types. The statement attribute is set to FALSE when the SQL-statement is allocated.

4.4.7 CLI descriptor areas

A *CLI descriptor area* provides an interface for a description of <dynamic parameter specification>s, <dynamic parameter specification> values, result columns of <dynamic select statement>s and <dynamic single row select statement>s, or <target specification>s for the result columns.

Each descriptor area comprises *header field*s and zero or more *item descriptor area*s. The header fields are specified in Table 6, "Fields in SQL/CLI row and parameter descriptor areas". The header fields include a COUNT field that indicates the number of item descriptor areas and an ALLOC_TYPE field that indicates whether the CLI descriptor area was allocated by the user or automatically allocated by the SQL/CLI implementation.

The header fields include ARRAY_SIZE, ARRAY_STATUS_POINTER, and ROWS_PROCESSED_POINTER. These three fields are used to support the fetching of multiple rows with one invocation of Fetch or FetchScroll.

Each CLI item descriptor area consists of the fields specified following "Status record fields" in Table 6, "Fields in SQL/CLI row and parameter descriptor areas".

The CLI descriptor areas for the four interface types are referred to as an *implementation parameter descriptor* (IPD), an *application parameter descriptor* (APD), an *implementation row descriptor* (IRD), and an *application row descriptor* (ARD), respectively. IPDs and IRDs are collectively known as *implementation descriptor area*s; APDs and ARDs are collectively known as *application descriptor area*s.

When an SQL-statement is allocated, a CLI descriptor area of each type is automatically allocated by the SQL/CLI implementation. The ALLOC_TYPE fields for these CLI descriptor areas are set to indicate AUTOMATIC. A CLI descriptor area allocated by the user has its ALLOC_TYPE field set to indicate USER, and can only be used as an APD or ARD. The handle values of the IPD, IRD, current APD, and current ARD are attributes of the allocated SQL-statement. The SQL/CLI application can determine the current values of these attributes by using the routine GetStmtAttr. The current APD and ARD are initially the automatically-allocated APD and ARD, respectively, but can subsequently be changed by changing the corresponding attribute value using the routine SetStmtAttr.

The routines GetDescField and GetDescRec enable information to be retrieved from any CLI descriptor area. The routines SetDescField and SetDescRec enable information to be set in any CLI descriptor area except an IRD. The routine BindCol implicitly sets information in the current ARD. The routine BindParameter implicitly sets information in the current APD and the current IPD. The CopyDesc routine enables the contents of any CLI descriptor area to be copied to any CLI descriptor area except an IRD.

NOTE 3 — Although there is no need to set a DATA_POINTER field in the IPD to align with the consistency check that applies in the case of an APD or ARD, setting this field causes the item descriptor area to be validated.

4.4.8 Obtaining diagnostics during multi-row fetch

When Fetch or FetchScroll is used to fetch a rowset, exceptions or warnings may be raised during the retrieval of one or more rows in the rowset. The status of each row (that is, information about whether that row in the rowset was successfully retrieved or not) is available in the array addressed by the ARRAY_STATUS_POINTER field of the applicable IRD. The cardinality of this array is the same as the ARRAY_SIZE field of the corresponding ARD. For each row in the rowset, the corresponding element of this array has one of the following values:

— A value of 0 (zero) indicates **Row success**, meaning that the row was fetched successfully.

— A value of 6 indicates **Row success with information**, meaning that the row was fetched successfully, but a completion condition was raised: *warning*.

— A value of 3 indicates **No row**, meaning that there is no row at this position in the rowset. This condition occurs when a partial rowset is retrieved because the result set ended.

— A value of 5 indicates **Row error**, meaning that the row was not fetched successfully and an exception condition was raised.

Each **Row success with information** or **Row Error** generates one or more status records in the diagnostics area. The ROW_NUMBER field for each status record has the value of the row position within the rowset to which this status record corresponds.

4.5 Client-server operation

This Subclause modifies Subclause 4.39, “Client-server operation”, in ISO/IEC 9075-2.

Insert this paragraph If the execution of a CLI routine causes the implicit or explicit execution of an <SQL procedure statement> by an SQL-server, diagnostic information is passed in an implementation-dependent manner to the SQL-client and then into the appropriate diagnostics area. The effect on diagnostic information of incompatibilities between the character repertoires supported by the SQL-client and the SQL-server is implementation-dependent.

(Blank page)

5 Call-Level Interface specifications

This Clause is modified by Clause 19, "Call-Level Interface specifications", in ISO/IEC 9075-9.

5.1 <CLI routine>

This Subclause is modified by Subclause 19.1, "<CLI routine>", in ISO/IEC 9075-9.

Function

Describe SQL/CLI routines in a generic fashion.

Format

```
<CLI routine> ::=
  <CLI routine name> <CLI parameter list> [ <CLI returns clause> ]

<CLI routine name> ::=
  <CLI name prefix><CLI generic name>

<CLI name prefix> ::=
    <CLI by-reference prefix>
  | <CLI by-value prefix>

<CLI by-reference prefix> ::=
  SQLR

<CLI by-value prefix> ::=
  SQL

<CLI generic name> ::=
    AllocConnect
  | AllocEnv
  | AllocHandle
  | AllocStmt
  | BindCol
  | BindParameter
  | Cancel
  | CloseCursor
  | ColAttribute
  | ColumnPrivileges
  | Columns
  | Connect
  | CopyDesc
  | DataSources
  | DescribeCol
  | Disconnect
```

```
  | EndTran
  | Error
  | ExecDirect
  | Execute
  | Fetch
  | FetchScroll
  |ForeignKeys
  | FreeConnect
  | FreeEnv
  | FreeHandle
  | FreeStmt
  | GetConnectAttr
  | GetCursorName
  | GetData
  | GetDescField
  | GetDescRec
  | GetDiagField
  | GetDiagRec
  | GetEnvAttr
  | GetFeatureInfo
  | GetFunctions
  | GetInfo
  | GetLength
  | GetParamData
  | GetPosition
  | GetSessionInfo
  | GetStmtAttr
  | GetSubString
  | GetTypeInfo
  | MoreResults
  | NextResult
  | NumResultCols
  | ParamData
  | Prepare
  | PrimaryKeys
  | PutData
  | RowCount
  | SetConnectAttr
  | SetCursorName
  | SetDescField
  | SetDescRec
  | SetEnvAttr
  | SetStmtAttr
  | SpecialColumns
  | StartTran
  | TablePrivileges
  | Tables
  | <implementation-defined CLI generic name>

<CLI parameter list> ::=
  <left paren> <CLI parameter declaration>
      [ { <comma> <CLI parameter declaration> }... ] <right paren>

<CLI parameter declaration> ::=
  <CLI parameter name> <CLI parameter mode> <CLI parameter data type>

<CLI parameter name> ::=
  !! See the individual CLI routine definitions
```

```
<CLI parameter mode> ::=
    IN
  | OUT
  | DEFIN
  | DEFOUT
  | DEF

<CLI parameter data type> ::=
    INTEGER
  | SMALLINT
  | ANY
  | CHARACTER <left paren> <length> <right paren>

<CLI returns clause> ::=
  RETURNS SMALLINT

<implementation-defined CLI generic name> ::=
  !! See the Syntax Rules
```

Syntax Rules

1) <CLI routine> is a pre-defined routine written in a programming language that is invoked by a compilation unit of the same programming language. Let *HL* be that programming language.

2) <CLI routine> that contains a <CLI returns clause> is called a *CLI function*. A <CLI routine> that does not contain a <CLI returns clause> is called a *CLI procedure*.

3) There shall be no <separator> between the <CLI name prefix> and the <CLI generic name>.

4) For each CLI function *CF*, there is a corresponding CLI procedure *CP*, with the same <CLI routine name>. The <CLI parameter list> for *CP* is the same as the <CLI parameter list> for *CF* but with the following additional <CLI parameter declaration>:

```
ReturnCode OUT SMALLINT
```

5) *HL* shall support either the invocation of *CF* or the invocation of *CP*. It is implementation-defined which is supported.

6) Case:

   a) If <CLI parameter mode> is IN, then the parameter is an *input parameter*. The value of an input argument is established when a CLI routine is invoked.

   b) If <CLI parameter mode> is OUT, then the parameter is an *output parameter*. The value of an output argument is established when a CLI routine is executed.

   c) If <CLI parameter mode> is DEFIN, then the parameter is a *deferred input parameter*. The value of a deferred input argument for a CLI routine *R* is not established when *R* is invoked, but subsequently during the execution of a related CLI routine.

   d) If <CLI parameter mode> is DEFOUT, then the parameter is a *deferred output parameter*. The value of a deferred output argument for a CLI routine *R* is not established by the execution of *R* but subsequently by the execution of a related CLI routine.

e) If <CLI parameter mode> is DEF, then the parameter is a *deferred parameter*. The value of a deferred argument for a CLI routine *R* is not established by the execution of *R* but subsequently by the execution of a related CLI routine.

7) The value of an output, deferred output, deferred input, or deferred parameter is an address. It is either a non-pointer host variable passed by reference or a pointer host variable passed by value.

8) A *by-value version* of a CLI routine is a version that expects each of its non-character input parameters to be provided as actual values. A *by-reference version* of a CLI routine is a version that expects each of its input parameters to be provided as an address. By-value and by-reference versions of the CLI routines shall be supported according to Table 3, "Supported calling conventions of SQL/CLI routines by language".

Table 3 — Supported calling conventions of SQL/CLI routines by language

Language	**By-value**	**By-reference**
Ada (ISO 8652)	Optional	Required
C (ISO/IEC 9899)	Required	Optional
COBOL (ISO 1989)	Optional	Required
Fortran (ISO/IEC 1539)	Not supported	Required
M (ISO/IEC 11756)	Optional	Required
Pascal (ISO/IEC 7185 and ISO/IEC 10206)	Optional	Required
PL/I (ISO 6160)	Optional	Required

9) If a <CLI routine> is a by-reference routine, then its <CLI routine name> shall contain a <CLI by-reference prefix>. Otherwise, its <CLI routine name> shall contain a <CLI by-value prefix>.

10) The <implementation-defined CLI generic name> for an implementation-defined CLI function shall be different from the <CLI generic name> of any other CLI function. The <implementation-defined CLI generic name> for an implementation-defined CLI procedure shall be different from the <CLI generic name> of any other CLI procedure.

11) Any <CLI routine name> that cannot be used by an implementation because of its length or because it is made identical to some other <CLI routine name> by truncation is effectively replaced with an abbreviated name according to the following rules:

a) Any <CLI by-value prefix> remains unchanged.

b) Any <CLI by-reference prefix> is replaced by SQR.

c) The <CLI generic name> is replaced by an abbreviated version according to Table 4, "Abbreviated SQL/CLI generic names".

Table 4 — Abbreviated SQL/CLI generic names

Generic Name	Abbreviation
AllocConnect	AC
AllocEnv	AE
AllocHandle	AH
AllocStmt	AS
BindCol	BC
BindParameter	BP
Cancel	CAN
CloseCursor	CC
ColAttribute	CO
ColumnPrivileges	CP
Columns	COL
Connect	CON
CopyDesc	CD
DataSources	DS
DescribeCol	DC
Disconnect	DIS
EndTran	ET
Error	ER
ExecDirect	ED
Execute	EX
Fetch	FT
FetchScroll	FTS
ForeignKeys	FK
FreeConnect	FC

Generic Name	Abbreviation
FreeEnv	FE
FreeHandle	FH
FreeStmt	FS
GetConnectAttr	GCA
GetCursorName	GCN
GetData	GDA
GetDescField	GDF
GetDescRec	GDR
GetDiagField	GXF
GetDiagRec	GXR
GetEnvAttr	GEA
GetFeatureInfo	GFI
GetFunctions	GFU
GetInfo	GI
GetLength	GLN
GetParamData	GPD
GetPosition	GPO
GetSessionInfo	GSI
GetStmtAttr	GSA
GetSubString	GSB
GetTypeInfo	GTI
MoreResults	MR
NextResult	NR
NumResultCols	NRC
ParamData	PRD

Generic Name	**Abbreviation**
Prepare	PR
PrimaryKeys	PK
PutData	PTD
RowCount	RC
SetConnectAttr	SCA
SetCursorName	SCN
SetDescField	SDF
SetDescRec	SDR
SetEnvAttr	SEA
SetStmtAttr	SSA
SpecialColumns	SC
StartTran	STN
TablePrivileges	TP
Tables	TAB
Implementation-defined CLI routine	Implementation-defined abbreviation

12) Let *CR* be a <CLI routine> and let *RN* be its <CLI routine name>. Let *RNU* be the value of UPPER(*RN*).

    Case:

    a) If *HL* supports case sensitive routine names, then the name used for the invocation of *CR* shall be *RN*.

    b) If *HL* does not support <simple Latin lower case letter>s, then the name used for the invocation of *CR* shall be *RNU*.

    c) If *HL* does not support case sensitive routine names, then the name used for the invocation of *CR* shall be *RN* or *RNU*.

13) Let *operative data type correspondence table* be the data type correspondence table for *HL* as specified in Subclause 5.15, “SQL/CLI data type correspondences”. Refer to the two columns of the operative data type correspondence table as the “*SQL data type column*” and the “*host data type column*”.

14) Let *TI*, *TS*, *TC*, and *TV* be the types listed in the host data type column for the rows that contains INTEGER, SMALLINT, CHARACTER(*L*) and CHARACTER VARYING(*L*), respectively, in the SQL data type column.

    a) If *TS* is “None”, then let *TS* = *TI*.

b) If *TC* is “None”, then let *TC* = *TV*.

c) For each parameter *P*,

Case:

i) If the CLI parameter data type is INTEGER, then the type of the corresponding argument shall be *TI*.

ii) If the CLI parameter data type is SMALLINT, then the type of the corresponding argument shall be *TS*.

iii) If the CLI parameter data type is CHARACTER(*L*), then the type of the corresponding argument shall be *TC*.

iv) If the CLI parameter data type is ANY, then

Case:

1) If *HL* is *C*, then the type of the corresponding argument shall be “`void *`”.

2) Otherwise, the type of the corresponding argument shall be any type (other than “None”) listed in the host data type column.

d) If the CLI routine is a CLI function, then the type of the returned value is *TS*.

Access Rules

None.

General Rules

1) The rules for invocation of a <CLI routine> are specified in Subclause 5.2, “<CLI routine> invocation”.

Conformance Rules

1) Without Feature C001, “CLI routine invocation in Ada”, a conforming SQL/CLI application shall not contain an invocation of a <CLI routine> written in Ada.

2) Without Feature C002, “CLI routine invocation in C”, a conforming SQL/CLI application shall not contain an invocation of a <CLI routine> written in C.

3) Without Feature C003, “CLI routine invocation in COBOL”, a conforming SQL/CLI application shall not contain an invocation of a <CLI routine> written in COBOL.

4) Without Feature C004, “CLI routine invocation in Fortran”, a conforming SQL/CLI application shall not contain an invocation of a <CLI routine> written in Fortran.

5) Without Feature C005, “CLI routine invocation in MUMPS”, a conforming SQL/CLI application shall not contain an invocation of a <CLI routine> written in M.

6) Without Feature C006, “CLI routine invocation in Pascal”, a conforming SQL/CLI application shall not contain an invocation of a <CLI routine> written in Pascal.

7) Without Feature C007, “CLI routine invocation in PL/I”, a conforming SQL/CLI application shall not contain an invocation of a <CLI routine> written in PL/I.

5.2 <CLI routine> invocation

Function

Specify the rules for invocation of a <CLI routine>.

Syntax Rules

1) Let *HL* be the programming language of the invoking host program.

2) A CLI function or CLI procedure is invoked by the *HL* mechanism for invoking functions or procedures, respectively.

3) Let *RNM* be the <CLI routine name> of the <CLI routine> invoked by the host program and let *RN* be the SQL/CLI routine identified by *RNM*. The number of arguments provided in the invocation shall be the same as the number of <CLI parameter declaration>s for *RN*.

4) Let *DA* be the data type of the ***i***-th argument in the invocation and let *DP* be the <CLI parameter data type> of the ***i***-th <CLI parameter declaration> of *RN*. *DA* shall be the *HL* equivalent of *DP* as specified by the rules of Subclause 5.1, "<CLI routine>".

General Rules

1) If the value of any input argument provided by the host program is not a value of the data type of the parameter, or if the value of any output argument resulting from the execution of the <CLI routine> is not a value supported by the SQL/CLI application for that parameter, then the effect is implementation-defined.

2) Let *GRN* be the <CLI generic name> of *RN*.

3) When the <CLI routine> is called by the SQL/CLI application:

   a) The values of all input arguments to *RN* are established.

   b) Case:

      i) If *RN* is a CLI routine with a statement handle as an input parameter, *RN* has no accompanying handle type parameter, and *GRN* is not Error, then:

         1) If the statement handle does not identify an allocated SQL-statement, then an exception condition is raised: *CLI-specific condition — invalid handle*. Otherwise, let *S* be the allocated SQL-statement identified by the statement handle.

         2) If *GRN* is not Cancel, then the diagnostics area associated with *S* is emptied.

         3) Let *C* be the allocated SQL-connection with which *S* is associated.

         4) If there is no established SQL-connection associated with *C*, then an exception condition is raised: *connection exception — connection does not exist*. Otherwise, let *EC* be the established SQL-connection associated with *C*.

         5) If *EC* is not the current SQL-connection, then the General Rules of Subclause 5.3, "Implicit set connection", are applied with EC as *dormant SQL-connection*.

6) If *GRN* is neither Cancel nor ParamData nor PutData and there is a deferred parameter number associated with *S*, then an exception condition is raised: *CLI-specific condition — function sequence error*.

7) *RN* is invoked.

ii) If *RN* is a CLI routine with a descriptor handle as an input parameter and *RN* has no accompanying handle type parameter and *GRN* is not CopyDesc, then:

1) If the descriptor handle does not identify an allocated CLI descriptor area, then an exception condition is raised: *CLI-specific condition — invalid handle*. Otherwise, let *D* be the allocated CLI descriptor area identified by the descriptor handle.

2) The diagnostics area associated with *D* is emptied.

3) Let *C* be the allocated SQL-connection with which *D* is associated.

4) If there is no established SQL-connection associated with *C*, then an exception condition is raised: *connection exception — connection does not exist*. Otherwise, let *EC* be the established SQL-connection associated with *C*.

5) If *EC* is not the current SQL-connection, then the General Rules of Subclause 5.3, "Implicit set connection", are applied with EC as *dormant SQL-connection*.

6) *RN* is invoked.

iii) Otherwise, *RN* is invoked.

4) Case:

a) If *RN* is a CLI function, then:

i) The values of all output arguments are established.

ii) Let *RC* be the return value.

b) If *RN* is a CLI procedure, then:

i) The values of all output arguments are established except for the argument associated with the ReturnCode parameter.

ii) Let *RC* be the argument associated with the ReturnCode parameter.

5) Case:

a) If *RN* did not complete execution because it requires more input data, then:

i) *RC* is set to indicate **Data needed**.

ii) An exception condition is raised: *CLI-specific condition — dynamic parameter value needed*.

b) If *RN* executed successfully, then:

i) Either a completion condition is raised: *successful completion*, or a completion condition is raised: *warning*, or a completion condition is raised: *no data*.

ii) Case:

1) If a completion condition is raised: *successful completion*, then *RC* is set to indicate **Success**.

2) If a completion condition is raised: *warning*, then *RC* is set to indicate **Success with information**.

3) If a completion condition is raised: *no data*, then *RC* is set to indicate **No data found**.

c) If *RN* did not execute successfully, then:

i) All changes made to SQL-data or schemas by the execution of *RN* are canceled.

ii) One or more exception conditions are raised as determined by the General Rules of this and other Subclauses of this part of ISO/IEC 9075 or by implementation-defined rules.

iii) Case:

1) If an exception condition is raised: *CLI-specific condition — invalid handle*, then *RC* is set to indicate **Invalid handle**.

2) Otherwise, *RC* is set to indicate **Error**.

6) Case:

a) If *GRN* is neither Error nor GetDiagField nor GetDiagRec, and *RC* indicates neither **Invalid handle** nor **Data needed**, then diagnostic information resulting from the execution of *RN* is placed into the appropriate diagnostics area as specified in Subclause 4.2, “Return codes”, and Subclause 4.3, “Diagnostics areas in SQL/CLI”.

b) Otherwise, no diagnostics area is updated.

5.3 Implicit set connection

Function

Specify the rules for an implicit SET CONNECTION statement.

General Rules

1) Let *DC* be the *dormant SQL-connection* specified in an application of this Subclause.

2) If an SQL-transaction is active for the current SQL-connection and the SQL-implementation does not support transactions that affect more than one SQL-server, then an exception condition is raised: *feature not supported — multiple server transactions*.

3) If *DC* cannot be selected, then an exception condition is raised: *connection exception — connection failure*.

4) The current SQL-connection *CC* and current SQL-session become a dormant SQL-connection and a dormant SQL-session, respectively. The SQL-session context for *CC* is preserved and is not affected in any way by operations performed over the selected SQL-connection.

   NOTE 4 — The SQL-session context is defined in Subclause 4.37, "SQL-sessions", in ISO/IEC 9075-2.

5) *DC* becomes the current SQL-connection and the SQL-session associated with *DC* becomes the *current SQL-session*. The SQL-session context is restored to the same state as at the time *DC* became dormant.

   NOTE 5 — The SQL-session context information is defined in Subclause 4.37, "SQL-sessions", in ISO/IEC 9075-2.

6) The SQL-server for the subsequent execution of SQL-statements via CLI routine invocations is set to that of the current SQL-connection.

5.4 Implicit cursor

Function

Specify the rules for an implicit DECLARE CURSOR and OPEN statement.

General Rules

1) Let *SS* and *AS* be a *SELECT SOURCE* and *ALLOCATED STATEMENT* specified in an application of this Subclause.

2) If there is no cursor associated with *AS*, then a cursor is associated with *AS* and the cursor name associated with *AS* becomes the name of the cursor.

3) The General Rules of Subclause 5.6, "Implicit EXECUTE USING and OPEN USING clauses", are applied with OPEN as *TYPE*, SS as *SOURCE*, and AS as *ALLOCATED STATEMENT*.

4) If the value of the CURSOR SCROLLABLE attribute of *AS* is SCROLLABLE, then let *CT* be 'SCROLL'; otherwise, let *CT* be an empty string.

5) Case:

   a) If the value of the CURSOR SENSITIVITY attribute of *AS* is INSENSITIVE, then let *CS* be 'INSENSITIVE'.

   b) If the value of the CURSOR SENSITIVITY attribute of *AS* is SENSITIVE, then let *CS* be 'SENSITIVE'.

   c) Otherwise, let *CS* be 'ASENSITIVE'.

6) If the value of the CURSOR HOLDABLE attribute of *AS* is HOLDABLE, then let *CH* be 'WITH HOLD'; otherwise, let *CH* be an empty string.

7) Let *CN* be the name of the cursor associated with *AS* and let *CR* be the following <declare cursor>:

```
DECLARE CN CS CT CURSOR CH FOR SS
```

8) Cursor *CN* is opened in the following steps:

   a) A copy of *SS* is effectively created in which:

      i) Each <dynamic parameter specification> is replaced by the value of the corresponding dynamic parameter.

      ii) Each <value specification> generally contained in *SS* that is CURRENT_USER, CURRENT_ROLE, SESSION_USER, SYSTEM_USER, CURRENT_CATALOG, CURRENT_SCHEMA, CURRENT_PATH, CURRENT_DEFAULT_TRANSFORM_GROUP, or CURRENT_TRANSFORM_GROUP_FOR_TYPE <path-resolved user-defined type name> is replaced by the value resulting from evaluation of CURRENT_USER, CURRENT_ROLE, SESSION_USER, SYSTEM_USER, CURRENT_CATALOG, CURRENT_SCHEMA, CURRENT_PATH, CURRENT_DEFAULT_TRANSFORM_GROUP, or CURRENT_TRANSFORM_GROUP_FOR_TYPE <path-resolved user-defined type name>, respectively, with all such evaluations effectively done at the same instant in time.

iii) Each <datetime value function> generally contained in *SS* is replaced by the value resulting from evaluation of that <datetime value function>, with all such evaluations effectively done at the same instant in time.

b) Let *T* be the table specified by the copy of *SS*.

c) A table descriptor for *T* is effectively created.

d) The General Rules of Subclause 14.1, "<declare cursor>", in ISO/IEC 9075-2 are applied to *CR*.

e) Case:

i) If *CR* specifies INSENSITIVE, then a copy of *T* is effectively created and cursor *CN* is placed in the open state and its position is before the first row of the copy of *T*.

ii) Otherwise, cursor *CN* is placed in the open state and its position is before the first row of *T*.

9) If *CR* specifies INSENSITIVE, and the SQL-implementation is unable to guarantee that significant changes will be invisible through *CR* during the SQL-transaction in which *CR* is opened and every subsequent SQL-transaction during which it may be held open, then an exception condition is raised: *cursor sensitivity exception — request rejected*.

10) If *CR* specifies SENSITIVE, and the SQL-implementation is unable to guarantee that significant changes will be visible through *CR* during the SQL-transaction in which *CR* is opened, then an exception condition is raised: *cursor sensitivity exception — request rejected*.

NOTE 6 — The visibility of significant changes through a sensitive holdable cursor during a subsequent SQL-transaction is implementation-defined.

11) Whether an implementation is able to disallow significant changes that would not be visible through a currently open cursor is implementation-defined.

5.5 Implicit DESCRIBE USING clause

This Subclause is modified by Subclause 19.2, "Implicit DESCRIBE USING clause", in ISO/IEC 9075-9.

Function

Specify the rules for an implicit DESCRIBE USING clause.

General Rules

1) Let *S* and *AS* be a *SOURCE* and an *ALLOCATED STATEMENT* specified in the rules of this Subclause.

2) Let *IRD* and *IPD* be the implementation row descriptor and implementation parameter descriptor, respectively, associated with *AS*.

3) Let *HL* be the programming language of the invoking host program.

4) The value of DYNAMIC_FUNCTION and DYNAMIC_FUNCTION_CODE in *IRD* and *IPD* are respectively a character string representation of the prepared statement and a numeric code that identifies the type of the prepared statement.

5) A representation of the column descriptors of the <select list> columns for the prepared statement is stored in *IRD* as follows:

   a) Case:

      i) If there is a select source associated with *AS*, then:

         1) Let *TBL* be the table defined by *S* and let *D* be the degree of *TBL*.

            Case:

            A) If the value of the statement attribute NEST DESCRIPTOR is <u>*True*</u>, then let NS_i, 1 (one) $\leq i \leq D$, be the number of subordinate descriptors of the descriptor for the *i*-th column of *T*.

            B) Otherwise, let NS_i, 1 (one) $\leq i \leq D$, be 0 (zero).

         2) TOP_LEVEL_COUNT is set to *D*. If *D* is 0 (zero), then let *TD* be 0 (zero); otherwise, let *TD* be $D + \sum_{i=1}^{D} (NS_i)$. COUNT is set to *TD*.

         3) Let *SL* be the collection of <select list> columns of *TBL*.

         4) Case:

            A) If some subset of *SL* is the primary key of *TBL*, then KEY_TYPE is set to 1 (one).

            B) If some subset of *SL* is the preferred key of *TBL*, then KEY_TYPE is set to 2.

            C) Otherwise, KEY_TYPE is set to 0 (zero).

      ii) Otherwise:

1) Let *D* be 0 (zero). Let *TD* be 0 (zero).

2) KEY_TYPE is set to 0 (zero).

b) If *TD* is zero, then no item descriptor areas are set. Otherwise, the first *TD* item descriptor areas are set so that the *i*-th item descriptor area contains the descriptor of the *j*-th column of *TBL* such that:

i) The descriptor for the first such column is assigned to the first descriptor area.

ii) The descriptor for the *j*+1-th column is assigned to the $i+NS_j+1$-th item descriptor area.

iii) If the value of the statement attribute NEST DESCRIPTOR is *True*, then the implicitly ordered subordinate descriptors for the *j*-th column are assigned to contiguous item descriptor areas starting at the *i*+1-th item descriptor area.

c) The descriptor of a column consists of values for LEVEL, TYPE, NULLABLE, NAME, UNNAMED, KEY_MEMBER, and other fields depending on the value of TYPE as described below. Those fields and fields that are not applicable for a particular value of TYPE are set to implementation-dependent values. The DATA_POINTER, INDICATOR_POINTER, and OCTET_LENGTH_POINTER fields are not relevant in this case.

i) If the item descriptor area is set to a descriptor that is immediately subordinate to another whose LEVEL value is some value *k*, then LEVEL is set to *k*+1; otherwise, LEVEL is set to 0 (zero).

ii) TYPE is set to a code as shown in Table 7, "Codes used for implementation data types in SQL/CLI", indicating the data type of the column or subordinate descriptor.

iii) Case:

1) If the value of LEVEL is 0 (zero), then:

A) If the resulting column is possibly nullable, then NULLABLE is set to 1 (one); otherwise NULLABLE is set to 0 (zero).

B) If the column name is implementation-dependent, then NAME is set to the implementation-dependent name of the column and UNNAMED is set to 1 (one); otherwise, NAME is set to the <derived column> name for the column and UNNAMED is set to 0 (zero).

C) Case:

I) If a <select list> column *C* is a member of a primary or preferred key of *TBL*, then KEY_MEMBER is set to 1 (one).

II) Otherwise, KEY_MEMBER is set to 0 (zero).

2) Otherwise:

A) NULLABLE is set to 1 (one).

B) Case:

I) If the item descriptor area describes a field of a row type, then

Case:

1) If the name of the field is implementation-dependent, then NAME is set to the implementation-dependent name of the field and UNNAMED is set to 1 (one).

2) Otherwise, NAME is set to the name of the field and UNNAMED is set to 0 (zero).

II) Otherwise, UNNAMED is set to 1 (one) and NAME is set to an implementation-dependent value.

C) KEY_MEMBER is set to 0 (zero).

iv) Case:

1) If TYPE indicates a <character string type>, then LENGTH is set to the length or maximum length in characters of the character string. OCTET_LENGTH is set to the maximum possible length in octets of the character string. If *HL* is C, then the lengths specified in LENGTH and OCTET_LENGTH do not include the implementation-defined null character that terminates a C character string. CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, and CHARACTER_SET_NAME are set to the <character set name> of the character string's character set. COLLATION_CATALOG, COLLATION_SCHEMA, and COLLATION_NAME are set to the <collation name> of the character string's collation.

2) If TYPE indicates a <binary string type>, then LENGTH and OCTET_LENGTH are both set to the length or maximum length in octets of the binary string.

3) If TYPE indicates an <exact numeric type>, then PRECISION and SCALE are set to the precision and scale of the exact numeric.

4) If TYPE indicates an <approximate numeric type>, then PRECISION is set to the precision of the approximate numeric.

5) If TYPE indicates a <datetime type>, then LENGTH is set to the length in positions of the datetime type, DATETIME_INTERVAL_CODE is set to a code as specified in Table 9, “Codes associated with datetime data types in SQL/CLI”, to indicate the specific datetime data type, and PRECISION is set to the <time precision> or <timestamp precision> as applicable.

6) If TYPE indicates INTERVAL, then LENGTH is set to the length in positions of the interval type, DATETIME_INTERVAL_CODE is set to a code as specified in Table 10, “Codes associated with <interval qualifier> in SQL/CLI”, to indicate the specific <interval qualifier>, DATETIME_INTERVAL_PRECISION is set to the <interval leading field precision>, and PRECISION is set to the <interval fractional seconds precision>, if applicable.

7) If TYPE indicates REF, then LENGTH and OCTET_LENGTH are set to the length in octets of the reference type, USER_DEFINED_TYPE_CATALOG, USER_DEFINED_TYPE_SCHEMA, and USER_DEFINED_TYPE_NAME are set to the <user-defined type name> of the <reference type>, and SCOPE_CATALOG, SCOPE_SCHEMA, and SCOPE_NAME are set to the qualified name of the referenceable base table.

8) If TYPE indicates USER-DEFINED TYPE, then USER_DEFINED_TYPE_CATALOG, USER_DEFINED_TYPE_SCHEMA, and USER_DEFINED_TYPE_NAME are set to the <user-defined type name> of the user-defined type. SPECIFIC_TYPE_CATALOG, SPECIFIC_TYPE_SCHEMA, and SPECIFIC_TYPE_NAME are set to the <user-defined type name> of the user-defined type and CURRENT_TRANSFORM_GROUP is set to the CURRENT_TRANSFORM_GROUP_FOR_TYPE for the user-defined type. USER_DEFINED_TYPE_CODE is set to a code as specified in Table 12, “Codes associated with user-defined types in SQL/CLI”, to indicate the category of the user-defined type.

9) If TYPE indicates ROW, then DEGREE is set to the degree of the row type.

10) If TYPE indicates ARRAY, then CARDINALITY is set to the maximum cardinality of the array type.

6) Let *C* be the allocated SQL-connection with which *AS* is associated.

7) If POPULATE IPD for *C* is *False*, then no further rules of this Subclause are applied.

8) If POPULATE IPD for *C* is *True*, then a descriptor for the <dynamic parameter specification>s for the prepared statement is stored in *IPD* as follows:

a) Let *D* be the number of <dynamic parameter specification>s in *S*.

Case:

i) If the value of the statement attribute NEST DESCRIPTOR is *True*, then let NS_i, 1 (one) ≤ *i* ≤ *D*, be the number of subordinate descriptors of the descriptor for the *i*-th input dynamic parameter.

ii) Otherwise, let NS_i, 1 (one) ≤ *i* ≤ *D*, be 0 (zero).

b) TOP_LEVEL_COUNT is set to *D*. If *D* is 0 (zero), then let *TD* be 0 (zero); otherwise, let *TD* be $D + \sum_{i=1}^{D} (NS_i)$. COUNT is set to *TD*.

NOTE 7 — The KEY_TYPE field is not relevant in this case.

c) If *TD* is zero, then no item descriptor areas are set. Otherwise, the first *TD* item descriptor areas are set so that the *i*-th item descriptor area contains a descriptor of the *j*-th <dynamic parameter specification> such that:

i) The descriptor for the first such <dynamic parameter specification> is assigned to the first descriptor area.

ii) The descriptor for the *j*+1-th <dynamic parameter specification> is assigned to the $i+NS_j+1$-th item descriptor area.

iii) If the value of the statement attribute NEST DESCRIPTOR is *True*, then the implicitly ordered subordinate descriptors for the *j*-th <dynamic parameter specification> are assigned to contiguous item descriptor areas starting at the *i*+1-th item descriptor area.

d) The descriptor of a <dynamic parameter specification> consists of values for LEVEL, TYPE, NULLABLE, NAME, UNNAMED, PARAMETER_MODE, PARAMETER_ORDINAL_POSITION, PARAMETER_SPECIFIC_CATALOG, PARAMETER_SPECIFIC_SCHEMA, PARAMETER_SPECIFIC_NAME, and other fields depending on the value of TYPE as described below. Those fields and fields that are not applicable for a particular value of TYPE are set to implementation-dependent values. The DATA_POINTER, INDICATOR_POINTER, OCTET_LENGTH_POINTER, RETURNED_CARDINALITY_POINTER, and KEY_MEMBER fields are not relevant in this case.

i) If the item descriptor area is set to a descriptor that is immediately subordinate to another whose LEVEL value is some value *k*, then LEVEL is set to *k*+1; otherwise, LEVEL is set to 0 (zero).

ii) TYPE is set to a code as shown in Table 7, “Codes used for implementation data types in SQL/CLI”, indicating the data type of the <dynamic parameter specification> or subordinate descriptor.

iii) NULLABLE is set to 1 (one).

NOTE 8 — This indicates that the <dynamic parameter specification> can have the null value.

iv) KEY_MEMBER is set to 0 (zero).

v) UNNAMED is set to 1 (one) and NAME is set to an implementation-dependent value.

vi) Case:

1) If TYPE indicates a <character string type>, then LENGTH is set to the length or maximum length in characters of the character string. OCTET_LENGTH is set to the maximum possible length in octets of the character string. If *HL* is C, then the lengths specified in LENGTH and OCTET_LENGTH do not include the implementation-defined null character that terminates a C character string. CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, and CHARACTER_SET_NAME are set to the <character set name> of the character string's character set. COLLATION_CATALOG, COLLATION_SCHEMA, and COLLATION_NAME are set to the <collation name> of the character string's collation.

2) If TYPE indicates a <binary string type>, then LENGTH and OCTET_LENGTH are both set to the length or maximum length in octets of the binary string.

3) If TYPE indicates an <exact numeric type>, then PRECISION and SCALE are set to the precision and scale of the exact numeric.

4) If TYPE indicates an <approximate numeric type>, then PRECISION is set to the precision of the approximate numeric.

5) If TYPE indicates a <datetime type>, then LENGTH is set to the length in positions of the datetime type, DATETIME_INTERVAL_CODE is set to a code as specified in Table 9, “Codes associated with datetime data types in SQL/CLI”, to indicate the specific datetime data type, and PRECISION is set to the <time precision> or <timestamp precision> as applicable.

6) If TYPE indicates INTERVAL, then LENGTH is set to the length in positions of the interval type, DATETIME_INTERVAL_CODE is set to a code as specified in Table 10, “Codes associated with <interval qualifier> in SQL/CLI”, to indicate the specific <interval qualifier>, DATETIME_INTERVAL_PRECISION is set to the <interval leading field precision>, and PRECISION is set to the <interval fractional seconds precision>, if applicable.

7) If TYPE indicates REF, then LENGTH and OCTET_LENGTH are set to the length in octets of the reference type, USER_DEFINED_TYPE_CATALOG, USER_DEFINED_TYPE_SCHEMA, and USER_DEFINED_TYPE_NAME are set to the <user-defined type name> of the <reference type>, and SCOPE_CATALOG, SCOPE_SCHEMA, and SCOPE_NAME are set to the qualified name of the referenceable base table.

8) If TYPE indicates USER-DEFINED TYPE, then USER_DEFINED_TYPE_CATALOG, USER_DEFINED_TYPE_SCHEMA, and USER_DEFINED_TYPE_NAME are set to the <user-defined type name> of the user-defined type. SPECIFIC_TYPE_CATALOG, SPECIFIC_TYPE_SCHEMA, and SPECIFIC_TYPE_NAME are set to the <user-defined type name> of the user-defined type and CURRENT_TRANSFORM_GROUP is set to the CURRENT_TRANSFORM_GROUP_FOR_TYPE <user-defined type name>.

9) If TYPE indicates ROW, then DEGREE is set to the degree of the row type.

10) If TYPE indicates ARRAY, then CARDINALITY is set to the maximum cardinality of the array type.

9) If LEVEL is 0 (zero) and the prepared statement being described is a <call statement>, then:

a) Let *SR* be the subject routine for the <routine invocation> of the <call statement>.

b) Let D_x be the *x*-th <dynamic parameter specification> simply contained in an SQL argument A_y of the <call statement>.

c) Let P_y be the *y*-th SQL parameter of *SR*.

NOTE 9 — A *P* whose <SQL parameter mode> is IN can be a <value expression> that contains zero, one, or more <dynamic parameter specification>s. Thus:

— Every D_x maps to one and only one P_y.

— Several D_x instances can map to the same P_y.

— There can be P_y instances that have no D_x instances that map to them.

d) The PARAMETER_MODE value in the descriptor for each D_x is set to the value from Table 11, "Codes associated with <parameter mode> in SQL/CLI", that indicates the <SQL parameter mode> of P_y.

e) The PARAMETER_ORDINAL_POSITION value in the descriptor for each D_x is set to the ordinal position of P_y.

f) The PARAMETER_SPECIFIC_CATALOG, PARAMETER_SPECIFIC_SCHEMA, and PARAMETER_SPECIFIC_NAME values in the descriptor for each D_x is set to the values that identify the catalog, schema, and specific name of *SR*.

5.6 Implicit EXECUTE USING and OPEN USING clauses

Function

Specify the rules for an implicit EXECUTE USING clause and an implicit OPEN USING clause.

General Rules

1) Let *T*, *S*, and *AS* be the *TYPE*, *SOURCE*, and *ALLOCATED STATEMENT* specified in the rules of this Subclause.

2) Let *IPD*, *ARD*, and *APD* be the current implementation parameter descriptor, current application row descriptor, and current application parameter descriptor, respectively, for *AS*.

3) Let *C* be the allocated SQL-connection with which *S* is associated.

4) *IPD* and *APD* describe the <dynamic parameter specification>s and <dynamic parameter specification> values, respectively, for the statement being executed. Let *D* be the number of <dynamic parameter specification>s in *S*. Let *NAPD* be the value of COUNT for *APD* and let *NIPD* be the value of COUNT for *IPD*.

   a) If *NAPD* is less than zero, then an exception condition is raised: *dynamic SQL error — invalid descriptor count*.

   b) If *NIPD* is less than zero, then an exception condition is raised: *dynamic SQL error — invalid descriptor count*.

   c) If *NIPD* is less than *D*, then an exception condition is raised: *dynamic SQL error — using clause does not match dynamic parameter specifications*.

   d) Let *NIDAL* be the number of item descriptor areas in *IPD* for which LEVEL is 0 (zero). If *NIDAL* is greater than *D*, then it is implementation-defined whether an exception condition is raised: *dynamic SQL error — using clause does not match dynamic parameter specifications*.

   e) If the first *NIPD* item descriptor areas of *IPD* are not valid as specified in Subclause 5.13, “Description of CLI item descriptor areas”, then an exception condition is raised: *dynamic SQL error — using clause does not match dynamic parameter specifications*.

   f) Let *AD* be the minimum of *NAPD* and *NIPD*.

   g) For each of the first *AD* item descriptor areas of *APD*, if TYPE indicates DEFAULT, then:

      i) Let *TP*, *P*, and *SC* be the values of the TYPE, PRECISION, and SCALE fields, respectively, for the corresponding item descriptor area of *IPD*.

      ii) The data type, precision, and scale of the described <dynamic parameter specification> value (or part thereof, if the item descriptor area is a subordinate descriptor) are set to *TP*, *P*, and *SC*, respectively, for the purposes of this invocation only.

   h) If the first *AD* item descriptor areas of *APD* are not valid as specified in Subclause 5.13, “Description of CLI item descriptor areas”, then an exception condition is raised: *dynamic SQL error — using clause does not match dynamic parameter specifications*.

   i) For the first *AD* item descriptor areas in *APD*:

i) If the number of item descriptor areas in which the value of LEVEL is 0 (zero) is not *D*, then an exception condition is raised: *dynamic SQL error — using clause does not match dynamic parameter specifications*.

ii) If all of the following are true, then an exception condition is raised: *dynamic SQL error — using clause does not match dynamic parameter specifications*.

1) The value of the host variable addressed by INDICATOR POINTER is not negative.

2) At least one of the following is true:

A) TYPE does not indicate ROW and the item descriptor area is not subordinate to an item descriptor area for which the value of the host variable addressed by the INDICATOR POINTER is not negative.

B) TYPE indicates ARRAY or ARRAY LOCATOR.

C) TYPE indicates MULTISET or MULTISET LOCATOR.

3) The value of the host variable addressed by DATA_POINTER is not a valid value of the data type represented by the item descriptor area.

j) For each of the first *AD* item descriptor areas *ADIDA* in *APD*:

i) If the OCTET_LENGTH_POINTER field of *ADIDA* has the same non-zero value as the INDICATOR_POINTER field of *IDA*, then *SHARE* is true for *ADIDA*; otherwise, *SHARE* is false for *ADIDA*.

Case:

1) If *SHARE* is true for *ADIDA* and the value of the commonly addressed host variable is the appropriate 'Code' for SQL NULL DATA in Table 27, “Miscellaneous codes used in CLI”, then *NULL* is true for *ADIDA*.

2) If *SHARE* is false for *ADIDA*, INDICATOR_POINTER is not zero, and the value of the host variable addressed by INDICATOR_POINTER is the appropriate 'Code' for SQL NULL DATA in Table 27, “Miscellaneous codes used in CLI”, then *NULL* is true for *ADIDA*.

3) Otherwise, *NULL* is false for *ADIDA*.

ii) If *NULL* is false for *ADIDA*, OCTET_LENGTH_POINTER is not 0 (zero), and the value of the host variable addressed by OCTET_LENGTH_POINTER is the appropriate 'Code' for SQL NULL DATA in Table 27, “Miscellaneous codes used in CLI”, then *DEFERRED* is true for *ADIDA*; otherwise, *DEFERRED* is false for *ADIDA*.

k) If all of the following are true for any item descriptor area in the first *AD* item descriptor areas of *APD*, then an exception condition is raised: *dynamic SQL error — using clause does not match dynamic parameter specifications*.

i) *DEFERRED* is true for the item descriptor area.

ii) Either of the following is true:

1) The value of LEVEL is zero and TYPE indicates ROW, ARRAY, or MULTISET.

2) LEVEL is greater than 0 (zero).

NOTE 10 — This rule states that a parameter whose type is ROW, ARRAY, or MULTISET shall be bound; it cannot be a deferred parameter.

l) For each item descriptor area whose LEVEL is 0 (zero) and for each of its subordinate descriptor areas, if any, for which *DEFERRED* is false in the first *AD* item descriptor areas of *APD* and whose corresponding <dynamic parameter specification> has a <parameter mode> of PARAM MODE IN or PARAM MODE INOUT, refer to the corresponding <dynamic parameter specification> value as an *immediate parameter value* and refer to the corresponding <dynamic parameter specification> as an *immediate parameter*.

m) Let *IDA* be the *i*-th item descriptor area of *APD* whose LEVEL value is 0 (zero). Let *SDT* be the data type represented by *IDA*. The *associated value* of *IDA*, denoted by *SV*, is defined as follows.

   Case:

   i) If *NULL* is true for *IDA*, then *SV* is the null value.

   ii) If TYPE indicates ROW, then *SV* is a row whose type is *SDT* and whose field values are the associated values of the immediately subordinate descriptor areas of *IDA*.

   iii) Otherwise:

      1) Let *V* be the value of the host variable addressed by DATA_POINTER.

      2) Case:

         A) If TYPE indicates CHARACTER, then

            Case:

            I) If OCTET_LENGTH_POINTER is zero or if OCTET_LENGTH_POINTER is not zero and the value of the host variable addressed by OCTET_LENGTH_POINTER indicates NULL TERMINATED, then let *L* be the number of characters of *V* that precede the implementation-defined null character that terminates a C character string.

            II) Otherwise, let *Q* be the value of the host variable addressed by OCTET_LENGTH_POINTER and let *L* be the number of characters wholly contained in the first *Q* octets of *V*.

         B) Otherwise, let *L* be zero.

      3) Let *SV* be *V* with effective data type *SDT*, as represented by the length value *L* and by the values of the TYPE, PRECISION, and SCALE fields.

n) Let *TDT* be the effective data type of the *i*-th immediate parameter as represented by the values of the TYPE, LENGTH, PRECISION, SCALE, DATETIME_INTERVAL_CODE, DATETIME_INTERVAL_PRECISION, CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, CHARACTER_SET_NAME, USER_DEFINED_TYPE_CATALOG, USER_DEFINED_TYPE_SCHEMA, USER_DEFINED_TYPE_NAME, SCOPE_CATALOG, SCOPE_SCHEMA, and SCOPE_NAME fields in the *i*-th item descriptor area of *IPD* for which the LEVEL value is 0 (zero), and all its subordinate descriptor areas.

o) Let *SDT* be the effective data type of the *i*-th bound parameter as represented by the values of the TYPE, LENGTH, PRECISION, SCALE, DATETIME_INTERVAL_CODE, DATETIME_INTERVAL_PRECISION, CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, CHARACTER_SET_NAME, USER_DEFINED_TYPE_CATALOG, USER_DEFINED_TYPE_SCHEMA,

USER_DEFINED_TYPE_NAME, SCOPE_CATALOG, SCOPE_SCHEMA, and SCOPE_NAME fields in the corresponding item descriptor area of *APD* for which the LEVEL is 0 (zero), and all its subordinate descriptor areas.

p) Case:

i) If *SDT* is a locator type, then let *TV* be the value *SV*.

ii) If *SDT* and *TDT* are predefined types, then:

1) Case:

A) If the <cast specification>

```
CAST ( SV AS TDT )
```

does not conform to the Syntax Rules of Subclause 6.12, "<cast specification>", in ISO/IEC 9075-2, and there is an implementation-defined conversion from type *SDT* to type *TDT*, then that implementation-defined conversion is effectively performed, converting *SV* to type *TDT*, and the result is the value *TV* of the *i*-th bound target.

B) Otherwise:

I) If the <cast specification>

```
CAST ( SV AS TDT )
```

does not conform to the Syntax Rules of Subclause 6.12, "<cast specification>", in ISO/IEC 9075-2, then an exception condition is raised: *dynamic SQL error — restricted data type attribute violation*.

II) The <cast specification>

```
CAST ( SV AS TDT )
```

is effectively performed and the result is the value *TV* of the *i*-th bound target.

2) Let *UDT* be the effective data type of the actual *i*-th immediate parameter, defined to be the data type represented by the values of the TYPE, LENGTH, PRECISION, SCALE, DATETIME_INTERVAL_CODE, DATETIME_INTERVAL_PRECISION, CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, CHARACTER_SET_NAME, USER_DEFINED_TYPE_CATALOG, USER_DEFINED_TYPE_SCHEMA, USER_DEFINED_TYPE_NAME, SCOPE_CATALOG, SCOPE_SCHEMA, and SCOPE_NAME fields that would automatically be set in the corresponding item descriptor area of *IPD* if POPULATE IPD was True for *C*.

3) Case:

A) If the <cast specification>

```
CAST ( TV AS UDT )
```

does not conform to the Syntax Rules of Subclause 6.12, "<cast specification>", in ISO/IEC 9075-2, and there is an implementation-defined conversion from type *SDT* to type *UDT*, then that implementation-defined conversion is effectively performed, converting *SV* to type *UDT* and the result is the value *TV* of the *i*-th immediate parameter.

B) Otherwise:

I) If the <cast specification>

```
CAST ( TV AS UDT )
```

does not conform to the Syntax Rules of Subclause 6.12, "<cast specification>", in ISO/IEC 9075-2, then an exception condition is raised: *dynamic SQL error — restricted data type attribute violation*.

II) The <cast specification>

```
CAST ( TV AS UDT )
```

is effectively performed and the result is the value of the *i*-th immediate parameter.

iii) If *SDT* is a predefined type and *TDT* is a user-defined type, then:

1) Let *DT* be the data type identified by *TDT*.

2) If the current SQL-session has a group name corresponding to the user-defined name of *DT*, then let *GN* be that group name; otherwise, let *GN* be the default transform group name associated with the current SQL-session.

3) The Syntax Rules of Subclause 9.21, "Determination of a to-sql function", in ISO/IEC 9075-2, are applied with DT as *TYPE* and GN as *GROUP*.

Case:

A) If there is an applicable to-sql function, then let *TSF* be that to-sql function. If *TSF* is an SQL-invoked method, then let *TSFPT* be the declared type of the second SQL parameter of *TSF*; otherwise, let *TSFPT* be the declared type of the first SQL parameter of *TSF*.

Case:

I) If *TSFPT* is compatible with *SDT*, then

Case:

1) If *TSF* is an SQL-invoked method, then *TSF* is effectively invoked with the value returned by the function invocation:

```
DT()
```

as the first parameter and *SV* as the second parameter. The result of evaluating the expression *TSF*(*DT*(), *SV*) is the value of the *i*-th immediate parameter.

2) Otherwise, *TSF* is effectively invoked with *SV* as the first parameter. The result of evaluating the expression *TSF*(*SV*) is the value of the *i*-th immediate parameter.

II) Otherwise, an exception condition is raised: *dynamic SQL error — restricted data type attribute violation*.

B) Otherwise, an exception condition is raised: *dynamic SQL error — data type transform function violation*.

q) If *DEFERRED* is true for at least one of the first *AD* item descriptor areas of *APD*, then:

   i) Let *PN* be the parameter number associated with the first such item descriptor area.

   ii) *PN* becomes the deferred parameter number associated with *AS*.

   iii) If *T* is 'EXECUTE', then *S* becomes the statement source associated with *AS*.

   iv) An exception condition is raised: *CLI-specific condition — dynamic parameter value needed*.

5.7 Implicit CALL USING clause

Function

Specify the rules for an implicit CALL USING clause.

General Rules

1) Let *S* and *AS* be a *SOURCE* and an *ALLOCATED STATEMENT* specified in the rules of this Subclause.

2) Let *IPD* and *APD* be the current implementation parameter descriptor and current application row descriptor, respectively, for *AS*.

3) *IPD* and *APD* describe the <dynamic parameter specification>s and <dynamic parameter specification> values, respectively, for the <call statement> being executed. Let *D* be the number of <dynamic parameter specification>s in *S*.

   a) Let *AD* be the value of the COUNT field of *APD*. If *AD* is less than zero, then an exception condition is raised: *dynamic SQL error — invalid descriptor count*.

   b) For each item descriptor area in the *APD* whose LEVEL is 0 (zero) in the first *AD* item descriptor areas of *APD*, and for all of their subordinate descriptor areas, refer to a <dynamic parameter specification> value whose corresponding item descriptor areas have a non-zero DATA_POINTER value and whose corresponding <dynamic parameter specification> has a <parameter mode> of PARAM MODE OUT or PARAM MODE INOUT as a *bound target* and refer to the corresponding <dynamic parameter specification> as a *bound parameter*.

   c) If any item descriptor area corresponding to a bound target in the first *AD* item descriptor areas of *APD* is not valid as specified in Subclause 5.13, “Description of CLI item descriptor areas”, then an exception condition is raised: *dynamic SQL error — using clause does not match target specifications*.

   d) Let *SDT* be the effective data type of the *i*-th bound parameter as represented by the values of the TYPE, LENGTH, PRECISION, SCALE, DATETIME_INTERVAL_CODE, DATETIME_INTERVAL_PRECISION, CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, CHARACTER_SET_NAME, USER_DEFINED_TYPE_CATALOG, USER_DEFINED_TYPE_SCHEMA, USER_DEFINED_TYPE_NAME, SCOPE_CATALOG, SCOPE_SCHEMA, and SCOPE_NAME fields in the *i*-th item descriptor area of *IPD* for which the LEVEL is 0 (zero) and all of its subordinate descriptor areas. Let *SV* be the value of the output parameter, with data type *SDT*.

   e) If TYPE indicates USER-DEFINED TYPE, then let the most specific type of the *i*-th bound parameter whose value is *SV* be represented by the values of the SPECIFIC_TYPE_CATALOG, SPECIFIC_TYPE_SCHEMA, and SPECIFIC_TYPE_NAME fields in the corresponding item descriptor area of *IPD*.

   f) Let *TYPE*, *OL*, *DP*, *IP*, and *LP* be the values of the TYPE, OCTET_LENGTH, DATA_POINTER, INDICATOR_POINTER, and OCTET_LENGTH_POINTER fields, respectively, in the item descriptor area of *APD* corresponding to the *i*-th bound target (or part thereof, if the item descriptor area is a subordinate descriptor).

   g) Case:

      i) If *TYPE* indicates CHARACTER, then:

1) Let *UT* be the code value corresponding to CHARACTER VARYING as specified in Table 7, “Codes used for implementation data types in SQL/CLI”.

2) Let *LV* be the implementation-defined maximum length for a CHARACTER VARYING data type.

ii) Otherwise, let *UT* be *TYPE* and let *LV* be 0 (zero).

h) Let *TDT* be the effective data type of the *i*-th bound target as represented by the type *UT*, the length value *LV*, and the values of the PRECISION, SCALE, CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, CHARACTER_SET_NAME, USER_DEFINED_TYPE_ CATALOG, USER_DEFINED_TYPE_SCHEMA, USER_DEFINED_TYPE_NAME, SCOPE_CATALOG, SCOPE_SCHEMA, and SCOPE_NAME fields in the corresponding item descriptor area of *APD* for which the LEVEL is 0 (zero) and all its subordinate descriptor areas.

i) Case:

i) If *TDT* is a locator type, then

Case:

1) If *SV* is not the null value, then a locator *L* that uniquely identifies *SV* is generated and the value *TV* of the *i*-th bound target is set to an implementation-dependent four-octet value that represents *L*.

2) Otherwise, the value *TV* of the *i*-th bound target is the null value.

ii) If *SDT* and *TDT* are predefined types, then

Case:

1) If the <cast specification>

```
CAST ( SV AS TDT )
```

does not conform to the Syntax Rules of Subclause 6.12, “<cast specification>”, in ISO/IEC 9075-2, and there is an implementation-defined conversion from type *SDT* to type *TDT*, then that implementation-defined conversion is effectively performed, converting *SV* to type *TDT*, and the result is the value *TV* of the *i*-th bound target.

2) Otherwise:

A) If the <cast specification>

```
CAST ( SV AS TDT )
```

does not conform to the Syntax Rules of Subclause 6.12, “<cast specification>”, in ISO/IEC 9075-2, then an exception condition is raised: *dynamic SQL error — restricted data type attribute violation*.

B) The <cast specification>

```
CAST ( SV AS TDT )
```

is effectively performed and the result is the value *TV* of the *i*-th bound target.

iii) If *SDT* is a user-defined type and *TDT* is a predefined data type, then:

1) Let *DT* be the data type identified by *SDT*.

2) If the current SQL-session has a group name corresponding to the user-defined name of *DT*, then let *GN* be that group name; otherwise, let *GN* be the default transform group name associated with the current SQL-session.

3) The Syntax Rules of Subclause 9.19, “Determination of a from-sql function”, in ISO/IEC 9075-2, are applied with DT as *TYPE* and GN as *GROUP*.

Case:

A) If there is an applicable from-sql function, then let *FSF* be that from-sql function and let *FSFRT* be the <returns data type> of *FSF*.

Case:

I) If *FSFRT* is compatible with *TDT*, then the from-sql function *TSF* is effectively invoked with *SV* as its input parameter and the result of evaluating *TSF*(*SV*) is the value *TV* of the *i*-th bound target.

II) Otherwise, an exception condition is raised: *dynamic SQL error — restricted data type attribute violation*.

B) Otherwise, an exception condition is raised: *dynamic SQL error — data type transform function violation*.

j) Let *IDA* be the top-level item descriptor area corresponding to the *i*-th output parameter.

k) Case:

i) If TYPE indicates ROW, then

Case:

1) If *TV* is the null value, then

Case:

A) If *IP* is a null pointer for *IDA* or for any of the subordinate descriptor areas of *IDA* that are not subordinate to an item descriptor area whose type indicates ARRAY, ARRAY LOCATOR, MULTISET, or MULTISET LOCATOR, then an exception condition is raised: *data exception — null value, no indicator parameter*.

B) Otherwise, the value of the host variable addressed by *IP* for *IDA*, and those in all subordinate descriptor areas of *IDA* that are not subordinate to an item descriptor area whose TYPE indicates ARRAY, ARRAY LOCATOR, MULTISET, or MULTISET LOCATOR are set to the appropriate 'Code' for SQL NULL DATA in Table 27, “Miscellaneous codes used in CLI”, and the values of variables addressed by *DP* and *LP* are implementation-dependent.

2) Otherwise, the *i*-th subordinate descriptor area of *IDA* is set to reflect the value of the *i*-th field of *TV* by applying General Rule 3)k) to the *i*-th subordinate descriptor area of *IDA* as *IDA*, the value of *i*-th field of *TV* as *TV*, the value of the *i*-th field of *SV* as *SV*, and the data type of the *i*-th field of *SV* as *SDT*.

ii) Otherwise,

Case:

1) If *TV* is the null value, then

   Case:

   A) If *IP* is a null pointer, then an exception condition is raised: *data exception — null value, no indicator parameter*.

   B) Otherwise, the value of the host variable addressed by *IP* is set to the appropriate 'Code' for SQL NULL DATA in Table 27, “Miscellaneous codes used in CLI”, and the values of the host variables addressed by *DP* and *LP* are implementation-dependent.

2) Otherwise:

   A) If *IP* is not a null pointer, then the value of the host variable addressed by *IP* is set to 0 (zero).

   B) Case:

      I) If *TYPE* indicates CHARACTER or CHARACTER LARGE OBJECT, then:

         1) If *TV* is a zero-length character string, then it is implementation-defined whether or not an exception condition is raised: *data exception — zero-length character string*.

         2) The General Rules of Subclause 5.9, “Character string retrieval”, are applied with *DP*, *TV*, *OL*, and *LP* as *TARGET*, *VALUE*, *TARGET OCTET LENGTH*, and *RETURNED OCTET LENGTH*, respectively.

      II) If *TYPE* indicates BINARY LARGE OBJECT, then the General Rules of Subclause 5.10, “Binary string retrieval”, are applied with *DP*, *TV*, *OL*, and *LP* as *TARGET*, *VALUE*, *TARGET OCTET LENGTH*, and *RETURNED OCTET LENGTH*, respectively.

      III) If *TYPE* indicates ARRAY, ARRAY LOCATOR, MULTISET, or MULTISET LOCATOR and if RETURNED_CARDINALITY_POINTER is not 0 (zero), then the value of the host variable addressed by RETURNED_CARDINALITY_POINTER is set to the cardinality of *TV*.

      IV) Otherwise, the value of the host variable addressed by *DP* is set to *TV*.

5.8 Implicit FETCH USING clause

Function

Specify the rules for an implicit FETCH USING clause.

General Rules

1) Let *S*, *RS*, *RP*, and *AS* be a *SOURCE*, *ROWS*, *ROWS PROCESSED*, and an *ALLOCATED STATEMENT* specified in the rules of this Subclause.

2) Let *IRD* and *ARD* be the current implementation row descriptor and current application row descriptor, respectively, associated with *AS*.

3) *IRD* and *ARD* describe the <select list> columns and <target specification>s, respectively, for the column values that are to be retrieved. Let *D* be the degree of the table defined by *S*.

   a) Let *AD* be the value of the COUNT field of *ARD*. If *AD* is less than zero, then an exception condition is raised: *dynamic SQL error — invalid descriptor count*.

   b) For each item descriptor area in *ARD* whose LEVEL is 0 (zero) in the first *AD* item descriptor areas of *ARD*, and for all of their subordinate descriptor areas, refer to a <target specification> whose corresponding item descriptor areas have a non-zero DATA_POINTER as a *bound target* and refer to the corresponding <select list> column as a *bound column*.

   c) If any item descriptor area corresponding to a bound target in the first *AD* item descriptor areas of *ARD* is not valid as specified in Subclause 5.13, "Description of CLI item descriptor areas", then an exception condition is raised: *dynamic SQL error — using clause does not match target specifications*.

   d) Let *SDT* be the effective data type of the *i*-th bound column as represented by the values of the TYPE, LENGTH, PRECISION, SCALE, DATETIME_INTERVAL_CODE, DATETIME_INTERVAL_PRECISION, CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, CHARACTER_SET_NAME, USER_DEFINED_TYPE_CATALOG, USER_DEFINED_TYPE_SCHEMA, USER_DEFINED_TYPE_NAME, SCOPE_CATALOG, SCOPE_SCHEMA, and SCOPE_NAME fields in the *i*-th item descriptor area of *IRD* whose LEVEL is 0 (zero) and all of its subordinate descriptor areas.

   e) If TYPE indicates USER-DEFINED TYPE, then let the most specific type of the *i*-th bound column whose value is *SV* be represented by the values of the SPECIFIC_TYPE_CATALOG, SPECIFIC_TYPE_SCHEMA, and SPECIFIC_TYPE_NAME fields in the corresponding item descriptor area of *IRD*.

   f) Let *TYPE*, *OL*, *DP*, *IP*, and *LP* be the values of the TYPE, OCTET_LENGTH, DATA_POINTER, INDICATOR_POINTER, and OCTET_LENGTH_POINTER fields, respectively, in the item descriptor area of *ARD* corresponding to the *i*-th bound target (or part thereof, if the item descriptor area is a subordinate descriptor).

   g) Let *ASP* be the value of the ARRAY_STATUS_POINTER field in *IRD*.

   h) For *RN* ranging from 1 (one) through *RS*, if the *RN*-th row of the rowset has been fetched, then:

      i) Let *SV* be the value of the <select list> column, with data type *SDT*.

ii) Let *DPE*, *IPE*, and *LPE* be the addresses of the *RN*-th element of the arrays addressed by *DP*, *IP*, and *LP*, respectively.

iii) Case:

1) If *TYPE* indicates CHARACTER, then:

A) Let *UT* be the code value corresponding to CHARACTER VARYING as specified in Table 7, "Codes used for implementation data types in SQL/CLI".

B) Let *LV* be the implementation-defined maximum length for a CHARACTER VARYING data type.

2) Otherwise, let *UT* be *TYPE* and let *LV* be 0 (zero).

iv) Let *TDT* be the effective data type of the *i*-th bound target as represented by the type *UT*, the length value *LV*, and the values of the PRECISION, SCALE, CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, CHARACTER_SET_NAME, USER_DEFINED_TYPE_CATALOG, USER_DEFINED_TYPE_SCHEMA, USER_DEFINED_TYPE_NAME, SCOPE_CATALOG, SCOPE_SCHEMA, and SCOPE_NAME fields in the item descriptor area of *ARD* whose LEVEL is 0 (zero) and all of its subordinate descriptor areas.

v) Let *LTDT* be the data type on the last fetch of the *i*-th bound target, if any. If any of the following is true, then is implementation-defined whether or not an exception condition is raised: *dynamic SQL error — restricted data type attribute violation*.

1) *LTDT* and *TDT* both identify a binary large object type and only one of *LTDT* and *TDT* is a binary large object locator.

2) *LTDT* and *TDT* both identify a character large object type and only one of *LTDT* and *TDT* is a character large object locator.

3) *LTDT* and *TDT* both identify an array type and only one of *LTDT* and *TDT* is an array locator.

4) *LTDT* and *TDT* both identify a multiset type and only one of *LTDT* and *TDT* is a multiset locator.

5) *LTDT* and *TDT* both identify a user-defined type and only one of *LTDT* and *TDT* is a user-defined type locator.

vi) Case:

1) If *TDT* is a locator type, then;

A) If *SV* is not the null value, then a locator *L* that uniquely identifies *SV* is generated and the value *TV* of the *i*-th bound target is set to an implementation-dependent four-octet value that represents *L*.

B) Otherwise, the value *TV* of the *i*-th bound target is the null value.

2) If *SDT* and *TDT* are predefined types, then

Case:

A) If the <cast specification>

```
CAST ( SV AS TDT )
```

does not conform to the Syntax Rules of Subclause 6.12, “<cast specification>”, in ISO/IEC 9075-2, and there is an implementation-defined conversion from type *SDT* to type *TDT*, then that implementation-defined conversion is effectively performed, converting *SV* to type *TDT*, and the result is the value *TV* of the *i*-th bound target.

B) Otherwise:

I) If the <cast specification>

```
CAST ( SV AS TDT )
```

does not conform to the Syntax Rules of Subclause 6.12, “<cast specification>”, in ISO/IEC 9075-2, then an exception condition is raised: *dynamic SQL error — restricted data type attribute violation*.

II) The <cast specification>

```
CAST ( SV AS TDT )
```

is effectively performed and the result is the value *TV* of the *i*-th bound target.

For every status record that results from the application of this Rule, the ROW_NUMBER field is set to *RN* and the COLUMN_NUMBER field is set to *i*. If *ASP* is not a null pointer, then the *RN*-th element of the array addressed by *ASP* is set to:

1) If there were completion conditions: *warning* raised during the application of this Rule, then 6 (indicating **Row success with information**).

2) If there were exception conditions raised during the application of this Rule, then 5 (indicating **Row error**).

III) The <cast specification>

```
CAST ( SV AS TDT )
```

is effectively performed and the result is the value *TV* of the *i*-th bound target.

3) If *SDT* is a user-defined type and *TDT* is a predefined data type, then:

A) Let *DT* be the data type identified by *SDT*.

B) If the current SQL-session has a group name corresponding to the user-defined name of *DT*, then let *GN* be that group name; otherwise, let *GN* be the default transform group name associated with the current SQL-session.

C) The Syntax Rules of Subclause 9.19, “Determination of a from-sql function”, in ISO/IEC 9075-2, are applied with *DT* and *GN* as *TYPE* and *GROUP*, respectively.

Case:

I) If there is an applicable from-sql function, then let *FSF* be that from-sql function and let *FSFRT* be the <returns data type> of *FSF*.

Case:

1) If *FSFRT* is compatible with *TDT*, then the from-sql function *TSF* is effectively invoked with *SV* as its input parameter and the the result of evaluating *TSF*(*SV*) is the value *TV* of the *i*-th bound target.

2) Otherwise, an exception condition is raised: *dynamic SQL error — restricted data type attribute violation*.

II) Otherwise, an exception condition is raised: *dynamic SQL error — data type transform function violation*.

vii) Let *IDA* be the top-level item descriptor area corresponding to the *i*-th bound column.

viii) Case:

1) If TYPE indicates ROW, then

Case:

A) If *TV* is the null value, then

Case:

I) If *IPE* is a null pointer for *IDA* or for any of the subordinate descriptor areas of *IDA* that are not subordinate to an item descriptor area whose type indicates ARRAY, ARRAY LOCATOR, MULTISET, or MULTISET LOCATOR, then an exception condition is raised: *data exception — null value, no indicator parameter*.

II) Otherwise, the value of the host variable addressed by *IPE* for *IDA*, and that in all subordinate descriptor areas of *IDA* that are not subordinate to an item descriptor area whose TYPE indicates ARRAY, ARRAY LOCATOR, MULTISET, or MULTISET LOCATOR, is set to the appropriate 'Code' for SQL NULL DATA in Table 27, “Miscellaneous codes used in CLI”, and the values of variables addressed by *DPE* and *LPE* are implementation-dependent.

B) Otherwise, the *i*-th subordinate descriptor area of *IDA* is set to reflect the value of the *i*-th field of *TV* by applying General Rule 3)h)viii) to the *i*-th subordinate descriptor area of *IDA* as *IDA*, the value of *i*-th field of *TV* as *TV*, the value of the *i*-th field of *SV* as *SV*, and the data type of the *i*-th field of *SV* as *SDT*.

2) Otherwise,

Case:

A) If *TV* is the null value, then

Case:

I) If *IPE* is a null pointer, then an exception condition is raised: *data exception — null value, no indicator parameter*.

II) Otherwise, the value of the host variable addressed by *IPE* is set to the appropriate 'Code' for SQL NULL DATA in Table 27, “Miscellaneous codes used in CLI”, and the values of the host variables addressed by *DPE* and *LPE* are implementation-dependent.

B) Otherwise:

I) If *IPE* is not a null pointer, then the value of the host variable addressed by *IPE* is set to 0 (zero).

II) Case:

1) If *TYPE* indicates CHARACTER or CHARACTER LARGE OBJECT, then:

a) If *TV* is a zero-length character string, then it is implementation-defined whether or not an exception condition is raised: *data exception — zero-length character string*.

b) The General Rules of Subclause 5.9, “Character string retrieval”, are applied with *DPE*, *TV*, *OL*, and *LPE* as *TARGET*, *VALUE*, *TARGET OCTET LENGTH*, and *RETURNED OCTET LENGTH*, respectively.

2) For every status record that results from the application of the preceding Rule, the ROW_NUMBER field is set to *RN* and the COLUMN_NUMBER field is set to *i*. If *ASP* is not a null pointer, then the *RN*-th element of the array addressed by *ASP* is set to:

a) If there were completion conditions: *warning* raised during the application of the preceding Rule, then 6 (indicating **Row success with information**).

b) If there were exception conditions raised during the application of the preceding Rule, then 5 (indicating **Row error**).

3) If *TYPE* indicates BINARY, BINARY VARYING, or BINARY LARGE OBJECT, then the General Rules of Subclause 5.10, “Binary string retrieval”, are applied with *DPE*, *TV*, *OL*, and *LPE* as *TARGET*, *VALUE*, *TARGET OCTET LENGTH*, and *RETURNED OCTET LENGTH*, respectively.

For every status record that results from the application of this Rule, the ROW_NUMBER field is set to *RN* and the COLUMN_NUMBER field is set to *i*. If *ASP* is not a null pointer, then the *RN*-th element of the array addressed by *ASP* is set to:

a) If there were completion conditions: *warning* raised during the application of this Rule, then 6 (indicating **Row success with information**).

b) If there were exception conditions raised during the application of this Rule, then 5 (indicating **Row error**).

4) If *TYPE* indicates ARRAY, ARRAY LOCATOR, MULTISET, or MULTISET LOCATOR, and if RETURNED_CARDINALITY_POINTER is not a null pointer, then the value of the host variable addressed by RETURNED_CARDINALITY_POINTER is set to the cardinality of *TV*.

5) Otherwise, the value of the host variable addressed by *DPE* is set to *TV* and the value of the host variable addressed by *LPE* is implementation-dependent.

3) If there were no exception conditions raised during the application of this Rule, then:

A) Increment *RP* by 1 (one).

B) If *ASP* is not a null pointer, then set the *RN*-th element of the array pointed to by *ASP* to 0 (zero, indicating **Row success**).

5.9 Character string retrieval

Function

Specify the rules for retrieving character string values.

General Rules

1) Let *T*, *V*, *TL*, and *RL* be a *TARGET*, *VALUE*, *TARGET OCTET LENGTH*, and *RETURNED OCTET LENGTH* specified in an application of this Subclause.

2) If *TL* is not greater than zero, then an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

3) Let *L* be the length in octets of *V*.

4) If *RL* is not a null pointer, then the value of the host variable addressed by *RL* is set to *L*.

5) Case:

   a) If null termination is *False* for the current SQL-environment, then

      Case:

      i) If *L* is not greater than *TL*, then the first *L* octets of *T* are set to *V* and the values of the remaining octets of *T* are implementation-dependent.

      ii) Otherwise, *T* is set to the first *TL* octets of *V* and a completion condition is raised: *warning — string data, right truncation*.

   b) Otherwise, let *NB* be the length in octets of a null terminator in the character set of *T*.

      Case:

      i) If *L* is not greater than (*TL*–*NB*), then the first (*L*+*NB*) octets of *T* are set to *V* concatenated with a single implementation-defined null character that terminates a C character string. The values of the remaining characters of *T* are implementation-dependent.

      ii) Otherwise, *T* is set to the first (*TL*–*NB*) octets of *V* concatenated with a single implementation-defined null character that terminates a C character string and a completion condition is raised: *warning — string data, right truncation*.

5.10 Binary string retrieval

Function

Specify the rules for retrieving binary string values.

General Rules

1) Let *T*, *V*, *TL*, and *RL* be a *TARGET*, *VALUE*, *TARGET OCTET LENGTH*, and *RETURNED OCTET LENGTH* specified in an application of this Subclause.
2) If *TL* is not greater than zero (0), then an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.
3) Let *L* be the length in octets of *V*.
4) If *RL* is not a null pointer, then *RL* is set to *L*.
5) Case:
   a) If *L* is not greater than *TL*, then the first *L* octets of *T* are set to *V* and the values of the remaining octets of *T* are implementation-dependent.
   b) Otherwise, *T* is set to the first *TL* octets of *V* and a completion condition is raised: *warning — string data, right truncation*.

5.11 Deferred parameter check

Function

Check for the existence of deferred dynamic parameters when accessing a CLI descriptor.

General Rules

1) Let *DA* be a DESCRIPTOR AREA specified in an application of this Subclause.

2) Let *C* be the allocated SQL-connection with which *DA* is associated.

3) Let *L1* be the set of all allocated SQL-statements associated with *C*.

4) Let *L2* be the set of all allocated SQL-statements in *L1* which have an associated deferred parameter number.

5) Let *L3* be the set of all CLI descriptor areas that are either the current application parameter descriptor for, or the implementation parameter descriptor associated with, an allocated SQL-statement in *L2*.

6) If *DA* is contained in *L3*, then an exception condition is raised: *CLI-specific condition — function sequence error*.

5.12 CLI-specific status codes

Some of the conditions that can occur during the execution of CLI routines are CLI-specific. The corresponding status codes are listed in Table 5, “SQLSTATE class and subclass values for SQL/CLI-specific conditions”.

Table 5 — SQLSTATE class and subclass values for SQL/CLI-specific conditions

Category	Condition	Class	Subcondition	Subclass
X	CLI-specific condition	HY	(*no subclass*)	000
			associated statement is not prepared	007
			attempt to concatenate a null value	020
			attribute cannot be set now	011
			column type out of range	097
			dynamic parameter value needed	*(See the Note at the end of the table)*
			function sequence error	010
			inconsistent descriptor information	021
			invalid attribute identifier	092
			invalid attribute value	024
			invalid cursor position	109
			invalid data type	004
			invalid data type in application descriptor	003
			invalid descriptor field identifier	091
			invalid fetch orientation	106
			invalid FunctionId specified	095
			invalid handle	*(See the Note at the end of the table)*

Category	Condition	Class	Subcondition	Subclass
			invalid information type	096
			invalid LengthPrecision value	104
			invalid parameter mode	105
			invalid retrieval code	103
			invalid string length or buffer length	090
			invalid transaction operation code	012
			invalid use of automatically-allocated descriptor handle	017
			invalid use of null pointer	009
			limit on number of handles exceeded	014
			memory allocation error	001
			memory management error	013
			non-string data cannot be sent in pieces	019
			non-string data cannot be used with string routine	055
			nullable type out of range	099
			operation canceled	008
			optional feature not implemented	C00
			row value out of range	107
			scope out of range	098
			server declined the cancellation request	018

NOTE 11 — No subclass value is defined for the subcondition *invalid handle* since no diagnostic information can be generated in this case, nor for the subcondition *dynamic parameter value needed*, since no diagnostic information is generated in this case.

5.13 Description of CLI item descriptor areas

This Subclause is modified by Subclause 19.3, "Description of CLI item descriptor areas", in ISO/IEC 9075-9.

Function

Specify the identifiers, data types and codes for fields used in CLI item descriptor areas.

Syntax Rules

1) A CLI item descriptor area comprises the fields specified in Table 6, "Fields in SQL/CLI row and parameter descriptor areas".

2) Given a CLI item descriptor area *IDA* in which the value of LEVEL is some value *N*, the *immediately subordinate* descriptor areas of *IDA* are those CLI item descriptor areas in which the value of LEVEL is *N*+1 and whose position in the CLI descriptor area follows that of *IDA* and precedes that of any CLI item descriptor area in which the value of LEVEL is less than *N*+1. The subordinate descriptor areas of *IDA* are those CLI item descriptor areas that are immediately subordinate descriptor areas of *IDA* or that are subordinate descriptor areas of an CLI item descriptor area that is immediately subordinate to *IDA*.

3) Given a data type *DT* and its descriptor *DE*, the immediately subordinate descriptors of *DE* are defined to be

   Case:

   a) If *DT* is ROW, then the field descriptors of the fields of *DT*. The *i*-th immediately subordinate descriptor is the descriptor of the *i*-th field of *DT*.

   b) If *DT* is ARRAY or MULTISET, then the descriptor of the associated element type of *DT*. The subordinate descriptors of *DE* are those descriptors that are immediately subordinate descriptors of *DE* or that are subordinate descriptors of a descriptor that is immediately subordinate to *DE*.

4) Given a descriptor *DE*, let SDE_j represent its *j*-th immediately subordinate descriptor. There is an implied ordering of the subordinate descriptors of *DE*, such that:

   a) SDE_1 is in the first ordinal position.

   b) The ordinal position of SDE_{j+1} is $K+NS+1$, where *K* is the ordinal position of SDE_j and *NS* is the number of subordinate descriptors of SDE_j. The implicitly ordered subordinate descriptors of SDE_j occupy contiguous ordinal positions starting at position $K+1$.

5) Let *IDA* be an item descriptor area in an implementation parameter descriptor. *IDA* is *valid* if and only if all of the following are true:

   a) TYPE is one of the code values in Table 7, "Codes used for implementation data types in SQL/CLI".

   b) If LEVEL is 0 (zero) for *IDA*, then let *TLC* be the value of TOP_LEVEL_COUNT of the implementation parameter descriptor associated with *IDA*. *IDA* shall be one of exactly *TLC* item descriptor areas in the implementation parameter descriptor.

   c) Exactly one of the following is true:

Case:

i) TYPE indicates CHARACTER or CHARACTER VARYING, or CHARACTER LARGE OBJECT and LENGTH is a valid length value for a <character string type>.

ii) TYPE indicates BINARY, BINARY VARYING, or BINARY LARGE OBJECT and LENGTH is a valid length value for a <binary string type>.

iii) TYPE indicates NUMERIC and PRECISION and SCALE are valid precision and scale values for the NUMERIC data type.

iv) TYPE indicates DECIMAL and PRECISION and SCALE are valid precision and scale values for the DECIMAL data type.

v) TYPE indicates SMALLINT, INTEGER, BIGINT, REAL, or DOUBLE PRECISION.

vi) TYPE indicates FLOAT and PRECISION is a valid precision value for the FLOAT data type.

vii) TYPE indicates BOOLEAN.

viii) TYPE indicates a <datetime type>, DATETIME_INTERVAL_CODE is one of the code values in Table 9, “Codes associated with datetime data types in SQL/CLI”, and PRECISION is a valid precision value for the <time precision> or <timestamp precision> of the indicated datetime data type.

ix) TYPE indicates an <interval type>, DATETIME_INTERVAL_CODE is one of the code values in Table 10, “Codes associated with <interval qualifier> in SQL/CLI”, to indicate the <interval qualifier> of the interval data type, DATETIME_INTERVAL_PRECISION is a valid <interval leading field precision>, and PRECISION is a valid precision value for <interval fractional seconds precision>, if applicable.

x) TYPE indicates USER-DEFINED TYPE.

xi) TYPE indicates REF.

xii) TYPE indicates ROW, the value *N* of DEGREE is a valid value for the degree of a row type, there are exactly *N* immediately subordinate descriptor areas of *IDA*, and those item descriptor areas are valid.

xiii) TYPE indicates ARRAY or ARRAY LOCATOR, the value of CARDINALITY is a valid value for the maximum cardinality of an array, there is exactly one immediately subordinate descriptor area of *IDA*, and that item descriptor area is valid.

xiv) TYPE indicates an implementation-defined data type.

6) Let *HL* be the programming language of the invoking host program. Let *operative data type correspondence table* be the data type correspondence table for *HL* as specified in Subclause 5.15, “SQL/CLI data type correspondences”. Refer to the two columns of the operative data type correspondence table as the *SQL data type column* and the *host data type column*.

7) A CLI item descriptor area in a CLI descriptor area that is not an implementation row descriptor is *consistent* if and only if all of the following are true:

a) TYPE indicates DEFAULT or is one of the code values in Table 8, “Codes used for application data types in SQL/CLI”.

b) All of the following are true:

i) TYPE is one of the code values in Table 8, “Codes used for application data types in SQL/CLI”.

ii) TYPE is neither ROW, ARRAY, nor MULTISET.

iii) The row that contains the SQL data type corresponding to TYPE in the SQL data type column of the operative data type correspondence table does not contain “None” in the host data type column.

c) Exactly one of the following is true:

i) TYPE indicates NUMERIC and PRECISION and SCALE are valid precision and scale values for the NUMERIC data type.

ii) TYPE indicates DECIMAL and PRECISION and SCALE are valid precision and scale values for the DECIMAL data type.

iii) TYPE indicates FLOAT and PRECISION is a valid precision value for the FLOAT data type.

iv) TYPE indicates DEFAULT, CHARACTER, CHARACTER LARGE OBJECT, CHARACTER LARGE OBJECT LOCATOR, BINARY, BINARY VARYING, BINARY LARGE OBJECT, BINARY LARGE OBJECT LOCATOR, SMALLINT, INTEGER, BIGINT, REAL, DOUBLE PRECISION, USER-DEFINED TYPE LOCATOR, or REF.

v) TYPE indicates ROW and, where *N* is the value of the DEGREE field in the corresponding item descriptor area in the implementation parameter descriptor, there are exactly *N* immediately subordinate descriptor areas of *IDA*, and those item descriptor areas are valid.

vi) TYPE indicates ARRAY, ARRAY LOCATOR, MULTISET, or MULTISET LOCATOR, there is exactly 1 (one) immediately subordinate descriptor area of *IDA*, and that item descriptor area is valid.

vii) TYPE indicates an implementation-defined data type.

8) Let *IDA* be a CLI item descriptor area in an application parameter descriptor. Let *IDA1* be the corresponding item descriptor area in the implementation parameter descriptor.

9) If the OCTET_LENGTH_POINTER field of *IDA* has the same non-zero value as the INDICATOR_POINTER field of *IDA*, then *SHARE* is true for *IDA*; otherwise, *SHARE* is false for *IDA*.

10) Case:

a) If *SHARE* is true and the value of the commonly addressed host variable is the appropriate 'Code' for SQL NULL DATA in Table 27, “Miscellaneous codes used in CLI”, then *NULL* is true for *IDA*.

b) If *SHARE* is false, INDICATOR_POINTER is not zero, and the value of the host variable addressed by INDICATOR_POINTER is the appropriate 'Code' for SQL NULL DATA in Table 27, “Miscellaneous codes used in CLI”, then *NULL* is true for *IDA*.

c) Otherwise, *NULL* is false for *IDA*.

11) If *NULL* is false, OCTET_LENGTH_POINTER is not zero, and the value of the host variable addressed by OCTET_LENGTH_POINTER the appropriate 'Code' for DATA AT EXEC in Table 27, “Miscellaneous codes used in CLI”, then *DEFERRED* is true for *IDA*; otherwise, *DEFERRED* is false for *IDA*.

12) *IDA* is *valid* if and only if:

a) TYPE is one of the code values in Table 8, “Codes used for application data types in SQL/CLI”, and at least one of the following is true:

i) TYPE is ROW, ARRAY, or MULTISET.

ii) The row of the operative data type correspondences table that contains the SQL data type corresponding to the value of TYPE in the SQL data type column does not contain 'None' in the host data type column.

b) If LEVEL is 0 (zero) for *IDA*, then let *TLC* be the value of TOP_LEVEL_COUNT in the application parameter descriptor associated with *IDA*. *IDA* shall be one of exactly *TLC* item descriptor areas in the implementation parameter descriptor.

c) One of the following is true:

Case:

i) TYPE indicates CHARACTER, CHARACTER LARGE OBJECT, BINARY, BINARY VARYING, or BINARY LARGE OBJECT, and one of the following is true:

1) *NULL* is true.

2) *DEFERRED* is true.

3) OCTET_LENGTH_POINTER is not zero, PARAMETER_MODE in *IDA1* is PARAM MODE IN or PARAM MODE INOUT, the value *V* of the host variable addressed by OCTET_LENGTH_POINTER is greater than zero, and the number of characters wholly contained in the first *V* octets of the host variable addressed by DATA_POINTER is a valid length value for a CHARACTER, CHARACTER LARGE OBJECT, BINARY, BINARY VARYING, or BINARY LARGE OBJECT data type, as indicated by TYPE.

4) OCTET_LENGTH_POINTER is not zero, PARAMETER_MODE in *IDA1* is PARAM MODE IN or PARAM MODE INOUT, the value of the host variable addressed by OCTET_LENGTH_POINTER indicates NULL TERMINATED, and the number of characters of the value of the host variable addressed by DATA_POINTER that precede the implementation-defined null character that terminates a C character string is a valid length value for a CHARACTER, CHARACTER LARGE OBJECT, BINARY, BINARY VARYING, or BINARY LARGE OBJECT data type, as indicated by TYPE.

5) OCTET_LENGTH_POINTER is zero, PARAMETER_MODE in *IDA1* is PARAM MODE IN or PARAM MODE INOUT, and the number of characters of the value of the host variable addressed by DATA_POINTER that precede the implementation-defined null character that terminates a C character string is a valid length value for a CHARACTER, CHARACTER LARGE OBJECT, BINARY, BINARY VARYING, or BINARY LARGE OBJECT data type, as indicated by TYPE.

6) PARAMETER_MODE in *IDA1* is PARAM MODE OUT.

ii) TYPE indicates CHARACTER LARGE OBJECT LOCATOR, BINARY LARGE OBJECT LOCATOR, or USER-DEFINED TYPE LOCATOR and one of the following is true:

1) *NULL* is true.

2) *DEFERRED* is true.

iii) TYPE indicates NUMERIC and PRECISION and SCALE are valid precision and scale values for the NUMERIC data type.

iv) TYPE indicates DECIMAL and PRECISION and SCALE are valid precision and scale values for the DECIMAL data type.

v) TYPE indicates SMALLINT, INTEGER, BIGINT, REAL, or DOUBLE PRECISION.

vi) TYPE indicates FLOAT and PRECISION is a valid precision value for the FLOAT data type.

vii) TYPE indicates REF and one of the following is true:

1) *NULL* is true.

2) *DEFERRED* is true.

viii) TYPE indicates ROW and, where *N* is the value of the DEGREE field in the corresponding item descriptor area in the implementation parameter descriptor, there are exactly *N* immediately subordinate descriptor areas of *IDA*, and those item descriptor areas are valid.

ix) TYPE indicates ARRAY, ARRAY LOCATOR, MULTISET, or MULTISET LOCATOR, there is exactly 1 (one) immediately subordinate descriptor area of *IDA*, and that item descriptor area is valid.

x) TYPE indicates an implementation-defined data type.

d) One of the following is true:

i) DATA_POINTER is zero and *NULL* is true.

ii) DATA_POINTER is zero and *DEFERRED* is true.

iii) DATA_POINTER is not zero and exactly one of the following is true:

1) *NULL* is true.

2) *DEFERRED* is true.

3) PARAMETER_MODE in *IDA1* is PARAM MODE IN or PARAM MODE INOUT and the value of the host variable addressed by DATA_POINTER is a valid value of the data type indicated by TYPE.

4) PARAMETER_MODE in *IDA1* is PARAM MODE OUT.

13) A CLI item descriptor area in an application row descriptor is *valid* if and only if:

a) TYPE is one of the code values in Table 8, “Codes used for application data types in SQL/CLI”, and at least one of the following is true:

i) TYPE is ROW, ARRAY, or MULTISET.

ii) The row of the operative data type correspondences table that contains the SQL data type corresponding to the value of TYPE in the SQL data type column does not contain 'None' in the host data type column.

b) If LEVEL is 0 (zero) for *IDA*, then let *TLC* be the value of TOP_LEVEL_COUNT in the application parameter descriptor associated with *IDA*. *IDA* shall be one of exactly *TLC* item descriptor areas in the implementation parameter descriptor.

c) One of the following is true:

Case:

i) TYPE indicates NUMERIC and PRECISION and SCALE are valid precision and scale values for the NUMERIC data type.

ii) TYPE indicates DECIMAL and PRECISION and SCALE are valid precision and scale values for the DECIMAL data type.

iii) TYPE indicates FLOAT and PRECISION is a valid precision value for the FLOAT data type.

iv) TYPE indicates CHARACTER, CHARACTER LARGE OBJECT, CHARACTER LARGE OBJECT LOCATOR, BINARY, BINARY VARYING, BINARY LARGE OBJECT, BINARY LARGE OBJECT LOCATOR, SMALLINT, INTEGER, BIGINT, REAL, DOUBLE PRECISION, USER-DEFINED TYPE LOCATOR, or REF.

v) TYPE indicates ROW and, where *N* is the value of the DEGREE field in the corresponding item descriptor area in the implementation parameter descriptor, there are exactly *N* immediately subordinate descriptor areas of *IDA*, and those item descriptor areas are valid.

vi) TYPE indicates ARRAY, ARRAY LOCATOR, MULTISET, or MULTISET LOCATOR, there is exactly 1 (one) immediately subordinate descriptor area of *IDA*, and that item descriptor area is valid.

vii) TYPE indicates an implementation-defined data type.

Table 6 — Fields in SQL/CLI row and parameter descriptor areas

Field	**Data Type**
ALLOC_TYPE	SMALLINT
ARRAY_SIZE	INTEGER
ARRAY_STATUS_POINTER	host variable address of INTEGER
COUNT	SMALLINT
DYNAMIC_FUNCTION	CHARACTER VARYING(*L*)†
DYNAMIC_FUNCTION_CODE	INTEGER
KEY_TYPE	SMALLINT
ROWS_PROCESSED_POINTER	host variable address of INTEGER
TOP_LEVEL_COUNT	SMALLINT
Implementation-defined header field	Implementation-defined data type
CARDINALITY	INTEGER
CHARACTER_SET_CATALOG	CHARACTER VARYING(*L*)†
CHARACTER_SET_NAME	CHARACTER VARYING(*L*)†
CHARACTER_SET_SCHEMA	CHARACTER VARYING(*L*)†

Field	Data Type
COLLATION_CATALOG	CHARACTER VARYING(*L*)†
COLLATION_NAME	CHARACTER VARYING(*L*)†
COLLATION_SCHEMA	CHARACTER VARYING(*L*)†
CURRENT_TRANSFORM_GROUP	CHARACTER VARYING(*L1*)†
DATA_POINTER	host variable address
DATETIME_INTERVAL_CODE	SMALLINT
DATETIME_INTERVAL_PRECISION	SMALLINT
DEGREE	INTEGER
INDICATOR_POINTER	host variable address of INTEGER
KEY_MEMBER	SMALLINT
LENGTH	INTEGER
LEVEL	INTEGER
NAME	CHARACTER VARYING(*L*)†
NULLABLE	SMALLINT
OCTET_LENGTH	INTEGER
OCTET_LENGTH_POINTER	host variable address of INTEGER
PARAMETER_MODE	SMALLINT
PARAMETER_ORDINAL_POSITION	SMALLINT
PARAMETER_SPECIFIC_CATALOG	CHARACTER VARYING(*L*)†
PARAMETER_SPECIFIC_NAME	CHARACTER VARYING(*L*)†
PARAMETER_SPECIFIC_SCHEMA	CHARACTER VARYING(*L*)†
PRECISION	SMALLINT
RETURNED_CARDINALITY_POINTER	host variable address of INTEGER
SCALE	SMALLINT

Field	Data Type
SCOPE_CATALOG	CHARACTER VARYING(*L*)†
SCOPE_NAME	CHARACTER VARYING(*L*)†
SCOPE_SCHEMA	CHARACTER VARYING(*L*)†
SPECIFIC_TYPE_CATALOG	CHARACTER VARYING(*L*)†
SPECIFIC_TYPE_NAME	CHARACTER VARYING(*L*)†
SPECIFIC_TYPE_SCHEMA	CHARACTER VARYING(*L*)†
TYPE	SMALLINT
UNNAMED	SMALLINT
USER_DEFINED_TYPE_CATALOG	CHARACTER VARYING(*L*)†
USER_DEFINED_TYPE_NAME	CHARACTER VARYING(*L*)†
USER_DEFINED_TYPE_SCHEMA	CHARACTER VARYING(*L*)†
USER_DEFINED_TYPE_CODE	SMALLINT
Implementation-defined item field	Implementation-defined data type
† **Where** *L* is an implementation-defined integer not less than 128, and *L1* is the implementation-defined maximum length for the <general value specification> CURRENT_TRANSFORM_GROUP_FOR_TYPE.	

General Rules

1) Table 7, “Codes used for implementation data types in SQL/CLI”, specifies the codes associated with the SQL data types used in implementation descriptor areas.

Table 7 — Codes used for implementation data types in SQL/CLI

Data Type	Code
ARRAY	50
BIGINT	25
BINARY	60
BINARY LARGE OBJECT	30

Data Type	Code
BINARY VARYING	61
BOOLEAN	16
CHARACTER	1 (one)
CHARACTER LARGE OBJECT	40
CHARACTER VARYING	12
DATE, TIME, TIME WITH TIME ZONE, TIMESTAMP, or TIMESTAMP WITH TIME ZONE	9
DECIMAL	3
DOUBLE PRECISION	8
FLOAT	6
INTEGER	4
INTERVAL	10
MULTISET	55
NUMERIC	2
REAL	7
REF	20
ROW	19
SMALLINT	5
USER-DEFINED TYPE	17
Implementation-defined data type	< 0 (zero)

2) Table 8, "Codes used for application data types in SQL/CLI", specifies the codes associated with the SQL data types used in application descriptor areas.

Table 8 — Codes used for application data types in SQL/CLI

Data Type	Code
Implementation-defined data type	< 0 (zero)

Data Type	Code
ARRAY LOCATOR	51
BIGINT	25
BINARY	60
BINARY LARGE OBJECT	30
BINARY LARGE OBJECT LOCATOR	31
BINARY VARYING	61
CHARACTER	1 (one)
CHARACTER LARGE OBJECT	40
CHARACTER LARGE OBJECT LOCATOR	41
DECIMAL	3
DOUBLE PRECISION	8
FLOAT	6
INTEGER	4
MULTISET LOCATOR	56
NUMERIC	2
REAL	7
REF	20
SMALLINT	5
USER-DEFINED TYPE LOCATOR	18

3) Table 9, “Codes associated with datetime data types in SQL/CLI”, specifies the codes associated with the datetime data types allowed in SQL/CLI.

Table 9 — Codes associated with datetime data types in SQL/CLI

Datetime Data Type	Code
DATE	1 (one)
TIME	2

Datetime Data Type	Code
TIME WITH TIME ZONE	4
TIMESTAMP	3
TIMESTAMP WITH TIME ZONE	5

4) Table 10, “Codes associated with <interval qualifier> in SQL/CLI”, specifies the codes associated with <interval qualifier>s for interval data types in SQL/CLI.

Table 10 — Codes associated with <interval qualifier> in SQL/CLI

Interval qualifier	Code
DAY	3
DAY TO HOUR	8
DAY TO MINUTE	9
DAY TO SECOND	10
HOUR	4
HOUR TO MINUTE	11
HOUR TO SECOND	12
MINUTE	5
MINUTE TO SECOND	13
MONTH	2
SECOND	6
YEAR	1 (one)
YEAR TO MONTH	7

5) Table 11, “Codes associated with <parameter mode> in SQL/CLI”, specifies the codes associated with the SQL parameter modes.

Table 11 — Codes associated with <parameter mode> in SQL/CLI

Parameter mode	Code
PARAM MODE IN	1 (one)
PARAM MODE INOUT	2
PARAM MODE OUT	4

Table 12 — Codes associated with user-defined types in SQL/CLI

User-defined Type	Code
DISTINCT	1 (one)
STRUCTURED	2

5.14 Other tables associated with CLI

This Subclause is modified by Subclause 19.4, "Other tables associated with CLI", in ISO/IEC 9075-9.

The tables contained in this Subclause are used to specify the codes used by the various CLI routines.

Table 13 — Codes used for SQL/CLI diagnostic fields

Field	Code	Type
CATALOG_NAME	18	Status
CLASS_ORIGIN	8	Status
COLUMN_NAME	21	Status
COLUMN_NUMBER	-1247	Status
CONDITION_IDENTIFIER	25	Status
CONDITION_NUMBER	14	Status
CONNECTION_NAME	10	Status
CONSTRAINT_CATALOG	15	Status
CONSTRAINT_NAME	17	Status
CONSTRAINT_SCHEMA	16	Status
CURSOR_NAME	22	Status
DYNAMIC_FUNCTION	7	Header
DYNAMIC_FUNCTION_CODE	12	Header
MESSAGE_LENGTH	23	Status
MESSAGE_OCTET_LENGTH	24	Status
MESSAGE_TEXT	6	Status
MORE	13	Header
NATIVE_CODE	5	Status
NUMBER	2	Header
PARAMETER_MODE	37	Status
PARAMETER_NAME	26	Status

Field	Code	Type
PARAMETER_ORDINAL_POSITION	38	Status
RETURNCODE	1 (one)	Header
ROUTINE_CATALOG	27	Status
ROUTINE_NAME	29	Status
ROUTINE_SCHEMA	28	Status
ROW_COUNT	3	Header
ROW_NUMBER	-1248	Status
SCHEMA_NAME	19	Status
SERVER_NAME	11	Status
SPECIFIC_NAME	30	Status
SQLSTATE	4	Status
SUBCLASS_ORIGIN	9	Status
TABLE_NAME	20	Status
TRANSACTION_ACTIVE	36	Header
TRANSACTIONS_COMMITTED	34	Header
TRANSACTIONS_ROLLED_BACK	35	Header
TRIGGER_CATALOG	31	Status
TRIGGER_NAME	33	Status
TRIGGER_SCHEMA	32	Status
Implementation-defined diagnostics header field	< 0 (zero)[1]	Header
Implementation-defined diagnostics status field	< 0 (zero)[1]	Status

[1] Except for values in this table that are less than 0 (zero).

Table 14 — Codes used for SQL/CLI handle types

Handle type	Code
CONNECTION HANDLE	2
DESCRIPTOR HANDLE	4
ENVIRONMENT HANDLE	1 (one)
STATEMENT HANDLE	3
Implementation-defined handle type	< 1 (one) or > 100

Table 15 — Codes used for transaction termination

Termination type	Code
COMMIT	0 (zero)
ROLLBACK	1 (one)
SAVEPOINT NAME ROLLBACK	2
SAVEPOINT NAME RELEASE	4
COMMIT AND CHAIN	6
ROLLBACK AND CHAIN	7
Implementation-defined termination type	< 0 (zero)

Table 16 — Codes used for environment attributes

Attribute	Code	May be set
NULL TERMINATION	10001	Yes
Implementation-defined environment attribute	≥ 0 (zero), except values given above	Implementation-defined

Table 17 — Codes used for connection attributes

Attribute	Code	May be set
POPULATE IPD	10001	No
SAVEPOINT NAME	10027	Yes
Implementation-defined connection attribute	≥ 0 (zero), except values given above	Implementation-defined

Table 18 — Codes used for statement attributes

Attribute	Code	May be set
APD HANDLE	10011	Yes
ARD HANDLE	10010	Yes
IPD HANDLE	10013	No
IRD HANDLE	10012	No
CURRENT OF POSITION	10027	Yes
CURSOR HOLDABLE	-3	Yes
CURSOR SCROLLABLE	-1	Yes
CURSOR SENSITIVITY	-2	Yes
METADATA ID	10014	Yes
NEST DESCRIPTOR	10029	Yes
Implementation-defined statement attribute	≥ 0 (zero), except values given above	Implementation-defined

Table 19 — Codes used for FreeStmt options

Option	Code
CLOSE CURSOR	0 (zero)

Option	Code
FREE HANDLE	1 (one)
UNBIND COLUMNS	2
UNBIND PARAMETERS	3
REALLOCATE	4

Table 20 — Data types of attributes

Attribute	Data type	Values
NULL TERMINATION	INTEGER	0 (*False*) 1 (*True*)
POPULATE IPD	INTEGER	0 (*False*) 1 (*True*)
APD HANDLE	INTEGER	Handle value
ARD HANDLE	INTEGER	Handle value
IPD HANDLE	INTEGER	Handle value
IRD HANDLE	INTEGER	Handle value
CURRENT OF POSITION	INTEGER	Integer value denoting the current row in the rowset
CURSOR HOLDABLE	INTEGER	0 (NONHOLDABLE) 1 (HOLDABLE)
CURSOR SCROLLABLE	INTEGER	0 (NONSCROLLABLE) 1 (SCROLLABLE)
CURSOR SENSITIVITY	INTEGER	0 (ASENSITIVE) 1 (INSENSITIVE) 2 (SENSITIVE)
METADATA ID	INTEGER	0 (FALSE) 1 (TRUE)
NEST DESCRIPTOR	INTEGER	0 (FALSE) 1 (TRUE)
SAVEPOINT NAME	CHARACTER	Not specified

Table 21 — Codes used for SQL/CLI descriptor fields

Field	Code	SQL Item Descriptor Name	Type
ALLOC_TYPE	1099	(*Not applicable*)	Header
ARRAY_SIZE	20	(*Not applicable*)	Header

Field	**Code**	**SQL Item Descriptor Name**	**Type**
ARRAY_STATUS_POINTER	21	(*Not applicable*)	Header
CARDINALITY	1040	CARDINALITY	Item
CHARACTER_SET_CATALOG	1018	CHARACTER_SET_CATALOG	Item
CHARACTER_SET_NAME	1020	CHARACTER_SET_NAME	Item
CHARACTER_SET_SCHEMA	1019	CHARACTER_SET_SCHEMA	Item
COLLATION_CATALOG	1015	COLLATION_CATALOG	Item
COLLATION_NAME	1017	COLLATION_NAME	Item
COLLATION_SCHEMA	1016	COLLATION_SCHEMA	Item
COUNT	1001	COUNT	Header
CURRENT_TRANSFORM_GROUP	1039	(*Not applicable*)	Item
DATA_POINTER	1010	DATA	Item
DATETIME_INTERVAL_CODE	1007	DATETIME_INTERVAL_CODE	Item
DATETIME_INTERVAL_PRECISION	26	DATETIME_INTERVAL_PRECISION	Item
DEGREE	1041	DEGREE	Item
DYNAMIC_FUNCTION	1031	DYNAMIC_FUNCTION	Header
DYNAMIC_FUNCTION_CODE	1032	DYNAMIC_FUNCTION_CODE	Header
INDICATOR_POINTER	1009	INDICATOR	Item
KEY_MEMBER	1030	KEY_MEMBER	Item
KEY_TYPE	1029	KEY_TYPE	Header
LENGTH	1003	LENGTH	Item
LEVEL	1042	LEVEL	Item
NAME	1011	NAME	Item
NULLABLE	1008	NULLABLE	Item
OCTET_LENGTH	1013	OCTET_LENGTH	Item

Field	Code	SQL Item Descriptor Name	Type
OCTET_LENGTH_POINTER	1004	Both OCTET_LENGTH (input) and RETURNED_OCTET_LENGTH (output)	Item
PARAMETER_MODE	1021	PARAMETER_MODE	Item
PARAMETER_ORDINAL_POSITION	1022	PARAMETER_ORDINAL_POSITION	Item
PARAMETER_SPECIFIC_CATALOG	1023	PARAMETER_SPECIFIC_CATALOG	Item
PARAMETER_SPECIFIC_NAME	1025	PARAMETER_SPECIFIC_NAME	Item
PARAMETER_SPECIFIC_SCHEMA	1024	PARAMETER_SPECIFIC_SCHEMA	Item
PRECISION	1005	PRECISION	Item
RETURNED_CARDINALITY_POINTER	1043	RETURNED_CARDINALITY	Item
ROW_PROCESSED_POINTER	34	(*Not applicable*)	Header
SCALE	1006	SCALE	Item
SCOPE_CATALOG	1033	SCOPE_CATALOG	Item
SCOPE_NAME	1034	SCOPE_NAME	Item
SCOPE_SCHEMA	1035	SCOPE_SCHEMA	Item
SPECIFIC_TYPE_CATALOG	1036	(*Not applicable*)	Item
SPECIFIC_TYPE_NAME	1038	(*Not applicable*)	Item
SPECIFIC_TYPE_SCHEMA	1037	(*Not applicable*)	Item
TOP_LEVEL_COUNT	1044	TOP_LEVEL_COUNT	Header
TYPE	1002	TYPE	Item
UNNAMED	1012	UNNAMED	Item
USER_DEFINED_TYPE_CATALOG	1026	USER_DEFINED_TYPE_CATALOG	Item
USER_DEFINED_TYPE_NAME	1028	USER_DEFINED_TYPE_NAME	Item
USER_DEFINED_TYPE_SCHEMA	1027	USER_DEFINED_TYPE_SCHEMA	Item
USER_DEFINED_TYPE_CODE	1045	USER_DEFINED_TYPE_CODE	Item

Field	Code	SQL Item Descriptor Name	Type
Implementation-defined descriptor header field	0 (zero) through 999, or ≥ 1200, excluding values defined in this table	Implementation-defined descriptor header field	Header
Implementation-defined descriptor item field	0 (zero) through 999, or ≥ 1200, excluding values defined in this table	Implementation-defined descriptor item field	Item

Table 22 — Ability to set SQL/CLI descriptor fields

	May be set			
Field	**ARD**	**IRD**	**APD**	**IPD**
ALLOC_TYPE	No	No	No	No†
ARRAY_SIZE		No		No
ARRAY_STATUS_POINTER				
CARDINALITY	No	No	No	
CHARACTER_SET_CATALOG		No		
CHARACTER_SET_NAME		No		
CHARACTER_SET_SCHEMA		No		
COLLATION_CATALOG		No		
COLLATION_NAME		No		
COLLATION_SCHEMA		No		

	May be set			
Field	**ARD**	**IRD**	**APD**	**IPD**
COUNT		No		
CURRENT_TRANSFORM_GROUP	No	No	No	No
DATA_POINTER		No		
DATETIME_INTERVAL_CODE		No		
DATETIME_INTERVAL_PRECISION		No		
DEGREE	No	No	No	
DYNAMIC_FUNCTION	No	No	No	No
DYNAMIC_FUNCTION_CODE	No	No	No	No
INDICATOR_POINTER		No		No
KEY_MEMBER	No	No	No	No
KEY_TYPE	No	No	No	No
LENGTH		No		
LEVEL		No		
NAME		No		
NULLABLE		No		
OCTET_LENGTH		No		
OCTET_LENGTH_POINTER		No		No
PARAMETER_MODE	No	No	No	
PARAMETER_ORDINAL_POSITION	No	No	No	
PARAMETER_SPECIFIC_CATALOG	No	No	No	
PARAMETER_SPECIFIC_NAME	No	No	No	
PARAMETER_SPECIFIC_SCHEMA	No	No	No	
PRECISION		No		
RETURNED_CARDINALITY_POINTER		No		No

	May be set			
Field	**ARD**	**IRD**	**APD**	**IPD**
ROWS_PROCESSED_POINTER	No		No	
SCALE		No		
SCOPE_CATALOG		No		
SCOPE_NAME		No		
SCOPE_SCHEMA		No		
SPECIFIC_TYPE_CATALOG	No	No	No	No
SPECIFIC_TYPE_NAME	No	No	No	No
SPECIFIC_TYPE_SCHEMA	No	No	No	No
TOP_LEVEL_COUNT		No		
TYPE		No		
UNNAMED		No		
USER_DEFINED_TYPE_CATALOG		No		
USER_DEFINED_TYPE_NAME		No		
USER_DEFINED_TYPE_SCHEMA		No		
USER_DEFINED_TYPE_CODE	No	No	No	No
Implementation-defined descriptor header field	ID	ID	ID	ID
Implementation-defined descriptor item field	ID	ID	ID	ID
† **Where** “No” means that the descriptor field is not settable, “ID” means that it is implementation-defined whether or not the descriptor field is settable, and the absence of any notation means that the descriptor field is settable.				

Table 23 — Ability to retrieve SQL/CLI descriptor fields

	May be retrieved			
Field	**ARD**	**IRD**	**APD**	**IPD**
ALLOC_TYPE		PS		
ARRAY_SIZE		No		No

	May be retrieved			
Field	**ARD**	**IRD**	**APD**	**IPD**
ARRAY_STATUS_POINTER				
CARDINALITY	No	PS	No	
CHARACTER_SET_CATALOG		PS		
CHARACTER_SET_NAME		PS		
CHARACTER_SET_SCHEMA		PS		
COLLATION_CATALOG		PS		
COLLATION_NAME		PS		
COLLATION_SCHEMA		PS		
COUNT		PS		
CURRENT_TRANSFORM_GROUP		PS		
DATA_POINTER		No		No†
DATETIME_INTERVAL_CODE		PS		
DATETIME_INTERVAL_PRECISION		PS		
DEGREE	No	PS	No	
DYNAMIC_FUNCTION	No		No	
DYNAMIC_FUNCTION_CODE	No		No	
INDICATOR_POINTER		No		No
KEY_MEMBER	No	PS	No	No
KEY_TYPE	No	PS	No	No
LENGTH		PS		
LEVEL		PS		
NAME		PS		
NULLABLE		PS		
OCTET_LENGTH		PS		

	May be retrieved			
Field	**ARD**	**IRD**	**APD**	**IPD**
OCTET_LENGTH_POINTER		No		No
PARAMETER_MODE	No	PS	No	No
PARAMETER_ORDINAL_POSITION	No	PS	No	No
PARAMETER_SPECIFIC_CATALOG	No	PS	No	No
PARAMETER_SPECIFIC_NAME	No	PS	No	No
PARAMETER_SPECIFIC_SCHEMA	No	PS	No	No
PRECISION		PS		
RETURNED_CARDINALITY_POINTER		No		No
ROWS_PROCESSED_POINTER	No		No	
SCALE		PS		
SCOPE_CATALOG		PS		
SCOPE_NAME		PS		
SCOPE_SCHEMA		PS		
SPECIFIC_TYPE_CATALOG		PS		
SPECIFIC_TYPE_NAME		PS		
SPECIFIC_TYPE_SCHEMA		PS		
TOP_LEVEL_COUNT		PS		
TYPE		PS		
UNNAMED		PS		
USER_DEFINED_TYPE_CATALOG		PS		
USER_DEFINED_TYPE_NAME		PS		
USER_DEFINED_TYPE_SCHEMA		PS		
USER_DEFINED_TYPE_CODE		PS		
Implementation-defined descriptor header field	ID	ID	ID	ID

	May be retrieved			
Field	**ARD**	**IRD**	**APD**	**IPD**
Implementation-defined descriptor item field	ID	ID	ID	ID
† **Where** "No" means that the descriptor field is not retrievable, *PS* means that the descriptor field is retrievable from the IRD only when a prepared or executed statement is associated with the IRD, the absence of any notation means that the descriptor field is retrievable, and "ID" means that it is implementation-defined whether or not the descriptor field is retrievable.				

Table 24 — SQL/CLI descriptor field default values

	Default values			
Field	**ARD**	**IRD**	**APD**	**IPD**
ALLOC_TYPE	AUTO-MATIC or USER	AUTO-MATIC	AUTO-MATIC or USER	AUTO-MATIC
ARRAY_SIZE	1 (one)		1 (one)	
ARRAY_STATUS_POINTER	Null	Null	Null	Null
CARDINALITY				
CHARACTER_SET_CATALOG				
CHARACTER_SET_NAME				
CHARACTER_SET_SCHEMA				
COLLATION_CATALOG				
COLLATION_NAME				
COLLATION_SCHEMA				
COUNT	0 (zero)		0 (zero)†	
CURRENT_TRANSFORM_GROUP				
DATA_POINTER	Null		Null	
DATETIME_INTERVAL_CODE				
DATETIME_INTERVAL_PRECISION				
DEGREE				

	Default values			
Field	**ARD**	**IRD**	**APD**	**IPD**
DYNAMIC_FUNCTION				
DYNAMIC_FUNCTION_CODE				
INDICATOR_POINTER	Null		Null	
KEY_MEMBER				
KEY_TYPE				
LENGTH				
LEVEL	0 (zero)		0 (zero)	
NAME				
NULLABLE				
OCTET_LENGTH				
OCTET_LENGTH_POINTER	Null		Null	
PARAMETER_MODE				
PARAMETER_ORDINAL_POSITION				
PARAMETER_SPECIFIC_CATALOG				
PARAMETER_SPECIFIC_NAME				
PARAMETER_SPECIFIC_SCHEMA				
PRECISION				
RETURNED_CARDINAL-ITY_POINTER	Null		Null	
ROWS_PROCESSED_POINTER		Null		Null
SCALE				
SCOPE_CATALOG				
SCOPE_NAME				
SCOPE_SCHEMA				
SPECIFIC_TYPE_CATALOG				

	Default values			
Field	**ARD**	**IRD**	**APD**	**IPD**
SPECIFIC_TYPE_NAME				
SPECIFIC_TYPE_SCHEMA				
TOP_LEVEL_COUNT	0 (zero)		0 (zero)	
TYPE	DEFAULT		DEFAULT	
UNNAMED				
USER_DEFINED_TYPE_CATALOG				
USER_DEFINED_TYPE_NAME				
USER_DEFINED_TYPE_SCHEMA				
USER_DEFINED_TYPE_CODE				
Implementation-defined descriptor header field	ID	ID	ID	ID
Implementation-defined descriptor item field	ID	ID	ID	ID
† **Where** “Null” means that the descriptor field's default value is a null pointer, the absence of any notation means that the descriptor field's default value is initially undefined, “ID” means that the descriptor field's default value is implementation-defined, and any other value specifies the descriptor field's default value.				

Table 25 — Codes used for fetch orientation

Fetch Orientation	Code
NEXT	1 (one)
FIRST	2
LAST	3
PRIOR	4
ABSOLUTE	5
RELATIVE	6

Table 26 — Multi-row fetch status codes

Return code meaning	Return code
Row success	0 (zero)
Row success with information	6
Row error	5
No row	3

Table 27 — Miscellaneous codes used in CLI

Context	Code	Indicates
Allocation type	1 (one)	AUTOMATIC
Allocation type	2	USER
Attribute value	0 (zero)	FALSE, NONSCROLLABLE, ASENSITIVE, NO NULLS, NONHOLDABLE
Attribute value	1 (one)	TRUE, SCROLLABLE, INSENSITIVE, NULLABLE, HOLD-ABLE
Attribute value	2	SENSITIVE
Data type	0 (zero)	ALL TYPES
Data type	-99	APD TYPE
Data type	-99	ARD TYPE
Data type	99	DEFAULT
Deferrable constraints	5	INITIALLY DEFERRED
Deferrable constraints	6	INITIALLY IMMEDIATE
Deferrable constraints	7	NOT DEFERRABLE
Input string length	-3	NULL TERMINATED
Input or output data	-1	SQL NULL DATA
Parameter length	-2	DATA AT EXEC

Context	Code	Indicates
Referential Constraint	0 (zero)	CASCADE
Referential Constraint	1 (one)	RESTRICT
Referential Constraint	4	SET DEFAULT
Referential Constraint	2	SET NULL
Referential Constraint	3	NO ACTION

Table 28 — Codes used to identify SQL/CLI routines

Generic Name	Code
AllocConnect	1 (one)
AllocEnv	2
AllocHandle	1001
AllocStmt	3
BindCol	4
BindParameter	72
Cancel	5
CloseCursor	1003
ColAttribute	6
ColumnPrivileges	56
Columns	40
Connect	7
CopyDesc	1004
DataSources	57
DescribeCol	8
Disconnect	9
EndTran	1005

Generic Name	Code
Error	10
ExecDirect	11
Execute	12
Fetch	13
FetchScroll	1021
ForeignKeys	60
FreeConnect	14
FreeEnv	15
FreeHandle	1006
FreeStmt	16
GetConnectAttr	1007
GetCursorName	17
GetData	43
GetDescField	1008
GetDescRec	1009
GetDiagField	1010
GetDiagRec	1011
GetEnvAttr	1012
GetFeatureInfo	1027
GetFunctions	44
GetInfo	45
GetLength	1022
GetParamData	1025
GetPosition	1023
GetSessionInfo	1028

Generic Name	**Code**
GetStmtAttr	1014
GetSubString	1024
GetTypeInfo	47
MoreResults	61
NextResult	73
NumResultCols	18
ParamData	48
Prepare	19
PrimaryKeys	65
PutData	49
RowCount	20
SetConnectAttr	1016
SetCursorName	21
SetDescField	1017
SetDescRec	1018
SetEnvAttr	1019
SetStmtAttr	1020
SpecialColumns	52
StartTran	74
TablePrivileges	70
Tables	54
Implementation-defined CLI routine	< 0 (zero), or 400 through 1299, or ≥ 2000

Table 29 — Codes and data types for implementation information

Information Type	Code	Data Type
CATALOG NAME	10003	CHARACTER(1)
COLLATING SEQUENCE	10004	CHARACTER(254)
CURSOR COMMIT BEHAVIOR	23	SMALLINT
DATA SOURCE NAME	2	CHARACTER(128)
DBMS NAME	17	CHARACTER(254)
DBMS VERSION	18	CHARACTER(254)
DEFAULT TRANSACTION ISOLATION	26	INTEGER
IDENTIFIER CASE	28	SMALLINT
MAXIMUM CATALOG NAME LENGTH	34	SMALLINT
MAXIMUM COLUMN NAME LENGTH	30	SMALLINT
MAXIMUM COLUMNS IN GROUP BY	97	SMALLINT
MAXIMUM COLUMNS IN ORDER BY	99	SMALLINT
MAXIMUM COLUMNS IN SELECT	100	SMALLINT
MAXIMUM COLUMNS IN TABLE	101	SMALLINT
MAXIMUM CONCURRENT ACTIVITIES	1 (one)	SMALLINT
MAXIMUM CURSOR NAME LENGTH	31	SMALLINT
MAXIMUM DRIVER CONNECTIONS	0 (zero)	SMALLINT
MAXIMUM IDENTIFIER LENGTH	10005	SMALLINT
MAXIMUM SCHEMA NAME LENGTH	32	SMALLINT

Information Type	Code	Data Type
MAXIMUM STATEMENT OCTETS	20000	SMALLINT
MAXIMUM STATEMENT OCTETS DATA	20001	SMALLINT
MAXIMUM STATEMENT OCTETS SCHEMA	20002	SMALLINT
MAXIMUM TABLE NAME LENGTH	35	SMALLINT
MAXIMUM TABLES IN SELECT	106	SMALLINT
MAXIMUM USER NAME LENGTH	107	SMALLINT
NULL COLLATION	85	SMALLINT
ORDER BY COLUMNS IN SELECT	90	CHARACTER(1)
SEARCH PATTERN ESCAPE	14	CHARACTER(1)
SERVER NAME	13	CHARACTER(128)
SPECIAL CHARACTERS	94	CHARACTER(254)
TRANSACTION CAPABLE	46	SMALLINT
TRANSACTION ISOLATION OPTION	72	INTEGER
Implementation-defined information type	Implementation-defined code	Implementation-defined data type
SQL implementation information	21000 through 24999	CHARACTER(*L*[1]) or INTEGER
SQL sizing information	25000 through 29999	INTEGER
Implementation-defined implementation information	11000 through 14999	CHARACTER(*L*[1]) or INTEGER
Implementation-defined sizing information	15000 through 19999	INTEGER
[1] *L* is the implementation-defined maximum length of a variable-length character string.		

NOTE 12 — Additional implementation information items are defined in Subclause 6.44, “SQL_IMPLEMENTATION_INFO base table”, in ISO/IEC 9075-2.

Additional sizing items are defined in Subclause 6.45, “SQL_SIZING base table”, in ISO/IEC 9075-2.

Table 30 — Codes and data types for session implementation information

Information Type	Code	Data Type	<general value specification>
CURRENT USER	47	CHARACTER(*L*†)	USER and CURRENT_USER
CURRENT DEFAULT TRANS-FORM GROUP	20004	CHARACTER(*L*†)	CURRENT_DEFAULT_TRANS-FORM_GROUP
CURRENT PATH	20005	CHARACTER(*L*†)	CURRENT_PATH
CURRENT ROLE	20006	CHARACTER(*L*†)	CURRENT_ROLE
SESSION USER	20007	CHARACTER(*L*†)	SESSION_USER
SYSTEM USER	20008	CHARACTER(*L*†)	SYSTEM_USER
CURRENT CATA-LOG	20009	CHARACTER(*L*†)	CURRENT_CATALOG
CURRENT SCHEMA	20010	CHARACTER(*L*†)	CURRENT_SCHEMA
† **Where** *L* is the implementation-defined maximum length of the corresponding <general value specification>.			

Table 31 — Values for TRANSACTION ISOLATION OPTION with StartTran

Information Type	Value
READ UNCOMMITTED	1 (one)
READ COMMITTED	2
REPEATABLE READ	4
SERIALIZABLE	8

Table 32 — Values for TRANSACTION ACCESS MODE with StartTran

Information Type	Value
READ ONLY	1 (one)

Information Type	Value
READ WRITE	2

Table 33 — Codes used for concise data types

Data Type	Code
Implementation-defined data type	< 0 (zero)
CHARACTER	1 (one)
CHAR	1 (one)
NUMERIC	2
DECIMAL	3
DEC	3
INTEGER	4
INT	4
SMALLINT	5
FLOAT	6
REAL	7
DOUBLE	8
BINARY	60
BINARY VARYING	61
VARBINARY	61
CHARACTER VARYING	12
CHAR VARYING	12
VARCHAR	12
BOOLEAN	16
USER-DEFINED TYPE	17
ROW	19
REF	20

Data Type	Code
BIGINT	25
BINARY LARGE OBJECT	30
BLOB	30
CHARACTER LARGE OBJECT	40
CLOB	40
ARRAY	50
MULTISET	55
DATE	91
TIME	92
TIMESTAMP	93
TIME WITH TIME ZONE	94
TIMESTAMP WITH TIME ZONE	95
INTERVAL YEAR	101
INTERVAL MONTH	102
INTERVAL DAY	103
INTERVAL HOUR	104
INTERVAL MINUTE	105
INTERVAL SECOND	106
INTERVAL YEAR TO MONTH	107
INTERVAL DAY TO HOUR	108
INTERVAL DAY TO MINUTE	109
INTERVAL DAY TO SECOND	110
INTERVAL HOUR TO MINUTE	111
INTERVAL HOUR TO SECOND	112
INTERVAL MINUTE TO SECOND	113

Table 34 — Codes used with concise datetime data types in SQL/CLI

Concise Data Type Code	Data Type Code	Datetime Interval Code
91	9	1 (one)
92	9	2
93	9	3
94	9	4
95	9	5

Table 35 — Codes used with concise interval data types in SQL/CLI

Concise Data Type Code	Data Type Code	Datetime Interval Code
101	10	1 (one)
102	10	2
103	10	3
104	10	4
105	10	5
106	10	6
107	10	7
108	10	8
109	10	9
110	10	10
111	10	11
112	10	12
113	10	13

Table 36 — Concise codes used with datetime data types in SQL/CLI

Datetime Interval Code	Concise Code
1 (one)	91
2	92
3	93
4	94
5	95

Table 37 — Concise codes used with interval data types in SQL/CLI

Datetime Interval Code	Code
1 (one)	101
2	102
3	103
4	104
5	105
6	106
7	107
8	108
9	109
10	110
11	111
12	112
13	113

Table 38 — Special parameter values

Value Name	Value	Data Type
ALL CATALOGS	'%'	CHARACTER(1)
ALL SCHEMAS	'%'	CHARACTER(1)
ALL TYPES	'%'	CHARACTER(1)

Table 39 — Column types and scopes used with SpecialColumns

Context	Code	Indicates
Special Column Type	1 (one)	BEST ROWID
Scope of Row Id	0 (zero)	SCOPE CURRENT ROW
Scope of Row Id	1 (one)	SCOPE TRANSACTION
Scope of Row Id	2	SCOPE SESSION
Pseudo Column Flag	0 (zero)	PSEUDO UNKNOWN
Pseudo Column Flag	1 (one)	NOT PSEUDO
Pseudo Column Flag	2	PSEUDO

5.15 SQL/CLI data type correspondences

This Subclause is modified by Subclause 19.5, “SQL/CLI data type correspondences”, in ISO/IEC 9075-9.

Function

Specify the SQL/CLI data type correspondences for SQL data types and host language types associated with the required parameter mechanisms, as shown in Table 3, “Supported calling conventions of SQL/CLI routines by language”.

In the following tables, let *P* be <precision>, *S* be <scale>, *L* be <length>, *T* be <time fractional seconds precision>, and *Q* be <interval qualifier>.

Tables

Table 40 — SQL/CLI data type correspondences for Ada

SQL Data Type	Ada Data Type
ARRAY	*None*
ARRAY LOCATOR	SQL_STANDARD.INT
BIGINT	SQL_STANDARD.BIGINT
BINARY (*L*)	SQL_STANDARD.CHAR, with P'LENGTH of *L*
BINARY LARGE OBJECT (*L*)	SQL_STANDARD.CHAR, with P'LENGTH of *L*
BINARY LARGE OBJECT LOCATOR	SQL_STANDARD.INT
BINARY VARYING (*L*)	SQL_STANDARD.CHAR, with P'LENGTH of *L*
BOOLEAN	SQL_STANDARD.BOOLEAN
CHARACTER (*L*)	SQL_STANDARD.CHAR, with P'LENGTH of *L*
CHARACTER LARGE OBJECT (*L*)	SQL_STANDARD.CHAR, with P'LENGTH of *L*
CHARACTER LARGE OBJECT LOCATOR	SQL_STANDARD.INT
CHARACTER VARYING (*L*)	*None*
DATE	*None*
DECIMAL(*P*,*S*)	*None*

SQL Data Type	Ada Data Type
DOUBLE PRECISION	SQL_STANDARD.DOUBLE_PRECISION
FLOAT(*P*)	*None*
INTEGER	SQL_STANDARD.INT
INTERVAL(*Q*)	*None*
MULTISET	*None*
MULTISET LOCATOR	SQL_STANDARD.INT
NUMERIC(*P*,*S*)	*None*
REAL	SQL_STANDARD.REAL
REF	SQL_STANDARD.CHAR with P'LENGTH of *L*
ROW	*None*
SMALLINT	SQL_STANDARD.SMALLINT
TIME(*T*)	*None*
TIMESTAMP(*T*)	*None*
USER-DEFINED TYPE	*None*
USER-DEFINED TYPE LOCATOR	SQL_STANDARD.INT

Table 41 — SQL/CLI data type correspondences for C

SQL Data Type	C Data Type
ARRAY	*None*
ARRAY LOCATOR	long
BIGINT	long long
BINARY (*L*)	char, with length *L*
BINARY LARGE OBJECT (*L*)	char, with length *L*
BINARY LARGE OBJECT LOCATOR	long
BINARY VARYING (*L*)	char, with length *L*

SQL Data Type	C Data Type
BOOLEAN	short
CHARACTER (*L*)	char, with length (*L*+1)**k*[1]
CHARACTER LARGE OBJECT (*L*)	char, with length (*L*+1)**k*[1]
CHARACTER LARGE OBJECT LOCATOR	long
CHARACTER VARYING (*L*)	char, with length (*L*+1)**k*[1]
DATE	*None*
DECIMAL(*P*,*S*)	*None*
DOUBLE PRECISION	double
FLOAT(*P*)	*None*
INTEGER	long
INTERVAL(*Q*)	*None*
MULTISET	*None*
MULTISET LOCATOR	long
NUMERIC(*P*,*S*)	*None*
REAL	float
REF	char, with length *L*
ROW	*None*
SMALLINT	short
TIME(*T*)	*None*
TIMESTAMP(*T*)	*None*
USER-DEFINED TYPE	*None*
USER-DEFINED TYPE LOCATOR	long
[1] *k* is the length in units of C **char** of the largest character in the character set associated with the SQL data type.	

Table 42 — SQL/CLI data type correspondences for COBOL

SQL Data Type	COBOL Data Type
ARRAY	*None*
ARRAY LOCATOR	PICTURE S9(*PI*) USAGE BINARY, where *PI* is implementation-defined
BIGINT	PICTURE S9(*BPI*) USAGE BINARY, where *BPI* is implementation-defined
BINARY (*L*)	alphanumeric, with length *L*
BINARY LARGE OBJECT (*L*)	alphanumeric, with length *L*
BINARY LARGE OBJECT LOCATOR	PICTURE S9(*PI*) USAGE BINARY, where *PI* is implementation-defined
BINARY VARYING (*L*)	alphanumeric, with length *L*
BOOLEAN	PICTURE X
CHARACTER (*L*)	alphanumeric, with length *L*
CHARACTER LARGE OBJECT (*L*)	alphanumeric, with length *L*
CHARACTER LARGE OBJECT LOCATOR	PICTURE S9(*PI*) USAGE BINARY, where *PI* is implementation-defined
CHARACTER VARYING (*L*)	*None*
DATE	*None*
DECIMAL(*P*,*S*)	*None*
DOUBLE PRECISION	*None*
FLOAT(*P*)	*None*
INTEGER	PICTURE S9(*PI*) USAGE BINARY, where *PI* is implementation-defined
INTERVAL(*Q*)	*None*
MULTISET	*None*
MULTISET LOCATOR	PICTURE S9(*PI*) USAGE BINARY, where *PI* is implementation-defined

SQL Data Type	**COBOL Data Type**
NUMERIC(*P*,*S*)	USAGE DISPLAY SIGN LEADING SEPARATE, with PICTURE as specified[1]
REAL	*None*
REF	alphanumeric, with length *L*
ROW	*None*
SMALLINT	PICTURE S9(*SPI*) USAGE BINARY, where *SPI* is implementation-defined
TIME(*T*)	*None*
TIMESTAMP(*T*)	*None*
USER-DEFINED TYPE	*None*
USER-DEFINED TYPE LOCATOR	PICTURE S9(*PI*) USAGE BINARY, where *PI* is implementation-defined

[1] Case:

1) If *S* = *P*, then a PICTURE with an 'S' followed by a 'V' followed by *P* '9's.

2) If *P* > *S* > 0 (zero), then a PICTURE with an 'S' followed by *P*-*S* '9's followed by a 'V' followed by *S* '9's.

3) If *S* = 0 (zero), then a PICTURE with an 'S' followed by *P* '9's optionally followed by a 'V'.

Table 43 — SQL/CLI data type correspondences for Fortran

SQL Data Type	**Fortran Data Type**
ARRAY	*None*
ARRAY LOCATOR	INTEGER
BIGINT	*None*
BINARY (*L*)	CHARACTER, with length *L*
BINARY LARGE OBJECT (*L*)	CHARACTER, with length *L*
BINARY LARGE OBJECT LOCATOR	INTEGER
BINARY VARYING (*L*)	CHARACTER, with length *L*
BOOLEAN	LOGICAL

SQL Data Type	Fortran Data Type
CHARACTER (*L*)	CHARACTER, with length *L*
CHARACTER LARGE OBJECT (*L*)	CHARACTER, with length *L*
CHARACTER LARGE OBJECT LOCATOR	INTEGER
CHARACTER VARYING (*L*)	*None*
DATE	*None*
DECIMAL(*P*,*S*)	*None*
DOUBLE PRECISION	DOUBLE PRECISION
FLOAT(*P*)	*None*
INTEGER	INTEGER
INTERVAL(*Q*)	*None*
MULTISET	*None*
MULTISET LOCATOR	INTEGER
NUMERIC(*P*,*S*)	*None*
REAL	REAL
REF	CHARACTER, with length *L*
ROW	*None*
SMALLINT	*None*
TIME(*T*)	*None*
TIMESTAMP(*T*)	*None*
USER-DEFINED TYPE	*None*
USER-DEFINED TYPE LOCATOR	INTEGER

Table 44 — SQL/CLI data type correspondences for M

SQL Data Type	M Data Type
ARRAY	*None*

SQL Data Type	**M Data Type**
ARRAY LOCATOR	character
BIGINT	*None*
BINARY (*L*)	character
BINARY LARGE OBJECT (*L*)	character
BINARY LARGE OBJECT LOCA-TOR	character
BINARY VARYING (*L*)	character
BOOLEAN	*None*
CHARACTER (*L*)	*None*
CHARACTER LARGE OBJECT (*L*)	character
CHARACTER LARGE OBJECT LOCATOR	character
CHARACTER VARYING (*L*)	character with maximum length *L*
DATE	*None*
DECIMAL(*P*,*S*)	character
DOUBLE PRECISION	*None*
FLOAT(*P*)	*None*
INTEGER	character
INTERVAL(*Q*)	*None*
MULTISET	*None*
MULTISET LOCATOR	character
NUMERIC(*P*,*S*)	character
REAL	character
REF	character
ROW	*None*
SMALLINT	*None*

SQL Data Type	M Data Type
TIME(*T*)	*None*
TIMESTAMP(*T*)	*None*
USER-DEFINED TYPE	*None*
USER-DEFINED TYPE LOCATOR	character

Table 45 — SQL/CLI data type correspondences for Pascal

SQL Data Type	Pascal Data Type
ARRAY	*None*
ARRAY LOCATOR	INTEGER
BIGINT	*None*
BINARY (*L*)	PACKED ARRAY[1..*L*] OF CHAR
BINARY LARGE OBJECT (*L*), *L* > 1 (one)	PACKED ARRAY[1..*L*] OF CHAR
BINARY LARGE OBJECT LOCATOR	INTEGER
BINARY VARYING (*L*)	PACKED ARRAY[1..*L*] OF CHAR
BOOLEAN	BOOLEAN
CHARACTER (1)	CHAR
CHARACTER (*L*), *L* > 1 (one)	PACKED ARRAY[1..*L*] OF CHAR
CHARACTER LARGE OBJECT (*L*), *L* > 1 (one)	PACKED ARRAY[1..*L*] OF CHAR
CHARACTER LARGE OBJECT LOCATOR	INTEGER
CHARACTER VARYING (*L*)	*None*
DATE	*None*
DECIMAL(*P*,*S*)	*None*
DOUBLE PRECISION	*None*

SQL Data Type	Pascal Data Type
FLOAT(*P*)	*None*
INTEGER	INTEGER
INTERVAL(*Q*)	*None*
MULTISET	*None*
MULTISET LOCATOR	INTEGER
NUMERIC(*P*,*S*)	*None*
REAL	REAL
REF, *L* > 1 (one)	PACKED ARRAY[1..*L*] OF CHAR
ROW	*None*
SMALLINT	*None*
TIME(*T*)	*None*
TIMESTAMP(*T*)	*None*
USER-DEFINED TYPE	*None*
USER-DEFINED TYPE LOCATOR	INTEGER

Table 46 — SQL/CLI data type correspondences for PL/I

SQL Data Type	PL/I Data Type
ARRAY	*None*
ARRAY LOCATOR	FIXED BINARY(*PI*), where *PI* is implementation-defined
BIGINT	FIXED BINARY(*BPI*), where *BPI* is implementation-defined
BINARY (*L*)	CHARACTER(*L*)
BINARY LARGE OBJECT (*L*)	CHARACTER VARYING(*L*)
BINARY LARGE OBJECT LOCATOR	FIXED BINARY(*PI*), where *PI* is implementation-defined
BINARY VARYING (*L*)	CHARACTER(*L*) VARYING
BOOLEAN	BIT(1)

SQL Data Type	**PL/I Data Type**
CHARACTER (*L*)	CHARACTER(*L*)
CHARACTER LARGE OBJECT (*L*)	CHARACTER VARYING(*L*)
CHARACTER LARGE OBJECT LOCATOR	FIXED BINARY(*PI*), where *PI* is implementation-defined
CHARACTER VARYING (*L*)	CHARACTER VARYING(*L*)
DATE	*None*
DECIMAL(*P*,*S*)	FIXED DECIMAL(*P*,*S*)
DOUBLE PRECISION	*None*
FLOAT(*P*)	FLOAT BINARY (*P*)
INTEGER	FIXED BINARY(*PI*), where *PI* is implementation-defined
INTERVAL(*Q*)	*None*
MULTISET	*None*
MULTISET LOCATOR	FIXED BINARY(*PI*), where *PI* is implementation-defined
NUMERIC(*P*,*S*)	*None*
REAL	*None*
REF	CHARACTER VARYING (*L*)
ROW	*None*
SMALLINT	FIXED BINARY(*SPI*), where *SPI* is implementation-defined
TIME(*T*)	*None*
TIMESTAMP(*T*)	*None*
USER-DEFINED TYPE LOCATOR	*None*
USER-DEFINED TYPE	FIXED BINARY(*PI*), where *PI* is implementation-defined

6 SQL/CLI routines

This Clause is modified by Clause 20, “SQL/CLI routines”, in ISO/IEC 9075-9.

Subclause 5.1, “<CLI routine>”, defines a generic CLI routine. This Subclause describes the individual CLI routines in alphabetical order.

For convenience, the variable <CLI name prefix> is omitted and the <CLI generic name> is used for the descriptions. For presentation purposes (and purely arbitrarily), the routines are presented as functions rather than as procedures.

6.1 AllocConnect

Function

Allocate an SQL-connection and assign a handle to it.

Definition

```
AllocConnect (
    EnvironmentHandle     IN        INTEGER,
    ConnectionHandle      OUT       INTEGER )
    RETURNS SMALLINT
```

General Rules

1) Let *EH* be the value of EnvironmentHandle.

2) AllocHandle is implicitly invoked with HandleType indicating CONNECTION HANDLE, with *EH* as the value of InputHandle and with ConnectionHandle as OutputHandle.

6.2 AllocEnv

Function

Allocate an SQL-environment and assign a handle to it.

Definition

```
AllocEnv (
    EnvironmentHandle      OUT       INTEGER )
    RETURNS SMALLINT
```

General Rules

1) AllocHandle is implicitly invoked with HandleType indicating ENVIRONMENT HANDLE, with zero as the value of InputHandle, and with EnvironmentHandle as OutputHandle.

6.3 AllocHandle

Function

Allocate a resource and assign a handle to it.

Definition

```
AllocHandle (
    HandleType        IN        SMALLINT,
    InputHandle       IN        INTEGER,
    OutputHandle      OUT       INTEGER )
    RETURNS SMALLINT
```

General Rules

1) Let *HT* be the value of HandleType and let *IH* be the value of InputHandle.

2) If *HT* is not one of the code values in Table 14, "Codes used for SQL/CLI handle types", then an exception condition is raised: *CLI-specific condition — invalid handle*.

3) Case:

   a) If *HT* indicates ENVIRONMENT HANDLE, then:

      i) If the maximum number of SQL-environments that can be allocated at one time has already been reached, then an exception condition is raised: *CLI-specific condition — limit on number of handles exceeded*. A skeleton SQL-environment is allocated and is assigned a unique value that is returned in OutputHandle.

      ii) Case:

         1) If the memory requirements to manage an SQL-environment cannot be satisfied, then OutputHandle is set to zero and an exception condition is raised: *CLI-specific condition — memory allocation error*.

            NOTE 13 — No diagnostic information is generated in this case as there is no valid environment handle that can be used in order to obtain diagnostic information.

         2) If the resources to manage an SQL-environment cannot be allocated for implementation-defined reasons, then an implementation-defined exception condition is raised. A skeleton SQL-environment is allocated and is assigned a unique value that is returned in OutputHandle.

         3) Otherwise, the resources to manage an SQL-environment are allocated and are referred to as an allocated SQL-environment. The allocated SQL-environment is assigned a unique value that is returned in OutputHandle.

   b) If *HT* indicates CONNECTION HANDLE, then:

      i) If *IH* does not identify an allocated SQL-environment or if it identifies an allocated skeleton SQL-environment, then OutputHandle is set to zero and an exception condition is raised: *CLI-specific condition — invalid handle*.

ii) Let *E* be the allocated SQL-environment identified by *IH*.

iii) The diagnostics area associated with *E* is emptied.

iv) If the maximum number of SQL-connections that can be allocated at one time has already been reached, then OutputHandle is set to zero and an exception condition is raised: *CLI-specific condition — limit on number of handles exceeded*.

v) Case:

1) If the memory requirements to manage an SQL-connection cannot be satisfied, then OutputHandle is set to zero and an exception condition is raised: *CLI-specific condition — memory allocation error*.

2) If the resources to manage an SQL-connection cannot be allocated for implementation-defined reasons, then OutputHandle is set to zero and an implementation-defined exception condition is raised.

3) Otherwise, the resources to manage an SQL-connection are allocated and are referred to as an *allocated SQL-connection*. The allocated SQL-connection is associated with *E* and is assigned a unique value that is returned in OutputHandle.

c) If *HT* indicates STATEMENT HANDLE, then:

i) If *IH* does not identify an allocated SQL-connection, then OutputHandle is set to zero and an exception condition is raised: *CLI-specific condition — invalid handle*.

ii) Let *C* be the allocated SQL-connection identified by *IH*.

iii) The diagnostics area associated with *C* is emptied.

iv) If there is no established SQL-connection associated with *C*, then OutputHandle is set to zero and an exception condition is raised: *connection exception — connection does not exist*. Otherwise, let *EC* be the established SQL-connection associated with *C*.

v) If the maximum number of SQL-statements that can be allocated at one time has already been reached, then OutputHandle is set to zero and an exception condition is raised: *CLI-specific condition — limit on number of handles exceeded*.

vi) If *EC* is not the current SQL-connection, then the General Rules of Subclause 5.3, "Implicit set connection", are applied with EC as *dormant SQL-connection*.

vii) If the memory requirements to manage an SQL-statement cannot be satisfied, then OutputHandle is set to zero and an exception condition is raised: *CLI-specific condition — memory allocation error*.

viii) If the resources to manage an SQL-statement cannot be allocated for implementation-defined reasons, then OutputHandle is set to zero and an implementation-defined exception condition is raised.

ix) The resources to manage an SQL-statement are allocated and are referred to as an *allocated SQL-statement*. The allocated SQL-statement is associated with *C* and is assigned a unique value that is returned in OutputHandle.

x) The following CLI descriptor areas are automatically allocated and associated with the allocated SQL-statement:

1) An implementation parameter descriptor.

2) An implementation row descriptor.

3) An application parameter descriptor.

4) An application row descriptor.

For each of these descriptor areas, the ALLOC_TYPE field is set to indicate AUTOMATIC. For each of these descriptor areas, fields with non-blank entries in Table 24, “SQL/CLI descriptor field default values”, are set to the specified default values. All other fields in the CLI item descriptor areas are initially undefined.

xi) The automatically allocated application parameter descriptor becomes the current application parameter descriptor for the allocated SQL-statement and the automatically allocated application row descriptor becomes the current application row descriptor for the allocated SQL-statement.

d) If *HT* indicates DESCRIPTOR HANDLE, then:

i) If *IH* does not identify an allocated SQL-connection then OutputHandle is set to zero and an exception condition is raised: *CLI-specific condition — invalid handle*.

ii) Let *C* be the allocated SQL-connection identified by *IH*.

iii) The diagnostics area associated with *C* is emptied.

iv) If there is no established SQL-connection associated with *C*, then OutputHandle is set to zero and an exception condition is raised: *connection exception — connection does not exist*. Otherwise, let *EC* be the established SQL-connection associated with *C*.

v) If the maximum number of CLI descriptor areas that can be allocated at one time has already been reached, then OutputHandle is set to zero and an exception condition is raised: *CLI-specific condition — limit on number of handles exceeded*.

vi) If *EC* is not the current SQL-connection, then the General Rules of Subclause 5.3, “Implicit set connection”, are applied with EC as *dormant SQL-connection*.

vii) Case:

1) If the memory requirements to manage a CLI descriptor area cannot be satisfied, then OutputHandle is set to zero and an exception condition is raised: *CLI-specific condition — memory allocation error*.

2) If the resources to manage a CLI descriptor area cannot be allocated for implementation-defined reasons, then OutputHandle is set to zero and an implementation-defined exception condition is raised.

3) Otherwise, the resources to manage a CLI descriptor area are allocated and are referred to as an allocated CLI descriptor area. The allocated CLI descriptor area is associated with *C* and is assigned a unique value that is returned in OutputHandle. The ALLOC_TYPE field of the allocated CLI descriptor area is set to indicate USER. Other fields of the allocated CLI descriptor area are set to the default values for an ARD specified in Table 24, “SQL/CLI descriptor field default values”. Fields in the CLI item descriptor areas not set to a default value are initially undefined.

6.4 AllocStmt

Function

Allocate an SQL-statement and assign a handle to it.

Definition

```
AllocStmt (
    ConnectionHandle      IN       INTEGER,
    StatementHandle       OUT      INTEGER )
    RETURNS SMALLINT
```

General Rules

1) Let *CH* be the value of ConnectionHandle.

2) AllocHandle is implicitly invoked with HandleType indicating STATEMENT HANDLE, with *CH* as the value of InputHandle, and with StatementHandle as OutputHandle.

6.5 BindCol

Function

Describe a target specification or array of target specifications.

Definition

```
BindCol (
    StatementHandle      IN        INTEGER,
    ColumnNumber         IN        SMALLINT,
    TargetType           IN        SMALLINT,
    TargetValue          DEFOUT    ANY,
    BufferLength         IN        INTEGER,
    StrLen_or_Ind        DEFOUT    INTEGER )
    RETURNS SMALLINT
```

General Rules

1) Let *S* be the allocated SQL-statement identified by StatementHandle.

2) Let *HV* be the value of the handle of the current application row descriptor for *S*.

3) Let *ARD* be the allocated CLI descriptor area identified by *HV* and let *N* be the value of the TOP_LEVEL_COUNT field of *ARD*.

4) Let *CN* be the value of ColumnNumber.

5) If *CN* is less than 1 (one), then an exception condition is raised: *dynamic SQL error — invalid descriptor index*.

6) If *CN* is greater than *N*, then

   Case:

   a) If the memory requirements to manage the larger *ARD* cannot be satisfied, then an exception condition is raised: *CLI-specific condition — memory allocation error*.

   b) Otherwise, the TOP_LEVEL_COUNT field of *ARD* is set to *CN* and the COUNT field of *ARD* is incremented by 1 (one).

7) Let *TT* be the value of TargetType.

8) Let *HL* be the programming language of the invoking host program. Let *operative data type correspondence table* be the data type correspondence table for *HL* as specified in Subclause 5.15, "SQL/CLI data type correspondences". Refer to the two columns of the operative data type correspondences table as the *SQL data type column* and the *host data type column*.

9) If either of the following is true, then an exception condition is raised: *CLI-specific condition — invalid data type in application descriptor*.

   a) *TT* does not indicate DEFAULT and is not one of the code values in Table 8, "Codes used for application data types in SQL/CLI".

b) *TT* is one of the code values in Table 8, “Codes used for application data types in SQL/CLI”, but the row that contains the corresponding SQL data type in the SQL data type column of the operative data type correspondence table contains 'None' in the host data type column.

10) Let *BL* be the value of BufferLength.

11) If *BL* is not greater than zero, then an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

12) Let *IDA* be the item descriptor area of *ARD* specified by *CN*.

13) If an exception condition is raised in any of the following General Rules, then the TYPE, OCTET_LENGTH, LENGTH, DATA_POINTER, INDICATOR_POINTER, and OCTET_LENGTH_POINTER fields of *IDA* are set to implementation-dependent values and the value of COUNT for *ARD* is unchanged.

14) The data type of the <target specification> described by *IDA* is set to *TT*.

15) The length in octets of the <target specification> described by *IDA* is set to *BL*.

16) The length in characters or positions of the <target specification> described by *IDA* is set to the maximum number of characters or positions that may be represented by the data type *TT*.

17) The address of the host variable or array of host variables that is to receive a value or values for the <target specification> or <target specification>s described by *IDA* is set to the address of TargetValue. If TargetValue is a null pointer, then the address is set to 0 (zero).

18) The address of the <indicator variable> or array of <indicator variable>s associated with the host variable or host variables addressed by the DATA_POINTER field of *IDA* is set to the address of StrLen_or_Ind.

19) The address of the host variable or array of host variables that is to receive the returned length (in characters) of the <target specification> or <target specification>s described by *IDA* is set to the address of StrLen_or_Ind.

20) Restrictions on the differences allowed between *ARD* and *IRD* are implementation-defined, except as specified in the General Rules of Subclause 5.8, “Implicit FETCH USING clause”, and the General Rules of Subclause 6.30, “GetData”.

6.6 BindParameter

Function

Describe a dynamic parameter specification and its value.

Definition

```
BindParameter (
    StatementHandle      IN        INTEGER,
    ParameterNumber      IN        SMALLINT,
    InputOutputMode      IN        SMALLINT,
    ValueType            IN        SMALLINT,
    ParameterType        IN        SMALLINT,
    ColumnSize           IN        INTEGER,
    DecimalDigits        IN        SMALLINT,
    ParameterValue       DEF       ANY,
    BufferLength         IN        INTEGER,
    StrLen_or_Ind        DEF       INTEGER )
    RETURNS SMALLINT
```

General Rules

1) Let *S* be the allocated SQL-statement identified by StatementHandle.

2) Let *HV* be the value of the handle of the current application parameter descriptor for *S*.

3) Let *APD* be the allocated CLI descriptor area identified by *HV* and let *N2* be the value of the TOP_LEVEL_COUNT field of *APD*.

4) Let *PN* be the value of ParameterNumber.

5) If *PN* is less than 1 (one), then an exception condition is raised: *dynamic SQL error — invalid descriptor index*.

6) Let *IOM* be the value of InputOutputMode.

7) If *IOM* is not one of the code values in Table 11, “Codes associated with <parameter mode> in SQL/CLI”, then an exception condition is raised: *CLI-specific condition — invalid parameter mode*.

8) Let *VT* be the value of ValueType.

9) Let *HL* be the programming language of the invoking host program. Let *operative data type correspondence table* be the data type correspondence table for *HL* as specified in Subclause 5.15, “SQL/CLI data type correspondences”. Refer to the two columns of the operative data type correspondence table as the *SQL data type column* and the *host data type column*.

10) If any of the following are true, then an exception condition is raised: *CLI-specific condition — invalid data type in application descriptor*.

    a) *VT* does not indicate DEFAULT and is not one of the code values in Table 8, “Codes used for application data types in SQL/CLI”.

b) *VT* is one of the code values in Table 8, “Codes used for application data types in SQL/CLI”, but the row that contains the corresponding SQL data type in the SQL data type column of the operative data type correspondence table contains 'None' in the host data type column.

11) Let *PT* be the value of ParameterType.

12) If *PT* is not one of the code values in Table 33, “Codes used for concise data types”, then an exception condition is raised: *CLI-specific condition — invalid data type*.

13) Let *IPD* be the implementation parameter descriptor associated with *S* and let *N1* be the value of the TOP_LEVEL_COUNT field of *IPD*.

14) If *PN* is greater than *N1*, then

Case:

a) If the memory requirements to manage the larger *IPD* cannot be satisfied, then an exception condition is raised: *CLI-specific condition — memory allocation error*.

b) Otherwise, the TOP_LEVEL_COUNT field of *IPD* is set to *PN* and the COUNT field of *APD* is incremented by 1 (one).

15) If *PN* is greater than *N2*, then

Case:

a) If the memory requirements to manage the larger *APD* cannot be satisfied, then an exception condition is raised: *CLI-specific condition — memory allocation error*.

b) Otherwise, the TOP_LEVEL_COUNT field of *APD* is set to *PN* and the COUNT field of *APD* is incremented by 1 (one).

16) Let *IDA1* be the item descriptor area of *IPD* specified by *PN*.

17) Let *CS* be the value of ColumnSize, let *DD* be the value of DecimalDigits, and let *BL* be the value of BufferLength.

18) Case:

a) If *PT* is one of the values listed in Table 34, “Codes used with concise datetime data types in SQL/CLI”, then:

i) The data type of the <dynamic parameter specification> described by *IDA1* is set to a code shown in the Data Type Code column of Table 34, “Codes used with concise datetime data types in SQL/CLI”, indicating the concise data type code.

ii) The datetime interval code of the <dynamic parameter specification> described by *IDA1* is set to a code shown in the Datetime Interval Code column in Table 34, “Codes used with concise datetime data types in SQL/CLI”, indicating the concise data type code.

iii) The length (in positions) of the <dynamic parameter specification> described by *IDA1* is set to *CS*.

iv) Case:

1) If the datetime interval code of the <dynamic parameter specification> indicates DATE, then the time fractional seconds precision of the <dynamic parameter specification> described by *IDA1* is set to zero.

2) Otherwise, the time fractional seconds precision of the <dynamic parameter specification> described by *IDA1* is set to *DD*.

b) If *PT* is one of the values listed in Table 35, “Codes used with concise interval data types in SQL/CLI”, then:

i) The data type of the <dynamic parameter specification> described by *IDA1* is set to a code shown in the Data Type Code column of Table 35, “Codes used with concise interval data types in SQL/CLI”, indicating the concise data type code.

ii) The datetime interval code of the <dynamic parameter specification> described by *IDA1* is set to a code shown in the Datetime Interval Code column in Table 35, “Codes used with concise interval data types in SQL/CLI”, indicating the concise data type code. Let *DIC* be that code.

iii) The length (in positions) of the <dynamic parameter specification> described by *IDA1* is set to *CS*.

iv) Let *LS* be 0 (zero).

v) If *IOM* is PARAM MODE IN or PARAM MODE INOUT, ParameterValue is not a null pointer, and *BL* is greater than zero, then:

1) Let *PV* be the value of ParameterValue.

2) Let *FC* be the value of

```
SUBSTR ( PV FROM 1 FOR 1 )
```

3) If *FC* is <plus sign> or <minus sign>, then let *LS* be 1 (one).

vi) Case:

1) If *DIC* indicates SECOND, DAY TO SECOND, HOUR TO SECOND, or MINUTE TO SECOND, then the interval fractional seconds precision of the <dynamic parameter specification> described by *IDA1* is set to *DD*. If *DD* is 0 (zero), then let *DP* be 0 (zero); otherwise, let *DP* be 1 (one).

2) Otherwise, the interval fractional seconds precision of the <dynamic parameter specification> described by *IDA1* is set to zero.

vii) Case:

1) If *DIC* indicates YEAR TO MONTH, DAY TO HOUR, HOUR TO MINUTE or MINUTE TO SECOND, then let *IL* be 3.

2) If *DIC* indicates DAY TO MINUTE or HOUR TO SECOND, then let *IL* be 6.

3) If *DIC* indicates DAY TO SECOND, then let *IL* be 9.

4) Otherwise, let *IL* be zero.

viii) Case:

1) If *DIC* indicates SECOND, DAY TO SECOND, HOUR TO SECOND, or MINUTE TO SECOND, then the interval leading field precision of the <dynamic parameter specification> described by *IDA1* is set to *CS*–*IL*–*DD*–*DP*–*LS*.

2) Otherwise, the interval leading field precision of the <dynamic parameter specification> described by *IDA1* is set to *CS–IL–LS*.

c) Otherwise:

i) The data type of the <dynamic parameter specification> described by *IDA1* is set to *PT*.

ii) If *PT* indicates a character string type, then the length (in characters) of the <dynamic parameter specification> described by *IDA1* is set to *CS*.

iii) If *PT* indicates a numeric type, then the precision of the <dynamic parameter specification> described by *IDA1* is set to *CS*.

iv) If *PT* indicates a numeric type, then the scale of the <dynamic parameter specification> described by *IDA1* is set to *DD*.

19) Let *IDA2* be the item descriptor area of *APD* specified by *PN*.

20) If an exception condition is raised in any of the following General Rules, then:

a) The TYPE, LENGTH, PRECISION, and SCALE fields of *IDA1* are set to implementation-dependent values and the values of the TOP_LEVEL_COUNT and COUNT fields of *IPD* are unchanged.

b) The TYPE, DATA_POINTER, INDICATOR_POINTER, and OCTET_LENGTH_POINTER fields of *IDA2* are set to implementation-dependent values and the values of the TOP_LEVEL_COUNT and COUNT fields of *APD* are unchanged.

21) The parameter mode of the <dynamic parameter specification> described by *IDA2* is set to *IOM*.

22) The data type of the <dynamic parameter specification> described by *IDA2* is set to *VT*.

23) The address of the host variable that is to provide a value for the <dynamic parameter specification> value described by *IDA2* is set to the address of ParameterValue. If ParameterValue is a null pointer, then the address is set to 0 (zero).

24) The address of the <indicator variable> associated with the host variable addressed by the DATA_POINTER field of *IDA2* is set to the address of StrLen_or_Ind.

25) The address of the host variable that is to define the length (in octets) of the <dynamic parameter specification> value described by *IDA2* is set to the address of StrLen_or_Ind.

26) If *IOM* is PARAM MODE OUT or PARAM MODE INOUT and *BL* is not greater than zero, then an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

27) The length in octets of the <dynamic parameter specification> value described by *IDA2* is set to *BL*.

28) If *IOM* is PARAM MODE IN or PARAM MODE INOUT, ParameterValue is not a null pointer, and *BL* is greater than 0 (zero), then let *PV* be the value of the <dynamic parameter specification> value described by *IDA2*.

29) Restrictions on the differences allowed between *APD* and *IPD* are implementation-defined, except as specified in the General Rules of Subclause 5.6, “Implicit EXECUTE USING and OPEN USING clauses”, Subclause 5.7, “Implicit CALL USING clause”, and the General Rules of Subclause 6.49, “ParamData”.

6.7 Cancel

Function

Attempt to cancel execution of a CLI routine.

Definition

```
Cancel (
    StatementHandle IN   INTEGER )
    RETURNS SMALLINT
```

General Rules

1) Let *S* be the allocated SQL-statement identified by StatementHandle.

2) Case:

   a) If there is a CLI routine concurrently operating on *S*, then:

      i) Let *RN* be the routine name of the concurrent CLI routine.

      ii) Let *C* be the allocated SQL-connection with which *S* is associated.

      iii) Let *EC* be the established SQL-connection associated with *C* and let *SS* be the SQL-server associated with *EC*.

      iv) *SS* is requested to cancel the execution of *RN*.

      v) If *SS* rejects the cancellation request, then an exception condition is raised: *CLI-specific condition — server declined the cancellation request*.

      vi) If *SS* accepts the cancellation request, then a completion condition is raised: *successful completion*.

      NOTE 14 — Acceptance of the request does not guarantee that the execution of *RN* will be cancelled.

      vii) If *SS* succeeds in canceling the execution of *RN*, then an exception condition is raised for *RN*: *CLI-specific condition — operation canceled*.

      NOTE 15 — Canceling the execution of *RN* does not destroy any diagnostic information already generated by its execution.

   NOTE 16 — The method of passing control between concurrently operating programs is implementation-dependent.

   b) If there is a deferred parameter number associated with *S*, then:

      i) The diagnostics area associated with *S* is emptied.

      ii) The deferred parameter number is removed from association with *S*.

      iii) Any statement source associated with *S* is removed from association with *S*.

   c) Otherwise:

      i) The diagnostics area associated with *S* is emptied.

ii) A completion condition is raised: *successful completion*.

6.8 CloseCursor

Function

Close a cursor.

Definition

```
CloseCursor (
    StatementHandle     IN   INTEGER )
    RETURNS SMALLINT
```

General Rules

1) Let *S* be the allocated SQL-statement identified by StatementHandle.

2) If there is no executed statement associated with *S*, then an exception condition is raised: *CLI-specific condition — function sequence error*.

3) Case:

   a) If there is no open cursor associated with *S*, then an exception condition is raised: *invalid cursor state*.

   b) Otherwise:

      i) The open cursor associated with *S* is placed in the closed state and its copy of the select source is destroyed.

      ii) Any fetched row associated with *S* is removed from association with *S*.

6.9 ColAttribute

Function

Get a column attribute.

Definition

```
ColAttribute (
    StatementHandle     IN   INTEGER,
    ColumnNumber        IN   SMALLINT,
    FieldIdentifier     IN   SMALLINT,
    CharacterAttribute  OUT  CHARACTER(L),
    BufferLength        IN   SMALLINT,
    StringLength        OUT  SMALLINT,
    NumericAttribute    OUT  INTEGER )
    RETURNS SMALLINT
```

where *L* has a maximum value equal to the implementation-defined maximum length of a variable-length character string.

General Rules

1) Let *S* be the allocated SQL-statement identified by StatementHandle.

2) If there is no prepared or executed statement associated with *S*, then an exception condition is raised: *CLI-specific condition — function sequence error*.

3) Let *IRD* be the implementation row descriptor associated with *S* and let *N* be the value of the TOP_LEVEL_COUNT field of *IRD*.

4) Let *FI* be the value of FieldIdentifier.

5) If *FI* is not one of the code values in Table 21, “Codes used for SQL/CLI descriptor fields”, then an exception condition is raised: *CLI-specific condition — invalid descriptor field identifier*.

6) Let *CN* be the value of ColumnNumber.

7) Let *TYPE* be the value of the Type column in the row of Table 21, “Codes used for SQL/CLI descriptor fields”, that contains *FI*.

8) Let *FDT* be the value of the Data Type column in the row of Table 6, “Fields in SQL/CLI row and parameter descriptor areas”, whose Field column contains the value of the Field column in the row of Table 21, “Codes used for SQL/CLI descriptor fields”, that contains *FI*.

9) If *TYPE* is 'ITEM', then:

   a) If *N* is zero, then an exception condition is raised: *dynamic SQL error — prepared statement not a cursor specification*.

   b) If *CN* is less than 1 (one), then an exception condition is raised: *dynamic SQL error — invalid descriptor index*.

c) If *CN* is greater than *N*, then a completion condition is raised: *no data*.

d) Let *IDA* be the item descriptor area of *IRD* specified by the *CN*-th descriptor area in *IRD* for which LEVEL is 0 (zero).

e) Let *DT* and *DIC* be the values of the TYPE and DATETIME_INTERVAL_CODE fields, respectively, for *IDA*.

10) If TYPE is 'HEADER', then:

a) If *CN* is less than 1 (one), then an exception condition is raised: *dynamic SQL error — invalid descriptor index*.

b) If *CN* is greater than *N*, then a completion condition is raised: *no data*.

c) Let *CN* be 0 (zero).

11) Let *DH* be the handle that identifies *IRD*.

12) Let *RI* be the number of the descriptor record in *IRD* that is the *CN*-th descriptor area for which LEVEL is 0 (zero).

Case:

a) If *FDT* indicates character string, then let the information be retrieved from *IRD* by implicitly executing GetDescField as follows:

```
GetDescField ( DH, RI, FI,
               CharacterAttribute, BufferLength, StringLength )
```

b) Otherwise,

Case:

i) If *FI* indicates TYPE, then

Case:

1) If *DT* indicates a <datetime type>, then NumericAttribute is set to the concise code value corresponding to the datetime interval code value *DIC* as defined in Table 36, “Concise codes used with datetime data types in SQL/CLI”.

2) If *DT* indicates INTERVAL, then NumericAttribute is set to the concise code value corresponding to the datetime interval code value *DIC* as defined in Table 37, “Concise codes used with interval data types in SQL/CLI”.

3) Otherwise, NumericAttribute is set to *DT*.

ii) Otherwise, let the information be retrieved from *IRD* by implicitly executing GetDescField as follows:

```
GetDescField ( DH, RI, FI,
    NumericAttribute, BufferLength, StringLength )
```

6.10 ColumnPrivileges

Function

Return a result set that contains a list of the privileges held on the columns whose names adhere to the requested pattern or patterns within a single specified table stored in the Information Schema of the connected data source.

Definition

```
ColumnPrivileges (
    StatementHandle       IN     INTEGER,
    CatalogName           IN     CHARACTER(L1),
    NameLength1           IN     SMALLINT,
    SchemaName            IN     CHARACTER(L2),
    NameLength2           IN     SMALLINT,
    TableName             IN     CHARACTER(L3),
    NameLength3           IN     SMALLINT,
    ColumnName            IN     CHARACTER(L4),
    NameLength4       IN     SMALLINT )
    RETURNS SMALLINT
```

where each of *L1*, *L2*, *L3*, and *L4* has a maximum value equal to the implementation-defined maximum length of a variable-length character string.

General Rules

1) Let *S* be the allocated SQL-statement identified by StatementHandle.

2) If an open cursor is associated with *S*, then an exception condition is raised: *invalid cursor state*.

3) Let *C* be the allocated SQL-connection with which *S* is associated.

4) Let *EC* be the established SQL-connection associated with *C* and let *SS* be the SQL-server on that connection.

5) Let *COLUMN_PRIVILEGES_QUERY* be a table, with the definition:

```
CREATE TABLE COLUMN_PRIVILEGES_QUERY (
    TABLE_CAT          CHARACTER VARYING(128),
    TABLE_SCHEM        CHARACTER VARYING(128) NOT NULL,
    TABLE_NAME         CHARACTER VARYING(128) NOT NULL,
    COLUMN_NAME        CHARACTER VARYING(128) NOT NULL,
    GRANTOR            CHARACTER VARYING(128),
    GRANTEE            CHARACTER VARYING(128) NOT NULL,
    PRIVILEGE          CHARACTER VARYING(128) NOT NULL,
    IS_GRANTABLE       CHARACTER VARYING(3) )
```

6) *COLUMN_PRIVILEGES_QUERY* contains a row for each privilege in *SS*'s Information Schema COLUMN_PRIVILEGES view where:

   a) Let *SUP* be the value of Supported that is returned by the execution of GetFeatureInfo with FeatureType = 'FEATURE' and FeatureId = 'C041' (corresponding to the feature “Information Schema metadata constrained by privileges”).

b) Case:

i) If the value of *SUP* is 1 (one), then *COLUMN_PRIVILEGES_QUERY* contains a row for each privilege in *SS*'s Information Schema COLUMN_PRIVILEGES view.

ii) Otherwise, *COLUMN_PRIVILEGES_QUERY* contains a row for each privilege in *SS*'s Information Schema COLUMN_PRIVILEGES view that meets implementation-defined authorization criteria.

7) For each row of *COLUMN_PRIVILEGES_QUERY*:

a) If the implementation does not support catalog names, then TABLE_CAT is the null value; otherwise, the value of TABLE_CAT in *COLUMN_PRIVILEGES_QUERY* is the value of the TABLE_CATALOG column in the COLUMN_PRIVILEGES view in the Information Schema.

b) The value of TABLE_SCHEM in *COLUMN_PRIVILEGES_QUERY* is the value of the TABLE_SCHEMA column in the COLUMN_PRIVILEGES view.

c) The value of TABLE_NAME in *COLUMN_PRIVILEGES_QUERY* is the value of the TABLE_NAME column in the COLUMN_PRIVILEGES view.

d) The value of COLUMN_NAME in *COLUMN_PRIVILEGES_QUERY* is the value of the COLUMN_NAME column in the COLUMN_PRIVILEGES view.

e) The value of GRANTOR in *COLUMN_PRIVILEGES_QUERY* is the value of the GRANTOR column in the COLUMN_PRIVILEGES view.

f) The value of GRANTEE in *COLUMN_PRIVILEGES_QUERY* is the value of the GRANTEE column in the COLUMN_PRIVILEGES view.

g) The value of PRIVILEGE in *COLUMN_PRIVILEGES_QUERY* is the value of the PRIVILEGE_TYPE column in the COLUMN_PRIVILEGES view.

h) The value of IS_GRANTABLE in *COLUMN_PRIVILEGES_QUERY* is the value of the IS_GRANTABLE column in the COLUMN_PRIVILEGES view.

8) Let *NL1*, *NL2*, *NL3*, and *NL4* be the values of NameLength1, NameLength2, NameLength3, and NameLength4, respectively.

9) Let *CATVAL*, *SCHVAL*, *TBLVAL*, and *COLVAL* be the values of CatalogName, SchemaName, TableName, and ColumnName, respectively.

10) If the METADATA ID attribute of *S* is TRUE, then:

a) If CatalogName is a null pointer and the value of the CATALOG NAME information type from Table 29, "Codes and data types for implementation information", is 'Y', then an exception condition is raised: *CLI-specific condition — invalid use of null pointer*.

b) If SchemaName is a null pointer or if ColumnName is a null pointer, then an exception condition is raised: *CLI-specific condition — invalid use of null pointer*.

11) If TableName is a null pointer, then an exception condition is raised: *CLI-specific condition — invalid use of null pointer*.

12) If CatalogName is a null pointer, then *NL1* is set to zero. If SchemaName is a null pointer, then *NL2* is set to zero. If TableName is a null pointer, then *NL3* is set to zero. If ColumnName is a null pointer, then *NL4* is set to zero.

13) Case:

   a) If *NL1* is not negative, then let *L* be *NL1*.

   b) If *NL1* indicates NULL TERMINATED, then let *L* be the number of octets of CatalogName that precede the implementation-defined null character that terminates a C character string.

   c) Otherwise, an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

   Let *CATVAL* be the first *L* octets of CatalogName.

14) Case:

   a) If *NL2* is not negative, then let *L* be *NL2*.

   b) If *NL2* indicates NULL TERMINATED, then let *L* be the number of octets of SchemaName that precede the implementation-defined null character that terminates a C character string.

   c) Otherwise, an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

   Let *SCHVAL* be the first *L* octets of SchemaName.

15) Case:

   a) If *NL3* is not negative, then let *L* be *NL3*.

   b) If *NL3* indicates NULL TERMINATED, then let *L* be the number of octets of TableName that precede the implementation-defined null character that terminates a C character string.

   c) Otherwise, an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

   Let *TBLVAL* be the first *L* octets of TableName.

16) Case:

   a) If *NL4* is not negative, then let *L* be *NL4*.

   b) If *NL4* indicates NULL TERMINATED, then let *L* be the number of octets of ColumnName that precede the implementation-defined null character that terminates a C character string.

   c) Otherwise, an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

   Let *COLVAL* be the first *L* octets of ColumnName.

17) Case:

   a) If the METADATA ID attribute of *S* is TRUE, then:

      i) Case:

         1) If the value of *NL1* is zero, then let *CATSTR* be a zero-length string.

         2) Otherwise,

            Case:

A) If `SUBSTRING(TRIM('`*`CATVAL`*`') FROM 1 FOR 1) = '"'` and if `SUB-STRING(TRIM('`*`CATVAL`*`') FROM CHAR_LENGTH(TRIM('`*`CATVAL`*`')) FOR 1) = '"'`, then let *TEMPSTR* be the value obtained from evaluating:

```
SUBSTRING(TRIM('CATVAL') FROM 2
FOR CHAR_LENGTH(TRIM('CATVAL')) - 2)
```

and let *CATSTR* be the character string:

```
TABLE_CAT = 'TEMPSTR' AND
```

B) Otherwise, let *CATSTR* be the character string:

```
UPPER(TABLE_CAT) = UPPER('CATVAL') AND
```

ii) Case:

1) If the value of *NL2* is zero, then let *SCHSTR* be a zero-length string.

2) Otherwise,

Case:

A) If `SUBSTRING(TRIM('`*`SCHVAL`*`') FROM 1 FOR 1) = '"'` and if `SUB-STRING(TRIM('`*`SCHVAL`*`') FROM CHAR_LENGTH(TRIM('`*`SCHVAL`*`')) FOR 1) = '"'`, then let *TEMPSTR* be the value obtained from evaluating:

```
SUBSTRING(TRIM('SCHVAL') FROM 2
  FOR CHAR_LENGTH(TRIM('SCHVAL')) - 2)
```

and let *SCHSTR* be the character string:

```
TABLE_SCHEM = 'TEMPSTR' AND
```

B) Otherwise, let *SCHSTR* be the character string:

```
UPPER(TABLE_SCHEM) = UPPER('SCHVAL') AND
```

iii) Case:

1) If the value of *NL3* is zero, then let *TBLSTR* be a zero-length string.

2) Otherwise,

Case:

A) If `SUBSTRING(TRIM('`*`TBLVAL`*`') FROM 1 FOR 1) = '"'` and if `SUB-STRING(TRIM('`*`TBLVAL`*`') FROM CHAR_LENGTH(TRIM('`*`TBLVAL`*`')) FOR 1) = '"'`, then let *TEMPSTR* be the value obtained from evaluating:

```
SUBSTRING(TRIM('TBLVAL') FROM 2
  FOR CHAR_LENGTH(TRIM('TBLVAL')) - 2)
```

and let *TBLSTR* be the character string:

```
TABLE_NAME = 'TEMPSTR' AND
```

B) Otherwise, let *TBLSTR* be the character string:

```
UPPER(TABLE_NAME) = UPPER('TBLVAL') AND
```

iv) Case:

1) If the value of *NL4* is zero, then let *COLSTR* be a zero-length string.

2) Otherwise,

Case:

A) If `SUBSTRING(TRIM('COLVAL') FROM 1 FOR 1) = '"'` and if `SUBSTRING(TRIM('COLVAL') FROM CHAR_LENGTH(TRIM('COLVAL')) FOR 1) = '"'`, then let *TEMPSTR* be the value obtained from evaluating:

```
SUBSTRING(TRIM('COLVAL') FROM 2
  FOR CHAR_LENGTH(TRIM('COLVAL')) - 2)
```

and let *COLSTR* be the character string:

```
COLUMN_NAME = 'TEMPSTR'
```

B) Otherwise, let *COLSTR* be the character string:

```
UPPER(COLUMN_NAME) = UPPER('COLVAL')
```

b) Otherwise,

i) Let *SPC* be the Code value from Table 29, "Codes and data types for implementation information", that corresponds to the Information Type SEARCH PATTERN ESCAPE in that same table.

ii) Let *ESC* be the value of InfoValue that is returned by the execution of GetInfo() with the value of InfoType set to *SPC*.

iii) If the value of *NL1* is zero, then let *CATSTR* be a zero-length string; otherwise, let *CATSTR* be the character string:

```
TABLE_CAT = 'CATVAL' AND
```

iv) If the value of *NL2* is zero, then let *SCHSTR* be a zero-length string; otherwise, let *SCHSTR* be the character string:

```
TABLE_SCHEM = 'SCHVAL' AND
```

v) If the value of *NL3* is zero, then let *TBLSTR* be a zero-length string; otherwise, let *TBLSTR* be the character string:

```
TABLE_NAME = 'TBLVAL' AND
```

vi) If the value of *NL4* is zero, then let *COLSTR* be a zero-length string; otherwise, let *COLSTR* be the character string:

```
COLUMN_NAME LIKE 'COLVAL' ESCAPE 'ESC' AND
```

18) Let *PRED* be the result of evaluating:

```
CATSTR || ' ' || SCHSTR || ' ' || TBLSTR || ' ' || COLSTR || ' ' || 1=1
```

19) Let *STMT* be the character string:

```
SELECT *
FROM COLUMN_PRIVILEGES_QUERY
WHERE PRED
ORDER BY TABLE_CAT, TABLE_SCHEM, TABLE_NAME, COLUMN_NAME, PRIVILEGE
```

20) ExecDirect is implicitly invoked with *S* as the value of StatementHandle, *STMT* as the value of StatementText, and the length of *STMT* as the value of TextLength.

6.11 Columns

Function

Based on the specified selection criteria, return a result set that contains information about columns of tables stored in the information schemas of the connected data source.

Definition

```
Columns (
    StatementHandle       IN     INTEGER,
    CatalogName           IN     CHARACTER(L1),
    NameLength1           IN     SMALLINT,
    SchemaName            IN     CHARACTER(L2),
    NameLength2           IN     SMALLINT,
    TableName             IN     CHARACTER(L3),
    NameLength3           IN     SMALLINT,
    ColumnName            IN     CHARACTER(L4),
    NameLength4           IN     SMALLINT )
    RETURNS SMALLINT
```

where each of *L1*, *L2*, *L3*, and *L4* has a maximum value equal to the implementation-defined maximum length of a variable-length character string.

General Rules

1) Let *S* be the allocated SQL-statement identified by StatementHandle.

2) If an open cursor is associated with *S*, then an exception condition is raised: *invalid cursor state*.

3) Let *C* be the allocated SQL-connection with which *S* is associated.

4) Let *EC* be the established SQL-connection associated with *C* and let *SS* be the SQL-server on that connection.

5) Let *COLUMNS_QUERY* be a table, with the definition:

```
CREATE TABLE COLUMNS_QUERY (
    TABLE_CAT          CHARACTER VARYING(128),
    TABLE_SCHEM        CHARACTER VARYING(128) NOT NULL,
    TABLE_NAME         CHARACTER VARYING(128) NOT NULL,
    COLUMN_NAME        CHARACTER VARYING(128) NOT NULL,
    DATA_TYPE          SMALLINT NOT NULL,
    TYPE_NAME          CHARACTER VARYING(128) NOT NULL,
    COLUMN_SIZE        INTEGER,
    BUFFER_LENGTH      INTEGER,
    DECIMAL_DIGITS     SMALLINT,
    NUM_PREC_RADIX     SMALLINT,
    NULLABLE           SMALLINT NOT NULL,
    REMARKS            CHARACTER VARYING(254),
    COLUMN_DEF         CHARACTER VARYING(254),
    SQL_DATA_TYPE      SMALLINT NOT NULL,
    SQL_DATETIME_SUB   INTEGER,
```

```
CHAR_OCTET_LENGTH  INTEGER,
ORDINAL_POSITION   INTEGER NOT NULL,
IS_NULLABLE        CHARACTER VARYING(254),
CHAR_SET_CAT       CHARACTER VARYING(128),
CHAR_SET_SCHEM     CHARACTER VARYING(128),
CHAR_SET_NAME      CHARACTER VARYING(128),
COLLATION_CAT      CHARACTER VARYING(128),
COLLATION_SCHEM    CHARACTER VARYING(128),
COLLATION_NAME     CHARACTER VARYING(128),
UDT_CAT            CHARACTER VARYING(128),
UDT_SCHEM          CHARACTER VARYING(128),
UDT_NAME           CHARACTER VARYING(128),
DOMAIN_CAT         CHARACTER VARYING(128),
DOMAIN_SCHEM       CHARACTER VARYING(128),
DOMAIN_NAME        CHARACTER VARYING(128),
SCOPE_CAT          CHARACTER VARYING(128),
SCOPE_SCHEM        CHARACTER VARYING(128),
SCOPE_NAME         CHARACTER VARYING(128),
MAX_CARDINALITY    INTEGER,
DTD_IDENTIFIER     CHARACTER VARYING(128),
IS_SELF_REF        CHARACTER VARYING(128),
UNIQUE (TABLE_CAT, TABLE_SCHEM, TABLE_NAME, COLUMN_NAME) )
```

6) *COLUMNS_QUERY* contains a row for each column described by *SS*'s Information Schema COLUMNS view where:

   a) Let *SUP* be the value of Supported that is returned by the execution of GetFeatureInfo with FeatureType = 'FEATURE' and FeatureId = 'C041' (corresponding to the feature “Information Schema metadata constrained by privileges”).

   b) Case:

      i) If the value of *SUP* is 1 (one), then *COLUMNS_QUERY* contains a row for each row describing a column in *SS*'s Information Schema COLUMNS view.

      ii) Otherwise, *COLUMNS_QUERY* contains a row for each row describing a column in *SS*'s Information Schema COLUMNS view that meets implementation-defined authorization criteria.

7) For each row of *COLUMNS_QUERY*:

   a) The value of TABLE_CAT in *COLUMNS_QUERY* is the value of the TABLE_CATALOG column in the COLUMNS view. If *SS* does not support catalog names, then TABLE_CAT is set to the null value.

   b) The value of TABLE_SCHEM in *COLUMNS_QUERY* is the value of the TABLE_SCHEMA column in the COLUMNS view.

   c) The value of TABLE_NAME in *COLUMNS_QUERY* is the value of the TABLE_NAME column in the COLUMNS view.

   d) The value of COLUMN_NAME in *COLUMNS_QUERY* is the value of the COLUMN_NAME column in the COLUMNS view.

   e) The value of DATA_TYPE in *COLUMNS_QUERY* is determined by the values of the DATA_TYPE and INTERVAL_TYPE columns in the COLUMNS view.

   Case:

i) If the value of DATA_TYPE in the COLUMNS view is 'INTERVAL', then the value of DATA_TYPE in *COLUMNS_QUERY* is the appropriate 'Code' from Table 33, “Codes used for concise data types”, that matches the interval specified in the INTERVAL_TYPE column in the COLUMNS view.

ii) Otherwise, the value of DATA_TYPE in *COLUMNS_QUERY* is the appropriate 'Code' from Table 33, “Codes used for concise data types”, that matches the value specified in the DATA_TYPE column in the COLUMNS view.

f) The value of TYPE_NAME in *COLUMNS_QUERY* is an implementation-defined value that is the character string by which the data type is known at the data source.

g) The value of COLUMN_SIZE in *COLUMNS_QUERY* is

Case:

i) If the value of DATA_TYPE in the COLUMNS view is 'CHARACTER', 'CHARACTER VARYING', 'CHARACTER LARGE OBJECT', 'BINARY', 'BINARY VARYING' or 'BINARY LARGE OBJECT', then the value is that of the CHARACTER_MAXIMUM_LENGTH in the same row of the COLUMNS view.

ii) If the value of DATA_TYPE in the COLUMNS view is 'DECIMAL' or 'NUMERIC', then the value is that of the NUMERIC_PRECISION column in the same row of the COLUMNS view.

iii) If the value of DATA_TYPE in the COLUMNS view is 'SMALLINT', 'INTEGER', 'BIGINT', 'FLOAT', 'REAL', or 'DOUBLE PRECISION', then the value is implementation-defined.

iv) If the value of DATA_TYPE in the COLUMNS view is 'DATE', 'TIME', 'TIMESTAMP', 'TIME WITH TIME ZONE', or 'TIMESTAMP WITH TIME ZONE', then the value of COLUMN_SIZE is that determined by SR 33), in Subclause 6.1, “<data type>”, in ISO/IEC 9075-2, where the value of <time fractional seconds precision> is the value of the DATETIME_PRECISION column in the same row of the COLUMNS view.

v) If the value of DATA_TYPE in the COLUMNS view is 'INTERVAL', then the value of COLUMN_SIZE is that determined by the General Rules of Subclause 10.1, “<interval qualifier>”, in ISO/IEC 9075-2, where:

1) The value of <interval qualifier> is the value of the INTERVAL_TYPE column in the same row of the COLUMNS view.

2) The value of <interval leading field precision> is the value of the INTERVAL_PRECISION column in the same row of the COLUMNS view.

3) The value of <interval fractional seconds precision> is the value of the NUMERIC_PRECISION column in the same row of the COLUMNS view.

vi) If the value of DATA_TYPE in the COLUMNS view is 'REF', then the value is the length in octets of the reference type.

vii) Otherwise, the value is implementation-dependent.

h) The value of BUFFER_LENGTH in COLUMNS_QUERY is implementation-defined.

NOTE 17 — The purpose of BUFFER_LENGTH in *COLUMNS_QUERY* is to record the number of octets transferred for the column with a Fetch routine, a FetchScroll routine, or a GetData routine when the TYPE field in the application row descriptor indicates DEFAULT. This length excludes any null terminator.

i) The value of DECIMAL_DIGITS in *COLUMNS_QUERY* is

Case:

i) If the value of DATA_TYPE in the COLUMNS view is one of 'DATE', 'TIME', 'TIMESTAMP', 'TIME WITH TIME ZONE', or 'TIMESTAMP WITH TIME ZONE', then the value of DECIMAL_DIGITS in *COLUMNS_QUERY* is the value of the DATETIME_PRECISION column in the COLUMNS view.

ii) If the value of DATA_TYPE in the COLUMNS view is one of 'NUMERIC', 'DECIMAL', 'SMALLINT', 'INTEGER', or 'BIGINT', then the value of DECIMAL_DIGITS in *COLUMNS_QUERY* is the value of the NUMERIC_SCALE column in the COLUMNS view.

iii) Otherwise, the value of DECIMAL_DIGITS in *COLUMNS_QUERY* is the null value.

j) The value of NUM_PREC_RADIX in *COLUMNS_QUERY* is the value of the NUMERIC_PRECISION_RADIX column in the COLUMNS view.

k) If the value of the IS_NULLABLE column in the COLUMNS view is 'NO', then the value of NULLABLE in COLUMNS_QUERY is set to the appropriate 'Code' for NO NULLS in Table 27, “Miscellaneous codes used in CLI”; otherwise it is set to the appropriate 'Code' for NULLABLE from Table 27, “Miscellaneous codes used in CLI”.

l) The value of REMARKS in *COLUMNS_QUERY* is an implementation-defined description of the column.

m) The value of COLUMN_DEF in *COLUMNS_QUERY* is the value of the COLUMN_DEFAULT column in the COLUMNS view.

n) The value of SQL_DATETIME_SUB in *COLUMNS_QUERY* is determined by the value of the DATA_TYPE column in the same row of the COLUMNS view.

Case:

i) If the value of DATA_TYPE in the COLUMNS view is the appropriate 'Code' for the any of the data types 'DATE', 'TIME', 'TIMESTAMP', 'TIME WITH TIME ZONE', or 'TIMESTAMP WITH TIME ZONE' from Table 33, “Codes used for concise data types”, then the value is the matching 'Datetime Interval Code' from Table 33, “Codes used for concise data types”.

ii) If the value of DATA_TYPE in the COLUMNS view is the appropriate 'Code' for any of the INTERVAL data types from Table 33, “Codes used for concise data types”, then the value is the matching 'Datetime Interval Code' from Table 33, “Codes used for concise data types”.

iii) Otherwise, the value is the null value.

o) The value of CHAR_OCTET_LENGTH in *COLUMNS_QUERY* is the value of the CHARACTER_OCTET_LENGTH column in the COLUMNS view.

p) The value of ORDINAL_POSITION in *COLUMNS_QUERY* is the value of the ORDINAL_POSITION column in the COLUMNS view.

q) The value of IS_NULLABLE in *COLUMNS_QUERY* is the value of the IS_NULLABLE column in the COLUMNS view.

r) The value of SQL_DATA_TYPE in *COLUMNS_QUERY* is determined by the value of the DATA_TYPE column in the same row of the COLUMNS view.

Case:

i) If the value of DATA_TYPE in the COLUMNS view is the appropriate 'Code' for any of the data types 'DATE', 'TIME', 'TIMESTAMP', 'TIME WITH TIME ZONE', or 'TIMESTAMP WITH TIME ZONE', from Table 33, “Codes used for concise data types”, then the value is the matching 'Code' from Table 7, “Codes used for implementation data types in SQL/CLI”.

ii) If the value of DATA_TYPE in the COLUMNS view is the appropriate 'Code' for any of the INTERVAL data types from Table 33, “Codes used for concise data types”, then the value is the matching 'Code' from Table 7, “Codes used for implementation data types in SQL/CLI”.

iii) Otherwise, the value is the same as the value of DATA_TYPE in COLUMNS_QUERY.

s) The value of CHAR_SET_CAT in *COLUMNS_QUERY* is the value of the CHARACTER_SET_CATALOG column in the COLUMNS view. If *SS* does not support catalog names, then CHAR_SET_CAT is set to the null value.

t) The value of CHAR_SET_SCHEM in *COLUMNS_QUERY* is the value of the CHARACTER_SET_SCHEMA column in the COLUMNS view.

u) The value of CHAR_SET_NAME in *COLUMNS_QUERY* is the value of the CHARACTER_SET_NAME column in the COLUMNS view.

v) The value of COLLATION_CAT in *COLUMNS_QUERY* is the value of the COLLATION_CATALOG column in the COLUMNS view. If *SS* does not support catalog names, then COLLATION_CAT is set to the null value.

w) The value of COLLATION _SCHEM in *COLUMNS_QUERY* is the value of the COLLATION_SCHEMA column in the COLUMNS view.

x) The value of COLLATION_NAME in *COLUMNS_QUERY* is the value of the COLLATION_NAME column in the COLUMNS view.

y) The value of UDT_CAT in *COLUMNS_QUERY* is the value of the USER_DEFINED_TYPE_CATALOG column in the COLUMNS view. If *SS* does not support catalog names, then UDT_CAT is set to the null value.

z) The value of UDT_SCHEM in *COLUMNS_QUERY* is the value of the USER_DEFINED_TYPE_SCHEMA column in the COLUMNS view.

aa) The value of UDT_NAME in *COLUMNS_QUERY* is the value of the USER_DEFINED_TYPE_NAME column in the COLUMNS view.

ab) The value of DOMAIN_CAT in COLUMNS_QUERY is the value of the DOMAIN_CATALOG column in the COLUMNS view. If *SS* does not support catalog names, then DOMAIN_CAT is set to the null value.

ac) The value of DOMAIN_SCHEM in COLUMNS_QUERY is the value of the DOMAIN_SCHEMA column in the COLUMNS view.

ad) The value of DOMAIN_NAME in COLUMNS_QUERY is the value of the DOMAIN_NAME column in the COLUMNS view.

ae) The value of SCOPE_CAT in COLUMNS_QUERY is the value of the SCOPE_CATALOG column in the COLUMNS view. If *SS* does not support catalog names, then SCOPE_CAT is set to the null value.

af) The value of SCOPE_SCHEM in COLUMNS_QUERY is the value of the SCOPE_SCHEMA column in the COLUMNS view.

ag) The value of SCOPE_NAME in COLUMNS_QUERY is the value of the SCOPE_NAME column in the COLUMNS view.

ah) The value of MAX_CARDINALITY in COLUMNS_QUERY is the value of the MAXIMUM_CARDINALITY column in the COLUMNS view.

ai) The value of DTD_IDENTIFIER in COLUMNS_QUERY is the value of the DTD_IDENTIFIER column in the COLUMNS view.

aj) The value of IS_SELF_REF in COLUMNS_QUERY is the value of the IS_SELF_REFERENCING column in the COLUMNS view.

8) Let *NL1*, *NL2*, *NL3*, and *NL4* be the values of NameLength1, NameLength2, NameLength3, and NameLength4, respectively.

9) Let *CATVAL*, *SCHVAL*, *TBLVAL*, and *COLVAL* be the values of CatalogName, SchemaName, TableName, and ColumnName, respectively.

10) If the METADATA ID attribute of *S* is TRUE, then:

a) If CatalogName is a null pointer and the value of the CATALOG NAME information type from Table 29, "Codes and data types for implementation information", is 'Y', then an exception condition is raised: *CLI-specific condition — invalid use of null pointer*.

b) If SchemaName is a null pointer, or if TableName is a null pointer, or if ColumnName is a null pointer, then an exception condition is raised: *CLI-specific condition — invalid use of null pointer*.

11) If CatalogName is a null pointer, then *NL1* is set to zero. If SchemaName is a null pointer, then *NL2* is set to zero. If TableName is a null pointer, then *NL3* is set to zero. If ColumnName is a null pointer, then *NL4* is set to zero.

12) Case:

a) If *NL1* is not negative, then let *L* be *NL1*.

b) If *NL1* indicates NULL TERMINATED, then let *L* be the number of octets of CatalogName that precede the implementation-defined null character that terminates a C character string.

c) Otherwise, an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

Let *CATVAL* be the first *L* octets of CatalogName.

13) Case:

a) If *NL2* is not negative, then let *L* be *NL2*.

b) If *NL2* indicates NULL TERMINATED, then let *L* be the number of octets of SchemaName that precede the implementation-defined null character that terminates a C character string.

c) Otherwise, an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

Let *SCHVAL* be the first *L* octets of SchemaName.

14) Case:

a) If *NL3* is not negative, then let *L* be *NL3*.

b) If *NL3* indicates NULL TERMINATED, then let *L* be the number of octets of TableName that precede the implementation-defined null character that terminates a C character string.

c) Otherwise, an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

Let *TBLVAL* be the first *L* octets of TableName.

15) Case:

a) If *NL4* is not negative, then let *L* be *NL4*.

b) If *NL4* indicates NULL TERMINATED, then let *L* be the number of octets of ColumnName that precede the implementation-defined null character that terminates a C character string.

c) Otherwise, an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

Let *COLVAL* be the first *L* octets of ColumnName.

16) Case:

a) If the METADATA ID attribute of *S* is TRUE, then:

i) Case:

1) If the value of *NL1* is zero, then let *CATSTR* be a zero-length string.

2) Otherwise,

Case:

A) If `SUBSTRING(TRIM('CATVAL') FROM 1 FOR 1) = '"'` and if `SUBSTRING(TRIM('CATVAL') FROM CHAR_LENGTH(TRIM('CATVAL')) FOR 1) = '"'`, then let *TEMPSTR* be the value obtained from evaluating:

```
SUBSTRING ( TRIM('CATVAL') FROM 2
  FOR CHAR_LENGTH ( TRIM('CATVAL') ) - 2 )
```

and let *CATSTR* be the character string:

```
TABLE_CAT = 'TEMPSTR' AND
```

B) Otherwise, let *CATSTR* be the character string:

```
UPPER(TABLE_CAT) = UPPER('CATVAL') AND
```

ii) Case:

1) If the value of *NL2* is zero, then let *SCHSTR* be a zero-length string.

2) Otherwise,

Case:

A) If `SUBSTRING(TRIM('SCHVAL') FROM 1 FOR 1) = '"'` and if `SUBSTRING(TRIM('SCHVAL') FROM CHAR_LENGTH(TRIM('SCHVAL')) FOR 1) = '"'`, then let *TEMPSTR* be the value obtained from evaluating:

```
SUBSTRING ( TRIM('SCHVAL') FROM 2
  FOR CHAR_LENGTH ( TRIM('SCHVAL') ) - 2 )
```

and let *SCHSTR* be the character string:

```
TABLE_SCHEM = 'TEMPSTR' AND
```

B) Otherwise, let *SCHSTR* be the character string:

```
UPPER(TABLE_SCHEM) = UPPER('SCHVAL') AND
```

iii) Case:

1) If the value of *NL3* is zero, then let *TBLSTR* be a zero-length string.

2) Otherwise,

Case:

A) If `SUBSTRING(TRIM('TBLVAL') FROM 1 FOR 1) = '"'` and if `SUBSTRING(TRIM('TBLVAL') FROM CHAR_LENGTH(TRIM('TBLVAL')) FOR 1) = '"'`, then let *TEMPSTR* be the value obtained from evaluating:

```
SUBSTRING ( TRIM('TBLVAL') FROM 2
  FOR CHAR_LENGTH ( TRIM('TBLVAL') ) - 2 )
```

and let *TBLSTR* be the character string:

```
TABLE_NAME = 'TEMPSTR' AND
```

B) Otherwise, let *TBLSTR* be the character string:

```
UPPER(TABLE_NAME) = UPPER('TBLVAL') AND
```

iv) Case:

1) If the value of *NL4* is zero, then let *COLSTR* be a zero-length string.

2) Otherwise,

Case:

A) If `SUBSTRING(TRIM('COLVAL') FROM 1 FOR 1) = '"'` and if `SUBSTRING(TRIM('COLVAL') FROM CHAR_LENGTH(TRIM('COLVAL')) FOR 1) = '"'`, then let *TEMPSTR* be the value obtained from evaluating:

```
SUBSTRING ( TRIM('COLVAL') FROM 2
  FOR CHAR_LENGTH ( TRIM('COLVAL') ) - 2 )
```

and let *COLSTR* be the character string:

```
COLUMN_NAME = 'TEMPSTR'
```

B) Otherwise, let *COLSTR* be the character string:

```
UPPER(COLUMN_NAME) = UPPER('COLVAL')
```

b) Otherwise:

i) Let *SPC* be the Code value from Table 29, “Codes and data types for implementation information”, that corresponds to the Information Type SEARCH PATTERN ESCAPE in that same table.

ii) Let *ESC* be the value of InfoValue that is returned by the execution of GetInfo() with the value of InfoType set to *SPC*.

iii) If the value of *NL1* is zero, then let *CATSTR* be a zero-length string; otherwise, let *CATSTR* be the character string:

```
TABLE_CAT = 'CATVAL' AND
```

iv) If the value of *NL2* is zero, then let *SCHSTR* be a zero-length string; otherwise, let *SCHSTR* be the character string:

```
TABLE_SCHEM LIKE 'SCHVAL' ESCAPE 'ESC' AND
```

NOTE 18 — The pattern value specified in the string to the right of LIKE may use the escape character that is indicated by the value of the SEARCH PATTERN ESCAPE information type from Table 29, “Codes and data types for implementation information”.

v) If the value of *NL3* is zero, then let *TBLSTR* be a zero-length string; otherwise, let *TBLSTR* be the character string:

```
TABLE_NAME LIKE 'TBLVAL' ESCAPE 'ESC' AND
```

NOTE 19 — The pattern value specified in the string to the right of LIKE may use the escape character that is indicated by the value of the SEARCH PATTERN ESCAPE information type from Table 29, “Codes and data types for implementation information”.

vi) If the value of *NL4* is zero, then let *COLSTR* be a zero-length string. Otherwise, let *COLSTR* be the character string:

```
COLUMN_NAME = 'COLVAL' AND
```

17) Let *PRED* be the result of evaluating:

```
CATSTR || ' ' || SCHSTR || ' ' ||
TBLSTR || ' ' || COLSTR || ' ' || 1=1
```

18) Let *STMT* be the character string:

```
SELECT *
FROM COLUMNS_QUERY
WHERE PRED
ORDER BY TABLE_CAT, TABLE_SCHEM, TABLE_NAME, ORDINAL_POSITION
```

19) ExecDirect is implicitly invoked with *S* as the value of StatementHandle, *STMT* as the value of StatementText, and the length of *STMT* as the value of TextLength.

6.12 Connect

Function

Establish a connection.

Definition

```
Connect (
    ConnectionHandle    IN   INTEGER,
    ServerName          IN   CHARACTER(L1),
    NameLength1         IN   SMALLINT,
    UserName            IN   CHARACTER(L2),
    NameLength2         IN   SMALLINT,
    Authentication      IN   CHARACTER(L3),
    NameLength3         IN   SMALLINT )
    RETURNS SMALLINT
```

where:

— *L1* has a maximum value of 128.

— *L2* has a maximum value equal to the implementation-defined maximum length of a variable-length character string.

— *L3* and has an implementation-defined maximum value.

General Rules

1) Case:

   a) If ConnectionHandle does not identify an allocated SQL-connection, then an exception condition is raised: *CLI-specific condition — invalid handle*.

   b) Otherwise:

      i) Let *C* be the allocated SQL-connection identified by ConnectionHandle.

      ii) The diagnostics area associated with *C* is emptied.

2) If an SQL-transaction is active for the current SQL-connection and the implementation does not support transactions that affect more than one SQL-server, then an exception condition is raised: *feature not supported — multiple server transactions*.

3) If there is an established SQL-connection associated with *C*, then an exception condition is raised: *connection exception — connection name in use*.

4) Case:

   a) If ServerName is a null pointer, then let *NL1* be zero.

   b) Otherwise, let *NL1* be the value of NameLength1.

5) Case:

   a) If *NL1* is not negative, then let *L1* be *NL1*.

   b) If *NL1* indicates NULL TERMINATED, then let *L1* be the number of octets of ServerName that precede the implementation-defined null character that terminates a C character string.

   c) Otherwise, an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

6) Case:

   a) If *L1* is zero, then let 'DEFAULT' be the value of *SN*.

   b) If *L1* is greater than 128, then an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

   c) Otherwise, let *SN* be the first *L1* octets of ServerName.

7) Let *E* be the allocated SQL-environment with which *C* is associated.

8) Case:

   a) If UserName is a null pointer, then let *NL2* be zero.

   b) Otherwise, let *NL2* be the value of NameLength2.

9) Case:

   a) If *NL2* is not negative, then let *L2* be *NL2*.

   b) If *NL2* indicates NULL TERMINATED, then let *L2* be the number of Octets of UserName that precede the implementation-defined null character that terminates a C character string.

   c) Otherwise, an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

10) Case:

   a) If Authentication is a null pointer, then let *NL3* be zero.

   b) Otherwise, let *NL3* be the value of NameLength3.

11) Case:

   a) If *NL3* is not negative, then let *L3* be *NL3*.

   b) If *NL3* indicates NULL TERMINATED, then let *L3* be the number of octets of Authentication that precede the implementation-defined null character that terminates a C character string.

   c) Otherwise, an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

12) Case:

   a) If the value of *SN* is 'DEFAULT', then:

      i) If *L2* is not zero, then an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

ii) If *L3* is not zero, then an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

iii) If an established default SQL-connection is associated with an allocated SQL-connection associated with *E*, then an exception condition is raised: *connection exception — connection name in use*.

b) Otherwise:

i) If *L2* is zero, then let *UN* be an implementation-defined <user identifier>.

ii) If *L2* is non-zero, then:

1) Let *UV* be the first *L2* octets of UserName and let *UN* be the result of

```
TRIM ( BOTH ' ' FROM 'UV' )
```

2) If *UN* does not conform to the Format and Syntax Rules of a <user identifier>, then an exception condition is raised: *invalid authorization specification*.

3) If *UN* does not conform to any implementation-defined restrictions on its value, then an exception condition is raised: *invalid authorization specification*.

iii) Case:

1) If *L3* is not zero, then let *AU* be the first *L3* octets of Authentication.

2) Otherwise, let *AU* be an implementation-defined authentication string, whose length may be zero.

13) Case:

a) If the value of *SN* is 'DEFAULT', then the default SQL-session is initiated and associated with the default SQL-server. The method by which the default SQL-server is determined is implementation-defined.

b) Otherwise, an SQL-session is initiated and associated with the SQL-server identified by *SN*. The method by which *SN* is used to determine the appropriate SQL-server is implementation-defined.

14) If an SQL-session is successfully initiated, then:

a) The current SQL-connection and current SQL-session, if any, become a *dormant SQL-connection* and a *dormant SQL-session* respectively. The SQL-session context information is preserved and is not affected in any way by operations performed over the initiated SQL-connection.

NOTE 20 — The SQL-session context information is defined in Subclause 4.37, “SQL-sessions”, in ISO/IEC 9075-2.

b) The initiated SQL-session becomes the *current SQL-session* and the SQL-connection established to that SQL-session becomes the *current SQL-connection* and is associated with *C*.

NOTE 21 — If an SQL-session is not successfully initiated, then the current SQL-connection and current SQL-session, if any, remain unchanged.

15) If the SQL-client cannot establish the SQL-connection, then an exception condition is raised: *connection exception — SQL-client unable to establish SQL-connection*.

16) If the SQL-server rejects the establishment of the SQL-connection, then an exception condition is raised: *connection exception — SQL-server rejected establishment of SQL-connection*.

NOTE 22 — *AU* and *UN* are used by the SQL-server, along with other implementation-dependent values, to determine whether to accept or reject the establishment of an SQL-session.

17) The SQL-server for the subsequent execution of SQL-statements via CLI routine invocations is set to the SQL-server identified by *SN*.

18) The SQL-session user identifier and the current user identifier are set to *UN*. The current role name is set to the null value.

6.13 CopyDesc

Function

Copy a CLI descriptor.

Definition

```
CopyDesc (
    SourceDescHandle     IN        INTEGER,
    TargetDescHandle     IN        INTEGER )
    RETURNS SMALLINT
```

General Rules

1) Case:

   a) If SourceDescHandle does not identify an allocated CLI descriptor area, then an exception condition is raised: *CLI-specific condition — invalid handle*.

   b) Otherwise, let *SD* be the CLI descriptor area identified by SourceDescHandle.

2) Case:

   a) If TargetDescHandle does not identify an allocated CLI descriptor area, then an exception condition is raised: *CLI-specific condition — invalid handle*.

   b) Otherwise:

      i) Let *TD* be the CLI descriptor area identified by TargetDescHandle.

      ii) The diagnostics area associated with *TD* is emptied.

3) The General Rules of Subclause 5.11, "Deferred parameter check", are applied to *SD* as the DESCRIPTOR AREA.

4) The General Rules of Subclause 5.11, "Deferred parameter check", are applied to *TD* as the DESCRIPTOR AREA.

5) If *TD* is an implementation row descriptor, then an exception condition is raised: *CLI-specific condition — cannot modify an implementation row descriptor*.

6) Let *AT* be the value of the ALLOC_TYPE field of *TD*.

7) The contents of *TD* are replaced by a copy of the contents of *SD*.

8) The ALLOC_TYPE field of *TD* is set to *AT*.

6.14 DataSources

Function

Get server name(s) that the SQL/CLI application can connect to, along with description information, if available.

Definition

```
DataSources (
    EnvironmentHandle    IN        INTEGER,
    Direction            IN        SMALLINT,
    ServerName           OUT       CHARACTER(L1),
    BufferLength1        IN        SMALLINT,
    NameLength1          OUT       SMALLINT,
    Description          OUT       CHARACTER(L2),
    BufferLength2        IN        SMALLINT,
    NameLength2          OUT       SMALLINT )
    RETURNS SMALLINT
```

where *L1* and *L2* have maximum values equal to the implementation-defined maximum length of a variable-length character string.

General Rules

1) Let *EH* be the value of EnvironmentHandle.

2) If *EH* does not identify an allocated SQL-environment or if it identifies an allocated skeleton SQL-environment, then an exception condition is raised: *CLI-specific condition — invalid handle*.

3) Let *E* be the allocated SQL-environment identified by *EH*. The diagnostics area associated with *E* is emptied.

4) Let *BL1* and *BL2* be the values of BufferLength1 and BufferLength2, respectively.

5) Let *D* be the value of Direction.

6) If *D* is not either the code value for NEXT or the code value for FIRST in Table 25, "Codes used for fetch orientation", then an exception condition is raised: *CLI-specific condition — invalid retrieval code*.

7) Let SN_1, SN_2, SN_3, *etc*., be an ordered set of the names of SQL-servers to which the SQL/CLI application might be eligible to connect (where the mechanism used to establish this set is implementation-defined).

   NOTE 23 — SN_1, SN_2, SN_3, *etc*., are the names that an SQL/CLI application would use in invocations of Connect, rather than the "actual" names of the SQL-servers.

8) Let D_1, D_2, D_3, *etc*., be strings describing the SQL-servers named by SN_1, SN_2, SN_3, *etc*. (again provided via an implementation-defined mechanism).

9) Case:

   a) If *D* indicates FIRST, or if DataSources has never been successfully called on *EH*, or if the previous call to DataSources on *EH* raised a completion condition: *no data*, then:

i) If there are no entries in the set SN_1, SN_2, SN_3, *etc*., then a completion condition is raised: *no data* and no further rules for this Subclause are applied.

ii) The General Rules of Subclause 5.9, “Character string retrieval”, are applied with ServerName, SN_1, *BL1*, and NameLength1 as *TARGET*, *VALUE*, *TARGET OCTET LENGTH*, and *RETURNED OCTET LENGTH*, respectively.

iii) The General Rules of Subclause 5.9, “Character string retrieval”, are applied with Description, D_1, *BL2*, and NameLength2 as *TARGET*, *VALUE*, *TARGET OCTET LENGTH*, and *RETURNED OCTET LENGTH*, respectively.

b) Otherwise,

i) Let SN_n be the ServerName value that was returned on the previous call to DataSources on *EH*.

ii) If there is no entry in the set after SN_n, then a completion condition is raised: *no data* and no further rules for this subclause are applied.

iii) The General Rules of Subclause 5.9, “Character string retrieval”, are applied with ServerName, SN_{n+1}, *BL1*, and NameLength1 as *TARGET*, *VALUE*, *TARGET OCTET LENGTH*, and *RETURNED OCTET LENGTH*, respectively.

iv) The General Rules of Subclause 5.9, “Character string retrieval”, are applied with Description, D_{n+1}, *BL2*, and NameLength2 as *TARGET*, *VALUE*, *TARGET OCTET LENGTH*, and *RETURNED OCTET LENGTH*, respectively.

6.15 DescribeCol

Function

Get column attributes.

Definition

```
DescribeCol (
    StatementHandle      IN        INTEGER,
    ColumnNumber         IN        SMALLINT,
    ColumnName           OUT       CHARACTER(L),
    BufferLength         IN        SMALLINT,
    NameLength           OUT       SMALLINT,
    DataType             OUT       SMALLINT,
    ColumnSize           OUT       INTEGER,
    DecimalDigits        OUT       SMALLINT,
    Nullable             OUT       SMALLINT )
    RETURNS SMALLINT
```

where *L* has a maximum value equal to the implementation-defined maximum length of a variable-length character string.

General Rules

1) Let *S* be the allocated SQL-statement identified by StatementHandle.

2) If there is no prepared or executed statement associated with *S*, then an exception condition is raised: *CLI-specific condition — function sequence error*.

3) Let *IRD* be the implementation row descriptor associated with *S* and let *N* be the value of the TOP_LEVEL_COUNT field of *IRD*.

4) If *N* is zero, then an exception condition is raised: *dynamic SQL error — prepared statement not a cursor specification*.

5) Let *CN* be the value of ColumnNumber.

6) If *CN* is less than 1 (one) or greater than *N*, then an exception condition is raised: *dynamic SQL error — invalid descriptor index*.

7) Let *RI* be the number of the descriptor record in *IRD* that is the *CN*-th descriptor area for which LEVEL is 0 (zero). Let *C* be the <select list> column described by the item descriptor area of *IRD* specified by *RI*.

8) Let *BL* be the value of BufferLength.

9) Information is retrieved from *IRD*:

   a) Case:

      i) If the data type of *C* is datetime, then DataType is set to the value of the Code column from Table 36, "Concise codes used with datetime data types in SQL/CLI", corresponding to the datetime interval code of *C*.

ii) If the data type of *C* is interval, then DataType is set to the value of the Code column from Table 37, “Concise codes used with interval data types in SQL/CLI”, corresponding to the datetime interval code of *C*.

iii) Otherwise, DataType is set to the data type of *C*.

b) Case:

i) If the data type of *C* is character string, then ColumnSize is set to the maximum length in octets of *C*.

ii) If the data type of *C* is exact numeric or approximate numeric, then ColumnSize is set to the maximum length of *C* in decimal digits.

iii) If the data type of *C* is datetime or interval, then ColumnSize is set to the length in positions of *C*.

iv) If the data type of *C* is a reference type, then ColumnSize is set to the length in octets of that reference type.

v) Otherwise, ColumnSize is set to an implementation-dependent value.

c) Case:

i) If the data type of *C* is exact numeric, then DecimalDigits is set to the scale of *C*.

ii) If the data type of *C* is datetime, then DecimalDigits is set to the time fractional seconds precision of *C*.

iii) If the data type of *C* is interval, then DecimalDigits is set to the interval fractional seconds precision of *C*.

iv) Otherwise, DecimalDigits is set to an implementation-dependent value.

d) If *C* can have the null value, then Nullable is set to 1 (one); otherwise, Nullable is set to 0 (zero).

e) The name associated with *C* is retrieved. If *C* has an implementation-dependent name, then the value retrieved is the implementation-dependent name for *C*; otherwise, the value retrieved is the <derived column> name of *C*. Let *V* be the value retrieved. The General Rules of Subclause 5.9, “Character string retrieval”, are applied with ColumnName, *V*, *BL*, and NameLength as *TARGET*, *VALUE*, *TARGET OCTET LENGTH*, and *RETURNED OCTET LENGTH*, respectively.

6.16 Disconnect

Function

Terminate an established connection.

Definition

```
Disconnect (
    ConnectionHandle    IN   INTEGER )
    RETURNS SMALLINT
```

General Rules

1) Case:

   a) If ConnectionHandle does not identify an allocated SQL-connection, then an exception condition is raised: *CLI-specific condition — invalid handle*.

   b) Otherwise:

      i) Let *C* be the allocated SQL-connection identified by ConnectionHandle.

      ii) The diagnostics area associated with *C* is emptied.

2) Case:

   a) If there is no established SQL-connection associated with *C*, then an exception condition is raised: *connection exception — connection does not exist*.

   b) Otherwise, let *EC* be the established SQL-connection associated with *C*.

3) Let *L1* be a list of the allocated SQL-statements associated with *C*. Let *L2* be a list of the allocated CLI descriptor areas associated with *C*.

4) If *EC* is active, then

   Case:

   a) If any allocated SQL-statement in *L1* has a deferred parameter number associated with it, then an exception condition is raised: *CLI-specific condition — function sequence error*.

   b) Otherwise, an exception condition is raised: *invalid transaction state — active SQL-transaction*.

5) For every allocated SQL-statement *AS* in *L1*:

   a) Let *SH* be the StatementHandle that identifies *AS*.

   b) FreeHandle is implicitly invoked with HandleType indicating STATEMENT HANDLE and with *SH* as the value of Handle.

      NOTE 24 — Any diagnostic information generated by the invocation is associated with *C* and not with *AS*.

6) For every allocated CLI descriptor area *AD* in *L2*:

a) Let *DH* be the DescriptorHandle that identifies *AD*.

b) FreeHandle is implicitly invoked with HandleType indicating DESCRIPTOR HANDLE and with *DH* as the value of Handle.

NOTE 25 — Any diagnostic information generated by the invocation is associated with *C* and not with *AD*.

7) Let *CC* be the current SQL-connection.

8) The SQL-session associated with *EC* is terminated. *EC* is terminated, regardless of any exception conditions that might occur during the disconnection process, and is no longer associated with *C*.

9) If any error is detected during the disconnection process, then a completion condition is raised: *warning — disconnect error*.

10) If *EC* and *CC* were the same SQL-connection, then there is no current SQL-connection. Otherwise, *CC* remains the current SQL-connection.

6.17 EndTran

Function

Terminate an SQL-transaction.

Definition

```
EndTran (
    HandleType          IN   SMALLINT,
    Handle              IN   INTEGER,
    CompletionType      IN   SMALLINT )
    RETURNS SMALLINT
```

General Rules

1) Let *HT* be the value of HandleType and let *H* be the value of Handle.

2) If *HT* is not one of the code values in Table 14, “Codes used for SQL/CLI handle types”, then an exception condition is raised: *CLI-specific condition — invalid handle*.

3) Case:

   a) If *HT* indicates STATEMENT HANDLE, then

      Case:

      i) If *H* does not identify an allocated SQL-statement, then an exception condition is raised: *CLI-specific condition — invalid handle*.

      ii) Otherwise, an exception condition is raised: *CLI-specific condition — invalid attribute identifier*.

   b) If *HT* indicates DESCRIPTOR HANDLE, then

      Case:

      i) If *H* does not identify an allocated CLI descriptor area, then an exception condition is raised: *CLI-specific condition — invalid handle*.

      ii) Otherwise, an exception condition is raised: *CLI-specific condition — invalid attribute identifier*.

   c) If *HT* indicates CONNECTION HANDLE, then

      Case:

      i) If *H* does not identify an allocated SQL-connection, then an exception condition is raised: *CLI-specific condition — invalid handle*.

      ii) Otherwise:

         1) Let *C* be the allocated SQL-connection identified by *H*.

         2) The diagnostics area associated with *C* is emptied.

3) If *C* has an associated established SQL-connection that is active, then let *L1* be a list containing *C*; otherwise, let *L1* be an empty list.

d) If *HT* indicates ENVIRONMENT HANDLE, then

Case:

i) If *H* does not identify an allocated SQL-environment or if it identifies an allocated SQL-environment that is a skeleton SQL-environment, then an exception condition is raised: *CLI-specific condition — invalid handle*.

ii) Otherwise:

1) Let *E* be the allocated SQL-environment identified by *H*.

2) The diagnostics area associated with *E* is emptied.

3) Let *L* be a list of the allocated SQL-connections associated with *E*. Let *L1* be a list of the allocated SQL-connections in *L* that have an associated established SQL-connection that is active.

4) Let *CT* be the value of CompletionType.

5) If *CT* is not one of the code values in Table 15, "Codes used for transaction termination", then an exception condition is raised: *CLI-specific condition — invalid transaction operation code*.

6) If *L1* is empty, then no further rules of this Subclause are applied.

7) If the current SQL-transaction is part of an encompassing transaction that is controlled by an agent other than the SQL-agent, then an exception condition is raised: *invalid transaction termination*.

8) Let *L2* be a list of the allocated SQL-statements associated with allocated SQL-connections in *L1*.

9) If any of the allocated SQL-statements in *L2* has an associated deferred parameter number, then an exception condition is raised: *CLI-specific condition — function sequence error*.

10) Let *L3* be a list of the open cursors associated with allocated SQL-statements in *L2*.

11) If *CT* indicates COMMIT, COMMIT AND CHAIN, ROLLBACK, or ROLLBACK AND CHAIN, then:

a) Case:

i) If *CT* indicates COMMIT or COMMIT AND CHAIN), then let *LOC* be the list of all non-holdable cursors in *L3*.

ii) Otherwise, let *LOC* be the list of all cursors in *L3*.

b) For *OC* ranging over all cursors in *LOC*:

i) Let *S* be the allocated SQL-statement with which *OC* is associated.

ii) *OC* is placed in the closed state and its copy of the select source is destroyed.

iii) Any fetched row associated with *S* is removed from association with *S*.

12) If *CT* indicates COMMIT or COMMIT AND CHAIN, then:

a) If an atomic execution context is active, then an exception condition is raised: *invalid transaction termination*.

b) For every temporary table associated with the current SQL-transaction that specifies the ON COMMIT DELETE option and that was updated by the current SQL-transaction, the invocation of EndTran with *CT* indicating COMMIT is effectively preceded by the execution of a <delete statement: searched> that specifies DELETE FROM *T*, where *T* is the <table name> of that temporary table.

c) The effects specified in the General Rules of Subclause 17.4, “<set constraints mode statement>”, in ISO/IEC 9075-2, occur as if the statement SET CONSTRAINTS ALL IMMEDIATE were executed.

d) Case:

i) If any constraint is not satisfied, then any changes to SQL-data or schemas that were made by the current SQL-transaction are canceled and an exception condition is raised: *transaction rollback — integrity constraint violation*.

ii) If the execution of any <triggered SQL statement> is unsuccessful, then all changes to SQL-data or schemas that were made by the current SQL-transaction are cancelled and an exception condition is raised: *transaction rollback — triggered action exception*.

iii) If any other error preventing commitment of the SQL-transaction has occurred, then any changes to SQL-data or schemas that were made by the current SQL-transaction are canceled and an exception condition is raised: *transaction rollback* with an implementation-defined subclass value.

iv) Otherwise, any changes to SQL-data or schemas that were made by the current SQL-transaction are made accessible to all concurrent and subsequent SQL-transactions.

e) Every savepoint established in the current SQL-transaction is destroyed.

f) Every valid non-holdable locator value is marked invalid.

g) The current SQL-transaction is terminated. If *CT* indicates COMMIT AND CHAIN, then a new SQL-transaction is initiated with the same access mode and isolation level as the SQL-transaction just terminated. Any branch transactions of the SQL-transaction are initiated with the same access mode and isolation level as the corresponding branch of the SQL-transaction just terminated.

13) If *CT* indicates SAVEPOINT NAME RELEASE, then:

a) If *HT* is not CONNECTION HANDLE, then an exception condition is raised: *CLI-specific condition — invalid handle*.

b) Let *SP* be the value of the SAVEPOINT NAME connection attribute of *C*.

c) If *SP* does not specify a savepoint established within the current SQL-transaction, then an exception condition is raised: *savepoint exception — invalid specification*.

d) The savepoint identified by *SP* and all savepoints established by the current SQL-transaction subsequent to the establishment of *SP* are destroyed.

14) If *CT* indicates ROLLBACK or ROLLBACK AND CHAIN, then:

a) If an atomic execution context is active, then an exception condition is raised: *invalid transaction termination*.

b) All changes to SQL-data or schemas that were made by the current SQL-transaction are canceled.

c) Every savepoint established in the current SQL-transaction is destroyed.

d) Every valid locator value is marked invalid.

e) The current SQL-transaction is terminated. If *CT* indicates ROLLBACK AND CHAIN, then a new SQL-transaction is initiated with the same access mode and isolation level as the SQL-transaction just terminated. Any branch transactions of the SQL-transaction are initiated with the same access mode and isolation level as the corresonding branch of the SQL-transaction just terminated.

15) If *CT* indicates SAVEPOINT NAME ROLLBACK, then:

a) If *HT* is not CONNECTION HANDLE, then an exception condition is raised: *CLI-specific condition — invalid handle*.

b) Let *SP* be the value of the SAVEPOINT NAME connection attribute of *C*.

c) If *SP* does not specify a savepoint established within the current SQL-transaction, then an exception condition is raised: *savepoint exception — invalid specification*.

d) If an atomic execution context is active and *SP* specifies a savepoint established before the beginning of the most recent atomic execution context, then an exception condition is raised: *savepoint exception — invalid specification*.

e) Any changes to SQL-data or schemas that were made by the current SQL-transaction subsequent to the establishment of *SP* are canceled.

f) All savepoints established by the current SQL-transaction subsequent to the establishment of *SP* are destroyed.

g) Every valid locator that was generated in the current SQL-transaction subsequent to the establishment of *SP* is marked invalid.

h) For every open cursor *OC* in *L3* that was opened subsequent to the establishment of *SP*:

i) Let *S* be the allocated SQL-statement with which *OC* is associated.

ii) *OC* is placed in the closed state and its copy of the select source is destroyed.

iii) Any fetched row associated with *OC* is removed from association with *S*.

i) The status of any open cursors in *L3* that were opened by the current SQL-transaction before the establishment of *SP* is implementation-defined.

NOTE 26 — The current SQL-transaction is not terminated, and there is no other effect on the SQL-data or schemas.

6.18 Error

Function

Return diagnostic information.

Definition

```
Error (
    EnvironmentHandle   IN   INTEGER,
    ConnectionHandle    IN   INTEGER,
    StatementHandle     IN   INTEGER,
    Sqlstate            OUT  CHARACTER(5),
    NativeError         OUT  INTEGER,
    MessageText         OUT  CHARACTER(L),
    BufferLength        IN   SMALLINT,
    TextLength          OUT  SMALLINT )
    RETURNS SMALLINT
```

where *L* has a maximum value equal to the implementation-defined maximum length of a variable-length character string.

General Rules

1) Case:

   a) If StatementHandle identifies an allocated SQL-statement, then let *IH* be the value of StatementHandle and let *HT* be the code value for STATEMENT HANDLE from Table 14, “Codes used for SQL/CLI handle types”.

   b) If StatementHandle is zero and ConnectionHandle identifies an allocated SQL-connection, then let *IH* be the value of ConnectionHandle and let *HT* be the code value for CONNECTION HANDLE from Table 14, “Codes used for SQL/CLI handle types”.

   c) If ConnectionHandle is zero and EnvironmentHandle identifies an allocated SQL-environment, then let *IH* be the value of EnvironmentHandle and let *HT* be the code value for ENVIRONMENT HANDLE from Table 14, “Codes used for SQL/CLI handle types”.

   d) Otherwise, an exception condition is raised: *CLI-specific condition — invalid handle*.

2) Let *R* be the most recently executed CLI routine, other than Error, GetDiagField, or GetDiagRec, for which *IH* was passed as a value of an input handle.

   NOTE 27 — The GetDiagField, GetDiagRec and Error routines may cause exception or completion conditions to be raised, but they do not cause status records to be generated.

3) Let *N* be the number of status records generated by the execution of *R*. Let *AP* be the number of status records generated by the execution of *R* already processed by Error. If *N* is zero or *AP* equals *N* then a completion condition is raised: *no data*, Sqlstate is set to '00000', the values of NativeError, MessageText, and TextLength are set to implementation-dependent values, and no further rules of this Subclause are applied.

4) Let *SR* be the first status record generated by the execution of *R* not yet processed by Error. Let *RN* be the number of the status record *SR*. Information is retrieved by implicitly executing GetDiagRec as follows:

```
GetDiagRec (HT, IH, RN, Sqlstate,
  NativeError, MessageText, BufferLength, TextLength)
```

5) Add *SR* to the list of status records generated by the execution of *R* already processed by Error.

6.19 ExecDirect

Function

Execute a statement directly.

Definition

```
ExecDirect (
    StatementHandle  IN     INTEGER,
    StatementText    IN     CHARACTER(L),
    TextLength       IN     INTEGER )
    RETURNS SMALLINT
```

where *L* has a maximum value equal to the implementation-defined maximum length of a variable-length character string.

General Rules

1) Let *S* be the allocated SQL-statement identified by StatementHandle.

2) If an open cursor is associated with *S*, then an exception condition is raised: *invalid cursor state*.

3) Let *TL* be the value of TextLength.

4) Case:

   a) If *TL* is not negative, then let *L* be *TL*.

   b) If *TL* indicates NULL TERMINATED, then let *L* be the number of octets of StatementText that precede the implementation-defined null character that terminates a C character string.

   c) Otherwise, an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

5) Case:

   a) If *L* is zero, then an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

   b) Otherwise, let *P* be the first *L* octets of StatementText.

6) If *P* is a <preparable dynamic delete statement: positioned> or a <preparable dynamic update statement: positioned>, then let *CN* be the cursor name referenced by *P*. Let *C* be the allocated SQL-connection with which *S* is associated. If *CN* is not the name of a cursor associated with another allocated SQL-statement associated with *C*, then an exception condition is raised: *invalid cursor name*.

7) If one or more of the following are true, then an exception condition is raised: *syntax error or access rule violation*.

   a) *P* does not conform to the Format, Syntax Rules or Access Rules for a <preparable statement> or *P* is a <start transaction statement>, a <commit statement>, a <rollback statement>, or a <release savepoint statement>.

NOTE 28 — See Table 32, “SQL-statement codes”, in ISO/IEC 9075-2 for the list of <preparable statement>s. Other parts of ISO/IEC 9075 may have corresponding tables that define additional codes representing statements defined by those parts of ISO/IEC 9075.

b) *P* contains a <simple comment>.

c) *P* contains a <dynamic parameter specification> whose data type is undefined as determined by the rules specified in Subclause 20.6, “<prepare statement>”, in ISO/IEC 9075-2.

8) The data type of any <dynamic parameter specification> contained in *P* is determined by the rules specified in Subclause 20.6, “<prepare statement>”, in ISO/IEC 9075-2.

9) Let *DTGN* be the default transform group name and *TFL* be the list of user-defined type name—transform group name pairs used to identify the group of transform functions for every user-defined type that is referenced in *P*. *DTGN* and *TFL* are not affected by the execution of a <set transform group statement> after *P* is prepared.

10) The following objects associated with *S* are destroyed:

a) Any prepared statement.

b) Any cursor.

c) Any select source.

If a cursor associated with *S* is destroyed, then so are any prepared statements that reference that cursor.

11) *P* is prepared.

12) Case:

a) If *P* is a <dynamic select statement> or a <dynamic single row select statement>, then:

i) *P* becomes the *select source* associated with *S*.

ii) If there is no cursor name associated with *S*, then a unique implementation-dependent name that has the prefix 'SQLCUR' or the prefix 'SQL_CUR' becomes the cursor name associated with *S*.

iii) The General Rules of Subclause 5.5, “Implicit DESCRIBE USING clause”, are applied to *P* and *S*, as *SOURCE* and *ALLOCATED STATEMENT*, respectively.

iv) The General Rules of Subclause 5.4, “Implicit cursor”, are applied to *P* and *S* as *SELECT SOURCE* and *ALLOCATED STATEMENT*, respectively.

b) Otherwise:

i) The General Rules of Subclause 5.5, “Implicit DESCRIBE USING clause”, are applied to *P* and *S*, as *SOURCE* and *ALLOCATED STATEMENT*, respectively.

ii) The General Rules of Subclause 5.6, “Implicit EXECUTE USING and OPEN USING clauses”, are applied to 'EXECUTE', *P*, and *S*, as *TYPE*, *SOURCE*, and *ALLOCATED STATEMENT*, respectively.

iii) Case:

1) If *P* is a <preparable dynamic delete statement: positioned>, then:

A) Let *CR* be the cursor referenced by *P* and let *SCR* be the statement associated with *CR*.

B) All the General Rules in Subclause 20.22, "<preparable dynamic delete statement: positioned>", in ISO/IEC 9075-2 apply to *P*. For the purposes of the application of these Rules, the row in *CR* identified by *SCR*'s CURRENT OF POSITION statement attribute is the *current row* of *CR*.

C) If the execution of *P* deleted the current row of *CR*, then the effect on the fetched row, if any, associated with the allocated SQL-statement under which that current row was established, is implementation-defined.

2) If *P* is a <preparable dynamic update statement: positioned>, then:

A) Let *CR* be the cursor referenced by *P* and let *SCR* be the statement associated with *CR*.

B) All the General Rules in Subclause 20.24, "<preparable dynamic update statement: positioned>", in ISO/IEC 9075-2 apply to *P*. For the purposes of the application of these Rules, the row in *CR* identified by *SCR*'s CURRENT OF POSITION statement attribute is the *current row* of *CR*.

C) If the execution of *P* updated the current row of *CR*, then the effect on the fetched row, if any, associated with the allocated SQL-statement under which that current row was established, is implementation-defined.

3) Otherwise, the results of the execution are the same as if the statement were contained in an <externally-invoked procedure> and executed; these are described in Subclause 13.3, "<externally-invoked procedure>", in ISO/IEC 9075-2.

iv) If *P* is a <call statement>, then the General Rules of Subclause 5.7, "Implicit CALL USING clause", are applied to *P* and *S*, as *SOURCE* and *ALLOCATED STATEMENT*, respectively.

13) Let *R* be the value of the ROW_COUNT field from the diagnostics area associated with *S*.

14) *R* becomes the row count associated with *S*.

15) If *P* executed successfully, then any executed statement associated with *S* is destroyed and *P* becomes the executed statement associated with *S*.

6.20 Execute

Function

Execute a prepared statement.

Definition

```
Execute (
    StatementHandle  IN   INTEGER )
    RETURNS SMALLINT
```

General Rules

1) Let *S* be the allocated SQL-statement identified by StatementHandle.

2) If there is no prepared statement associated with *S*, then an exception condition is raised: *CLI-specific condition — function sequence error*. Otherwise, let *P* be the statement that was prepared.

3) If an open cursor is associated with *S*, then an exception condition is raised: *invalid cursor state*.

4) *P* is executed.

5) Case:

   a) If *P* is a <dynamic select statement> or a <dynamic single row select statement>, then the General Rules of Subclause 5.4, “Implicit cursor”, are applied to *P* and *S* as *SELECT SOURCE* and *ALLOCATED STATEMENT*, respectively.

   b) Otherwise:

      i) The General Rules of Subclause 5.6, “Implicit EXECUTE USING and OPEN USING clauses”, are applied with EXECUTE as *TYPE*, P as *SOURCE*, and S as *ALLOCATED STATEMENT*.

      ii) Case:

         1) If *P* is a <preparable dynamic delete statement: positioned>, then:

            A) Let *CR* be the cursor referenced by *P* and let *SCR* be the statement associated with *CR*.

            B) All the General Rules in Subclause 20.22, “<preparable dynamic delete statement: positioned>”, in ISO/IEC 9075-2 apply to *P*. For the purposes of the application of these Rules, the row in *CR* identified by *SCR*'s CURRENT OF POSITION statement attribute is the *current row* of *CR*.

            C) If the execution of *P* deleted the current row of *CR*, then the effect on the fetched row, if any, associated with the allocated SQL-statement under which that current row was established, is implementation-defined.

         2) If *P* is a <preparable dynamic update statement: positioned>, then:

            A) Let *CR* be the cursor referenced by *P* and let *SCR* be the statement associated with *CR*.

B) All the General Rules in Subclause 20.24, “<preparable dynamic update statement: positioned>”, in ISO/IEC 9075-2 apply to *P*. For the purposes of the application of these Rules, the row in *CR* identified by *SCR*'s CURRENT OF POSITION statement attribute is the *current row* of *CR*.

C) If the execution of *P* updated the current row of *CR*, then the effect on the fetched row, if any, associated with the allocated SQL-statement under which that current row was established, is implementation-defined.

3) Otherwise, the results of the execution are the same as if the statement were contained in an <externally-invoked procedure> and executed; these are described in Subclause 13.3, “<externally-invoked procedure>”, in ISO/IEC 9075-2.

iii) If *P* is a <call statement>, then the General Rules of Subclause 5.7, “Implicit CALL USING clause”, are applied to *P* and *S*, as *SOURCE* and *ALLOCATED STATEMENT*, respectively.

6) Let *R* be the value of the ROW_COUNT field from the diagnostics area associated with *S*.

7) *R* becomes the row count associated with *S*.

8) If *P* executed successfully, then any executed statement associated with *S* is destroyed and *P* becomes the executed statement associated with *S*.

6.21 Fetch

Function

Fetch the next row of a cursor.

Definition

```
Fetch (
    StatementHandle      IN        INTEGER )
    RETURNS SMALLINT
```

General Rules

1) Let *S* be the allocated SQL-statement identified by StatementHandle.

2) If there is no executed statement associated with *S*, then an exception condition is raised: *CLI-specific condition — function sequence error*.

3) If there is no open cursor associated with *S*, then an exception condition is raised: *invalid cursor state*. Otherwise, let *CR* be the open cursor associated with *S* and let *T* be the table associated with the open cursor.

4) Let *ARD* be the current application row descriptor for *S* and let *N* be the value of the TOP_LEVEL_COUNT field of *ARD*.

5) For each item descriptor area in *ARD* whose LEVEL is 0 (zero) in the first *AD* item descriptor areas of *ARD*, and for all of their subordinate descriptor areas, refer to a <target specification> whose corresponding item descriptor area has a non-zero value of DATA_POINTER as a *bound target* and refer to the corresponding <select list> column as a *bound column*.

6) Let *IDA* be the item descriptor area of *ARD* corresponding to the *i*-th bound target and let *TT* be the value of the TYPE field of *IDA*.

7) If *TT* indicates DEFAULT, then:

   a) Let *IRD* be the implementation row descriptor associated with *S*.

   b) Let *CT*, *P*, and *SC* be the values of the TYPE, PRECISION, and SCALE fields, respectively, for the item descriptor area of *IRD* corresponding to the *i*-th bound column.

   c) The data type, precision, and scale of the <target specification> described by *IDA* are effectively set to *CT*, *P*, and *SC*, respectively, for the purposes of this Fetch invocation only.

8) Let *R* be the rowset on which *CR* is positioned and let *AS* be the value of the ARRAY_SIZE field in the header of the ARD for *S*.

9) If *T* is empty, or if *R* contains the last row of *T*, or if *CR* is positioned after the end of the result set, then:

   a) *CR* is positioned after the last row of *T*. An empty rowset becomes the fetched rowset associated with *CR*.

   b) No database values are assigned to bound targets.

c) A completion condition is raised: *no data* and no further rules of this Subclause are applied.

10) Case:

a) If the position of *CR* is before the start of *T*, then

Case:

i) If the number of rows in *T* is less than or equal to *AS*, then *CR* is positioned on the rowset that has all the rows in *T*.

ii) Otherwise, *CR* is positioned on the rowset that has the first *AS* rows of *T*.

b) Otherwise, let T_t be the table that contains all the rows of *T* that immediately follow the last row of *R*, preserving their order in *T*.

Case:

i) If the number of rows in T_t is less than or equal to *AS*, then *CR* is positioned on the rowset that has all the rows in T_t.

ii) Otherwise, *CR* is positioned on the rowset that has the first *AS* rows of T_t.

11) Let *NR* be the rowset on which *CR* is positioned. Let *ASP* and *RPP* be the values of the ARRAY_STATUS_POINTER and ROWS_PROCESSED_POINTER fields, respectively, in the header of the IRD of *S*.

12) If *RPP* is not a null pointer, then set the value of the host variable addressed by *RPP* to zero.

13) Let *RS* be the number of rows in *NR*. For *RN* ranging from 1 (one) to *RS*:

a) Let *RNR* be the *RN*-th row of *NR*. Let *ROWS_PROCESSED* be 0 (zero).

Case:

i) If an exception condition is raised during derivation of any <derived column> associated with *RNR* and *ASP* is not a null pointer, then set the *RN*-th element of *ASP* to 5 (indicating **Row error**). For all status records that result from the application of this rule, the ROW_NUMBER field is set to *RN* and the COLUMN_NUMBER field is set to the appropriate column number, if any.

ii) Otherwise, the row *RNR* is fetched and *ROWS_PROCESSED* is incremented by 1 (one).

14) Case:

a) If *ROWS_PROCESSED* is greater than 0 (zero), then:

i) Let *SS* be the select source associated with *S*.

ii) *NR* becomes the fetched rowset associated with *S*.

iii) Set *ROWS_PROCESSED* to 0 (zero).

iv) The General Rules of Subclause 5.8, “Implicit FETCH USING clause”, are applied with *SS*, *RS*, *ROWS_PROCESSED*, and *S* as *SOURCE*, *ROWS*, *ROWS PROCESSED*, and *ALLOCATED STATEMENT*, respectively.

Case:

1) If *ROWS_PROCESSED* is greater than 0 (zero), *RN* is less than *AS*, and *ASP* is not a null pointer, then the *RN*+1-th through *AS*-th elements of *ASP* are set to 3 (indicating **No row**). If *ROWS_PROCESSED* is less than *RN*, then a completion condition is raised: *warning*. If *RPP* is not a null pointer, then the value of the host variable addressed by *RPP* is set to the value of *ROWS_PROCESSED*.

2) If *ROWS_PROCESSED* is 0 (zero), then the values of all bound targets are implementation-dependent, and *CR* remains positioned on *NR*.

b) Otherwise, the values of all bound targets are implementation-dependent and *CR* remains positioned on *NR*.

6.22 FetchScroll

Function

Position a cursor on the specified row and retrieve values from that row.

Definition

```
FetchScroll (
    StatementHandle          IN      INTEGER,
    FetchOrientation         IN      SMALLINT,
    FetchOffset              IN      INTEGER )
    RETURNS SMALLINT
```

General Rules

1) Let *S* be the allocated SQL-statement identified by StatementHandle.

2) If there is no executed statement associated with *S*, then an exception condition is raised: *CLI-specific condition — function sequence error*.

3) If there is no open cursor associated with *S*, then an exception condition is raised: *invalid cursor state*; otherwise, let *CR* be the open cursor associated with *S* and let *T* be the table associated with the open cursor.

4) If FetchOrientation is not one of the code values in Table 25, “Codes used for fetch orientation”, then an exception condition is raised: *CLI-specific condition — invalid fetch orientation*.

5) Let *FO* be the value of FetchOrientation.

6) If the value of the CURSOR SCROLLABLE attribute of *S* is NONSCROLLABLE, and *FO* does not indicate NEXT, then an exception condition is raised: *CLI-specific condition — invalid fetch orientation*.

7) Let *ARD* be the current application row descriptor for *S* and let *N* be the value of the TOP_LEVEL_COUNT field of *ARD*.

8) For each item descriptor area in *ARD* whose LEVEL is 0 (zero) in the first *AD* item descriptor areas of *ARD*, and for all of their subordinate descriptor areas, refer to a <target specification> whose corresponding item descriptor area has a non-zero value of DATA_POINTER as a *bound target* and refer to the corresponding <select list> column as a *bound column*.

9) Let *IDA* be the item descriptor area of *ARD* corresponding to the *i*-th bound target and let *TT* be the value of the TYPE field of *IDA*.

10) If *TT* indicates DEFAULT, then:

    a) Let *IRD* be the implementation row descriptor associated with *S*.

    b) Let *CT*, *P*, and *SC* be the values of the TYPE, PRECISION, and SCALE fields, respectively, for the item descriptor area of *IRD* corresponding to the *i*-th bound column.

    c) The data type, precision, and scale of the <target specification> described by *IDA* are effectively set to *CT*, *P*, and *SC*, respectively, for the purposes of this fetch only.

11) Case:

 a) If *FO* indicates ABSOLUTE or RELATIVE, then let *J* be the value of FetchOffset.

 b) If *FO* indicates NEXT or FIRST, then let *J* be +1.

 c) If *FO* indicates PRIOR or LAST, then let *J* be –1.

12) Let *R* be the rowset on which *CR* is positioned and let *AS* be the value of the ARRAY_SIZE field in the header of the ARD for *S*.

13) Let T_t be a table of the same degree as *T*.

 Case:

 a) If *FO* indicates ABSOLUTE, FIRST, or LAST, then let T_t contain all rows of *T*, preserving their order in *T*.

 b) If *FO* indicates NEXT or indicates RELATIVE with a positive value of *J*, then

  Case:

  i) If the table *T* identified by cursor *CR* is empty or if *R* contains the last row of *T*, then let T_t be a table of no rows.

  ii) If *CR* is positioned before the start of the result set, then let T_t contain all rows of *T*, preserving their order in *T*.

  iii) Otherwise, let T_t contain all rows of *T* after the first row of *R*, preserving their order in *T*.

 c) If *FO* indicates PRIOR or indicates RELATIVE with a negative value of *J*, then

  Case:

  i) If the table *T* identified by cursor *CR* is empty or if *R* contains the first row of *T*, then let T_t be a table of no rows.

  ii) If *CR* is positioned after the end of the result set, then let T_t contain all rows of *T*, preserving their order in *T*.

  iii) Otherwise, let T_t contain all rows of *T* before the first row of *R*, preserving their order in *T*.

 d) If *FO* indicates RELATIVE with a zero value of *J*, then

  Case:

  i) If *R* is not empty, then let T_t be a table comprising all the rows in *R*, preserving their order in *R*.

  ii) Otherwise, let T_t be an empty table.

14) Let *N* be the number of rows in T_t. If *J* is positive, then let *K* be *J*. If *J* is negative, then let *K* be *N*+*J*+1. If *J* is zero, then let *K* be 1 (one).

15) Case:

 a) If *K* is greater than 0 (zero), then

  Case:

i) If (*K* + *AS* - 1) is greater than *N*, then

Case:

Case:

1) If *J* is less than 0 (zero), then

Case:

A) If (*K* + *AS* - 1) is greater than the number of rows in *T*, then *CR* is positioned on the rowset that has all the rows in *T*.

B) Otherwise, *CR* is positioned on the rowset whose first row is the *K*-th row of *T*; that rowset has *AS* rows.

2) Otherwise, if *K* is less than *N*, then *CR* is positioned on the rowset that has all the rows in T_t.

ii) Otherwise, *CR* is positioned on the rowset whose first row is the *K*-th row of T_t; that rowset has *AS* rows.

b) If *K* is less than 0 (zero), but the absolute value of *K* is less than or equal to *AS*, then

Case:

i) If *AS* is greater than the number of rows in *T*, then *CR* is positioned on the rowset that has all the rows in *T*.

ii) Otherwise, *CR* is positioned on the rowset that has the first *AS* rows in *T*.

c) Otherwise, no SQL-data values are assigned and a completion condition is raised: *no data*.

Case:

i) If *FO* indicates RELATIVE with *J* equal to zero, then the position of *CR* is unchanged.

ii) If *FO* indicates NEXT, indicates ABSOLUTE or RELATIVE with *K* greater than *N*, or indicates LAST, then *CR* is positioned after the last row.

iii) Otherwise, *FO* indicates PRIOR, FIRST, or ABSOLUTE or RELATIVE with *K* not greater than *N* and *CR* is positioned before the first row.

No further rules of this Subclause are applied.

16) Let *NR* be the rowset on which *CR* is positioned. Let *ASP* and *RPP* be the values of the ARRAY_STATUS_POINTER and ROWS_PROCESSED_POINTER fields respectively in the header of the IRD of *S*.

17) If *RPP* is not a null pointer, then set the value of the host variable addressed by *RPP* to 0 (zero).

18) Let *RS* be the number of rows in *NR*. For *RN* ranging from 1 (one) to *RS*:

a) Let *R* be the *RN*-th row of *NR*. Let *ROWS_PROCESSED* be 0 (zero).

Case:

i) If an exception condition is raised during derivation of any <derived column> associated with *R* and *ASP* is not a null pointer, then set the *RN*-th element of *ASP* to 5 (indicating **Row error**).

For all status records that result from the application of this Rule, the ROW_NUMBER field is set to *RN* and the COLUMN_NUMBER field is set to the appropriate column number, if any.

ii) Otherwise the row *R* is fetched and *ROWS_PROCESSED* is incremented by 1 (one).

19) Case:

a) If *ROWS_PROCESSED* is greater than 0 (zero), then:

i) Let *SS* be the select source associated with *S*.

ii) *NR* becomes the fetched rowset associated with *S*.

iii) Set *ROWS_PROCESSED* to 0 (zero).

iv) The general rules of Subclause 5.8, "Implicit FETCH USING clause", are applied with *SS*, *RS*, *ROWS_PROCESSED*, and *S* as *SOURCE*, *ROWS*, *ROWS PROCESSED*, and *ALLOCATED STATEMENT*, respectively.

Case:

1) If *ROWS_PROCESSED* is greater than 0 (zero), *RN* is less than *AS*, and *ASP* is not 0 (zero), then set the *RN*+1-th through *AS*-th elements of *ASP* to 3 (indicating **No row**). If *ROWS_PROCESSED* is less than *RN*, then a completion condition is raised: *warning*. If *RPP* is not a null pointer, then the value of the host variable addressed by *RPP* is set to the value of *ROWS_PROCESSED*.

2) If *ROWS_PROCESSED* is 0 (zero), then the values of all bound targets are implementation-dependent and *CR* remains positioned on *NR*.

b) Otherwise, the values of all bound targets are implementation-dependent and *CR* remains positioned on *R*.

6.23 ForeignKeys

Function

Return a result set that contains information about foreign keys either in or referencing a single specified table stored in the Information Schema of the connected data source. The result set contains information about either:

— The primary key of a single specified table together with the foreign keys in all other tables that reference that primary key.

— The foreign keys of a single specified table together with the primary or unique keys to which they refer.

Definition

```
ForeignKeys (
    StatementHandle         IN      INTEGER,
    PKCatalogName           IN      CHARACTER(L1),
    NameLength1             IN      SMALLINT,
    PKSchemaName            IN      CHARACTER(L2),
    NameLength2             IN      SMALLINT,
    PKTableName             IN      CHARACTER(L3),
    NameLength3             IN      SMALLINT,
    FKCatalogName           IN      CHARACTER(L4),
    NameLength4             IN      SMALLINT,
    FKSchemaName            IN      CHARACTER(L5),
    NameLength5             IN      SMALLINT,
    FKTableName             IN      CHARACTER(L6),
    NameLength6             IN      SMALLINT )
    RETURNS SMALLINT
```

where each of *L1*, *L2*, *L3*, *L4*, *L5*, and *L6* has a maximum value equal to the implementation-defined maximum length of a variable-length character string.

General Rules

1) Let *S* be the allocated SQL-statement identified by StatementHandle.

2) If an open cursor is associated with *S*, then an exception condition is raised: *invalid cursor state*.

3) Let *C* be the allocated SQL-connection with which *S* is associated.

4) Let *EC* be the established SQL-connection associated with *C* and let *SS* be the SQL-server on that connection.

5) Let *FOREIGN_KEYS_QUERY* be a table, with the definition:

```
CREATE TABLE FOREIGN_KEYS_QUERY (
    UK_TABLE_CAT            CHARACTER VARYING(128),
    UK_TABLE_SCHEM          CHARACTER VARYING(128) NOT NULL,
    UK_TABLE_NAME           CHARACTER VARYING(128) NOT NULL,
    UK_COLUMN_NAME          CHARACTER VARYING(128) NOT NULL,
    FK_TABLE_CAT            CHARACTER VARYING(128),
    FK_TABLE_SCHEM          CHARACTER VARYING(128) NOT NULL,
    FK_TABLE_NAME           CHARACTER VARYING(128) NOT NULL,
```

```
FK_COLUMN_NAME          CHARACTER VARYING(128) NOT NULL,
ORDINAL_POSITION        SMALLINT NOT NULL,
UPDATE_RULE             SMALLINT,
DELETE_RULE             SMALLINT,
FK_NAME                 CHARACTER VARYING(128),
UK_NAME                 CHARACTER VARYING(128),
DEFERABILITY            SMALLINT,
UNIQUE_OR_PRIMARY       CHARACTER(7) )
```

6) Let *PKN* and *FKN* be the value of PKTableName and FKTableName, respectively.

7) Case:

a) If CHAR_LENGTH(*PKN*) = 0 (zero) and CHAR_LENGTH(*FKN*) ≠ 0 (zero), then the result set returned describes all the foreign keys (if any) of the specified table, and describes the primary or unique keys to which they refer.

i) Let *FKS* represent the set of rows formed by a natural inner join on the values in the CONSTRAINT_CATALOG, CONSTRAINT_SCHEMA, and CONSTRAINT_NAME columns between the rows in *SS*'s Information Schema REFERENTIAL_CONSTRAINTS view and the matching rows in *SS*'s Information Schema TABLE_CONSTRAINTS view.

ii) Let *UK* represent the row in *SS*'s Information Schema TABLE_CONSTRAINTS view that defines the primary or unique key referenced by an individual foreign key in *FKS*. This row is obtained by matching the values in the UNIQUE_CONSTRAINT_CATALOG, UNIQUE_CONSTRAINT_SCHEMA, and UNIQUE_CONSTRAINT_NAME columns in a row of *FKS* to the values in the CONSTRAINT_CATALOG, CONSTRAINT_SCHEMA, and CONSTRAINT_NAME columns in TABLE_CONSTRAINTS.

iii) Let *FK_COLS* represent the set of rows in *SS*'s Information Schema KEY_COLUMN_USAGE view that define the columns within an individual foreign key row in *FKS*.

iv) Let *FKS_COLS* represent the set of rows in the combination of all *FK_COLS* sets.

v) Let *UK_COLS* represent the set of rows in *SS*'s Information Schema KEY_COLUMN_USAGE view that define the columns within an individual *UK*.

vi) Let *UKS_COLS* represent the set of rows in the combination of all *UK_COLS* sets.

vii) Let *XKS_COLS* represent the set of extended rows formed by the inner equijoin of *FKS_COLS* and *UKS_COLS* matching CONSTRAINT_CATALOG, CONSTRAINT_SCHEMA, CONSTRAINT_NAME, and POSITION_IN_UNIQUE_CONSTRAINT in FKS_COLS with CONSTRAINT_CATALOG, CONSTRAINT_SCHEMA, CONSTRAINT_NAME, and ORDINAL_POSITION in *UKS_COLS*, respectively.

Let *FKS_COLS_NAME* be the name of each column of *FKS_COLS* considered in turn; the names of the columns of *XKS_COLS* originating from *FKS_COLS* are respectively `'F_' || FKS_COLS_NAME`.

Let *UKS_COLS_NAME* be the name of each column of *UKS_COLS* considered in turn; the names of the columns of *XKS_COLS* originating from *UKS_COLS* are respectively `'U_' || UKS_COLS_NAME`.

viii) *FOREIGN_KEYS_QUERY* contains a row for each row in *XKS_COLS* where:

1) Let *SUP* be the value of Supported that is returned by the execution of GetFeatureInfo with FeatureType = 'FEATURE' and FeatureId = 'C041' (corresponding to the feature “Information Schema metadata constrained by privileges”).

2) Case:

   A) If the value of *SUP* is 1 (one), then *FOREIGN_KEYS_QUERY* contains a row for each column of all the foreign keys within a specific table in *SS*'s Information Schema TABLE_CONSTRAINTS view.

   B) Otherwise, *FOREIGN_KEYS_QUERY* contains a row for each column of all the foreign keys within a specific table in *SS*'s Information Schema TABLE_CONSTRAINTS view in accordance with implementation-defined authorization criteria.

ix) For each row of *FOREIGN_KEYS_QUERY*:

1) If the implementation does not support catalog names, then UK_TABLE_CAT is set to the null value; otherwise, the value of UK_TABLE_CAT in *FOREIGN_KEYS_QUERY* is the value of the U_TABLE_CATALOG column in *XKS_COLS*.

2) The value of UK_TABLE_SCHEM in *FOREIGN_KEYS_QUERY* is the value of the U_TABLE_SCHEMA column in *XKS_COLS*.

3) The value of UK_TABLE_NAME in *FOREIGN_KEYS_QUERY* is the value of the U_TABLE_NAME column in *XKS_COLS*.

4) The value of UK_COLUMN_NAME in *FOREIGN_KEYS_QUERY* is the value of the U_COLUMN_NAME column in *XKS_COLS*.

5) If the implementation does not support catalog names, then UK_TABLE_CAT is set to the null value; otherwise, the value of FK_TABLE_CAT in *FOREIGN_KEYS_QUERY* is the value of the F_TABLE_CATALOG column in *XKS_COLS*.

6) The value of FK_TABLE_SCHEM in *FOREIGN_KEYS_QUERY* is the value of the F_TABLE_SCHEMA column in *XKS_COLS*.

7) The value of FK_TABLE_NAME in *FOREIGN_KEYS_QUERY* is the value of the F_TABLE_NAME column in *XKS_COLS*.

8) The value of FK_COLUMN_NAME in *FOREIGN_KEYS_QUERY* is the value of the F_COLUMN_NAME column in *XKS_COLS*.

9) The value of ORDINAL_POSITION in *FOREIGN_KEYS_QUERY* is the value of the F_ORDINAL_POSITION column in *XKS_COLS*.

10) The value of UPDATE_RULE in *FOREIGN_KEYS_QUERY* is determined by the value of the UPDATE_RULE column in *XKS_COLS* as follows:

   A) Let *UR* be the value in the UPDATE_RULE column.

   B) If *UR* is 'CASCADE', then the value of UPDATE_RULE is the code for CASCADE in Table 27, “Miscellaneous codes used in CLI”.

   C) If *UR* is 'RESTRICT', then the value of UPDATE_RULE is the code for RESTRICT in Table 27, “Miscellaneous codes used in CLI”.

D) If *UR* is 'SET NULL', then the value of UPDATE_RULE is the code for SET NULL in Table 27, “Miscellaneous codes used in CLI”.

E) If *UR* is 'NO ACTION', then the value of UPDATE_RULE is the code for NO ACTION in Table 27, “Miscellaneous codes used in CLI”.

F) If *UR* is 'SET DEFAULT', then the value of UPDATE_RULE is the code for SET DEFAULT in Table 27, “Miscellaneous codes used in CLI”.

11) The value of DELETE_RULE in *FOREIGN_KEYS_QUERY* is determined by the value of the DELETE_RULE column in *XKS_COLS* as follows:

A) Let *DR* be the value in the DELETE_RULE column.

B) If *DR* is 'CASCADE', then the value of DELETE_RULE is the code for CASCADE in Table 27, “Miscellaneous codes used in CLI”.

C) If *DR* is 'RESTRICT', then the value of DELETE_RULE is the code for RESTRICT in Table 27, “Miscellaneous codes used in CLI”.

D) If *DR* is 'SET NULL', then the value of DELETE_RULE is the code for SET NULL in Table 27, “Miscellaneous codes used in CLI”.

E) If *DR* is 'NO ACTION', then the value of DELETE_RULE is the code for NO ACTION in Table 27, “Miscellaneous codes used in CLI”.

F) If *DR* is 'SET DEFAULT', then the value of DELETE_RULE is the code for SET DEFAULT in Table 27, “Miscellaneous codes used in CLI”.

12) The value of FK_NAME in *FOREIGN_KEYS_QUERY* is the value of the CONSTRAINT_NAME column in *XKS_COLS*.

13) The value of UK_NAME in *FOREIGN_KEYS_QUERY* is the value of the UNIQUE_CONSTRAINT_NAME column in *XKS_COLS*.

14) If there are no implementation-defined mechanisms for setting the value of DEFERABILITY in *FOREIGN_KEYS_QUERY* to the value of the code for INITIALLY DEFERRED or to the value of the code for INITIALLY IMMEDIATE in Table 27, “Miscellaneous codes used in CLI”, then the value of DEFERABILITY in *FOREIGN_KEYS_QUERY* is the code for NOT DEFERRABLE in Table 27, “Miscellaneous codes used in CLI”; otherwise, the value of DEFERABILITY in *FOREIGN_KEYS_QUERY* can be the code for INITIALLY DEFERRED, the value of the code for INITIALLY IMMEDIATE, or the code for NOT DEFERRABLE in Table 27, “Miscellaneous codes used in CLI”.

15) The value of UNIQUE_OR_PRIMARY in *FOREIGN_KEYS_QUERY* is 'UNIQUE' if the foreign key references a UNIQUE key and 'PRIMARY' if the foreign key references a primary key.

x) Let *NL1*, *NL2*, and *NL3* be the values of NameLength4, NameLength5, and NameLength6, respectively.

xi) Let *CATVAL*, *SCHVAL*, and *TBLVAL* be the values of FKCatalogName, FKSchemaName, and FKTableName, respectively.

xii) If the METADATA ID attribute of *S* is TRUE, then:

1) If FKCatalogName is a null pointer and the value of the CATALOG NAME information type from Table 29, “Codes and data types for implementation information”, *Y*, then an exception condition is raised: *CLI-specific condition — invalid use of null pointer*.

2) If FKSchemaName is a null pointer or if FKTableName is a null pointer, then an exception condition is raised: *CLI-specific condition — invalid use of null pointer*.

xiii) If FKCatalogName is a null pointer, then *NL1* is set to zero. If FKSchemaName is a null pointer, then *NL2* is set to zero. If FKTableName is a null pointer, then *NL3* is set to zero.

xiv) Case:

1) If *NL1* is not negative, then let *L* be *NL1*.

2) If *NL1* indicates NULL TERMINATED, then let *L* be the number of octets of FKCatalogName that precede the implementation-defined null character that terminates a C character string.

3) Otherwise, an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

Let *CATVAL* be the first *L* octets of FKCatalogName.

xv) Case:

1) If *NL2* is not negative, then let *L* be *NL2*.

2) If *NL2* indicates NULL TERMINATED, then let *L* be the number of octets of FKSchemaName that precede the implementation-defined null character that terminates a C character string.

3) Otherwise, an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

Let *SCHVAL* be the first *L* octets of FKSchemaName.

xvi) Case:

1) If *NL3* is not negative, then let *L* be *NL3*.

2) If *NL3* indicates NULL TERMINATED, then let *L* be the number of octets of FKTableName that precede the implementation-defined null character that terminates a C character string.

3) Otherwise, an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

Let *TBLVAL* be the first *L* octets of FKTableName.

xvii) Case:

1) If the METADATA ID attribute of *S* is TRUE, then:

A) Case:

I) If the value of *NL1* is zero, then let *CATSTR* be a zero-length string.

II) Otherwise,

Case:

1) If `SUBSTRING(TRIM('`*`CATVAL`*`') FROM 1 FOR 1) = '"'` and if `SUBSTRING(TRIM('`*`CATVAL`*`') FROM CHAR_LENGTH(TRIM('`*`CATVAL`*`')) FOR 1) = '"'`, then let *TEMPSTR* be the value obtained from evaluating:

```
SUBSTRING(TRIM('CATVAL') FROM 2
          FOR CHAR_LENGTH(TRIM('CATVAL')) - 2)
```

and let *CATSTR* be the character string:

```
FK_TABLE_CAT = 'TEMPSTR' AND
```

2) Otherwise, let *CATSTR* be the character string:

```
UPPER(FK_TABLE_CAT) = UPPER('CATVAL') AND
```

B) Case:

I) If the value of *NL2* is zero, then let *SCHSTR* be a zero-length string.

II) Otherwise,

Case:

1) If `SUBSTRING(TRIM('`*`SCHVAL`*`') FROM 1 FOR 1) = '"'` and if `SUBSTRING(TRIM('`*`SCHVAL`*`') FROM CHAR_LENGTH(TRIM('`*`SCHVAL`*`')) FOR 1) = '"'`, then let *TEMPSTR* be the value obtained from evaluating:

```
SUBSTRING(TRIM('SCHVAL') FROM 2
          FOR CHAR_LENGTH(TRIM('SCHVAL')) - 2)
```

and let *SCHSTR* be the character string:

```
FK_TABLE_SCHEM = 'TEMPSTR' AND
```

2) Otherwise, let *SCHSTR* be the character string:

```
UPPER(FK_TABLE_SCHEM) = UPPER('SCHVAL') AND
```

C) Case:

I) If the value of *NL3* is zero, then let *TBLSTR* be a zero-length string.

II) Otherwise,

Case:

1) If `SUBSTRING(TRIM('`*`TBLVAL`*`') FROM 1 FOR 1) = '"'` and if `SUBSTRING(TRIM('`*`TBLVAL`*`') FROM CHAR_LENGTH(TRIM('`*`TBLVAL`*`')) FOR 1) = '"'`, then let *TEMPSTR* be the value obtained from evaluating:

```
SUBSTRING(TRIM('TBLVAL') FROM 2
          FOR CHAR_LENGTH(TRIM('TBLVAL')) - 2)
```

and let *TBLSTR* be the character string:

```
FK_TABLE_NAME = 'TEMPSTR' AND
```

2) Otherwise, let *TBLSTR* be the character string:

```
UPPER(FK_TABLE_NAME) = UPPER('TBLVAL') AND
```

2) Otherwise:

A) If the value of *NL1* is zero, then let *CATSTR* be a zero-length string; otherwise, let *CATSTR* be the character string:

```
FK_TABLE_CAT = 'CATVAL' AND
```

B) If the value of *NL2* is zero, then let *SCHSTR* be a zero-length string; otherwise, let *SCHSTR* be the character string:

```
FK_TABLE_SCHEM = 'SCHVAL' AND
```

C) If the value of *NL3* is zero, then let *TBLSTR* be a zero-length string; otherwise, let *TBLSTR* be the character string:

```
FK_TABLE_NAME = 'TBLVAL' AND
```

xviii) Let *PRED* be the result of evaluating:

```
CATSTR || ' ' || SCHSTR || ' ' || TBLSTR || ' ' || 1=1
```

xix) Let *STMT* be the character string:

```
SELECT *
FROM FOREIGN_KEYS_QUERY
WHERE PRED
ORDER BY FK_TABLE_CAT, FK_TABLE_SCHEM, FK_TABLE_NAME, ORDINAL_POSITION
```

xx) ExecDirect is implicitly invoked with *S* as the value of StatementHandle, *STMT* as the value of StatementText, and the length of *STMT* as the value of TextLength.

b) If CHAR_LENGTH(*PKN*) ≠ 0 (zero) and CHAR_LENGTH(*FKN*) = 0 (zero), then the result set returned contains a description of the primary key (if any) of the specified table together with the descriptions of foreign keys in all other tables that reference that primary key.

i) Let *PKS* represent the set of rows in *SS*'s Information Schema TABLE_CONSTRAINTS view where the value of CONSTRAINT_TYPE is 'PRIMARY KEY'.

ii) Let *X* represent the set of rows formed by a natural inner join on the values in the CONSTRAINT_CATALOG, CONSTRAINT_SCHEMA, and CONSTRAINT_NAME columns between the rows in *SS*'s Information Schema REFERENTIAL_CONSTRAINTS view and the matching rows in *SS*'s Information Schema TABLE_CONSTRAINTS view.

iii) Let *FKS* represent the rows defining the foreign keys that reference an individual primary key in *PKS*. These rows are obtained by matching the values of CONSTRAINT_CATALOG, CONSTRAINT_SCHEMA, and CONSTRAINT_NAME columns in a row of *PKS* to the values in the UNIQUE_CONSTRAINT_CATALOG, UNIQUE_CONSTRAINT_SCHEMA, and UNIQUE_CONSTRAINT_NAME columns in *X*.

iv) Let *FKSS* represent the set of rows in the combination of all *FKS* sets.

v) Let *PK_COLS* represent the set of rows in *SS*'s Information Schema KEY_COLUMN_USAGE view that define the columns within an individual primary key row in *PKS*.

vi) Let *PKS_COLS* represent the set of rows in the combination of all *PK_COLS* sets.

vii) Let *FK_COLS* represent the set of rows in *SS*'s Information Schema KEY_COLUMN_USAGE view that define the columns within an individual foreign key in *FKSS*.

viii) Let *FKS_COLS* represent the set of rows in the combination of all *FK_COLS* sets.

ix) Let *XKS_COLS* represent the set of extended rows formed by the inner equijoin of *PKS_COLS* and *UKS_COLS* matching CONSTRAINT_CATALOG, CONSTRAINT_SCHEMA, CONSTRAINT_NAME, and ORDINAL_POSITION of *PKS_COLS* with CONSTRAINT_CATALOG, CONSTRAINT_SCHEMA, CONSTRAINT_NAME, and POSITION_IN_UNIQUE_CONSTRAINT of *FKS_COLS*, respectively.

Let *PKS_COLS_NAME* be the name of each column of *PKS_COLS* considered in turn; the names of the columns of *XKS_COLS* originating from *PKS_COLS* are respectively `'P_' || UKS_COLS_NAME`.

Let *FKS_COLS_NAME* be the name of each column of *FKS_COLS* considered in turn; the names of the columns of *XKS_COLS* originating from *FKS_COLS* are respectively `'F_' || FKS_COLS_NAME`.

x) *FOREIGN_KEYS_QUERY* contains a row for each row in *XKS_COLS* where:

1) Let *SUP* be the value of Supported that is returned by the execution of GetFeatureInfo with FeatureType = 'FEATURE' and FeatureId = 'C041' (corresponding to the feature “Information Schema metadata constrained by privileges”).

2) Case:

A) If the value of *SUP* is 1 (one), then *FOREIGN_KEYS_QUERY* contains one or more rows describing the foreign keys that reference the primary key of a specific table in *SS*'s Information Schema TABLE_CONSTRAINTS view.

B) Otherwise, *FOREIGN_KEYS_QUERY* contains a row for each column of all the foreign keys that reference the primary key of a specific table in *SS*'s Information Schema TABLE_CONSTRAINTS view in accordance with implementation-defined authorization criteria.

xi) For each row of *FOREIGN_KEYS_QUERY*:

1) If the implementation does not support catalog names, then UK_TABLE_CAT is set to the null value; otherwise, the value of UK_TABLE_CAT in *FOREIGN_KEYS_QUERY* is the value of the P_TABLE_CATALOG column in *XKS_COLS*.

2) The value of UK_TABLE_SCHEM in *FOREIGN_KEYS_QUERY* is the value of the P_TABLE_SCHEMA column in *XKS_COLS*.

3) The value of UK_TABLE_NAME in *FOREIGN_KEYS_QUERY* is the value of the P_TABLE_NAME column in *XKS_COLS*.

4) The value of UK_COLUMN_NAME in *FOREIGN_KEYS_QUERY* is the value of the P_COLUMN_NAME column in *XKS_COLS*.

5) If the implementation does not support catalog names, then UK_TABLE_CAT is set to the null value; otherwise, the value of UK_TABLE_CAT in *FOREIGN_KEYS_QUERY* is the value of the F_TABLE_CATALOG column in *XKS_COLS*.

6) The value of FK_TABLE_SCHEM in *FOREIGN_KEYS_QUERY* is the value of the F_TABLE_SCHEMA column in *XKS_COLS*.

7) The value of FK_TABLE_NAME in *FOREIGN_KEYS_QUERY* is the value of the F_TABLE_NAME column in *XKS_COLS*.

8) The value of FK_COLUMN_NAME in *FOREIGN_KEYS_QUERY* is the value of the F_COLUMN_NAME column in *XKS_COLS*.

9) The value of ORDINAL_POSITION in *FOREIGN_KEYS_QUERY* is the value of the F_ORDINAL_POSITION column in *XKS_COLS*.

10) The value of UPDATE_RULE in *FOREIGN_KEYS_QUERY* is determined by the value of the UPDATE_RULE column in *XKS_COLS* as follows.

    A) Let *UR* be the value in the UPDATE_RULE column.

    B) If *UR* is 'CASCADE', then the value of UPDATE_RULE is the code for CASCADE in Table 27, “Miscellaneous codes used in CLI”.

    C) If *UR* is 'RESTRICT', then the value of UPDATE_RULE is the code for RESTRICT in Table 27, “Miscellaneous codes used in CLI”.

    D) If *UR* is 'SET NULL', then the value of UPDATE_RULE is the code for SET NULL in Table 27, “Miscellaneous codes used in CLI”.

    E) If *UR* is 'NO ACTION', then the value of UPDATE_RULE is the code for NO ACTION in Table 27, “Miscellaneous codes used in CLI”.

    F) If *UR* is 'SET DEFAULT', then the value of UPDATE_RULE is the code for SET DEFAULT in Table 27, “Miscellaneous codes used in CLI”.

11) The value of DELETE_RULE in *FOREIGN_KEYS_QUERY* is determined by the value of the DELETE_RULE column in *XKS_COLS*.

    A) Let *DR* be the value in the DELETE_RULE column.

    B) If *DR* is 'CASCADE', then the value of DELETE_RULE is the code for CASCADE in Table 27, “Miscellaneous codes used in CLI”.

    C) If *DR* is 'RESTRICT', then the value of DELETE_RULE is the code for RESTRICT in Table 27, “Miscellaneous codes used in CLI”.

    D) If *DR* is 'SET NULL', then the value of DELETE_RULE is the code for SET NULL in Table 27, “Miscellaneous codes used in CLI”.

    E) If *DR* is 'NO ACTION', then the value of DELETE_RULE is the code for NO ACTION in Table 27, “Miscellaneous codes used in CLI”.

    F) If *DR* is 'SET DEFAULT', then the value of DELETE_RULE is the code for SET DEFAULT in Table 27, “Miscellaneous codes used in CLI”.

12) The value of FK_NAME in *FOREIGN_KEYS_QUERY* is the value of the CONSTRAINT_NAME column in *XKS_COLS*.

13) The value of UK_NAME in *FOREIGN_KEYS_QUERY* is the value of the UNIQUE_CONSTRAINT_NAME column in *XKS_COLS*.

14) If there are no implementation-defined mechanisms for setting the value of DEFERABILITY in *FOREIGN_KEYS_QUERY* to the value of the code for INITIALLY DEFERRED or to the value of the code for INITIALLY IMMEDIATE in Table 27, “Miscellaneous codes used in CLI”, then the value of DEFERABILITY in *FOREIGN_KEYS_QUERY* is the code for NOT DEFERRABLE in Table 27, “Miscellaneous codes used in CLI”; otherwise, the value of DEFERABILITY in *FOREIGN_KEYS_QUERY* can be the code for INITIALLY DEFERRED, the value of the code for INITIALLY IMMEDIATE, or the code for NOT DEFERRABLE in Table 27, “Miscellaneous codes used in CLI”.

15) The value of UNIQUE_OR_PRIMARY in *FOREIGN_KEYS_QUERY* is 'PRIMARY'.

xii) Let *NL1*, *NL2*, and *NL3* be the values of NameLength1, NameLength2, and NameLength3, respectively.

xiii) Let *CATVAL*, *SCHVAL*, and *TBLVAL* be the values of PKCatalogName, PKSchemaName, and PKTableName, respectively.

xiv) If the METADATA ID attribute of *S* is TRUE, then:

1) If PKCatalogName is a null pointer and the value of the CATALOG NAME information type from Table 29, “Codes and data types for implementation information”, *Y*, then an exception condition is raised: *CLI-specific condition — invalid use of null pointer*.

2) If PKSchemaName is a null pointer or if PKTableName is a null pointer, then an exception condition is raised: *CLI-specific condition — invalid use of null pointer*.

xv) If PKCatalogName is a null pointer, then *NL1* is set to zero. If PKSchemaName is a null pointer, then *NL2* is set to zero. If PKTableName is a null pointer, then *NL3* is set to zero.

xvi) Case:

1) If *NL1* is not negative, then let *L* be *NL1*.

2) If *NL1* indicates NULL TERMINATED, then let *L* be the number of octets of PKCatalogName that precede the implementation-defined null character that terminates a C character string.

3) Otherwise, an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

Let *CATVAL* be the first *L* octets of PKCatalogName.

xvii) Case:

1) If *NL2* is not negative, then let *L* be *NL2*.

2) If *NL2* indicates NULL TERMINATED, then let *L* be the number of octets of PKSchemaName that precede the implementation-defined null character that terminates a C character string.

3) Otherwise, an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

Let *SCHVAL* be the first *L* octets of PKSchemaName.

xviii) Case:

1) If *NL3* is not negative, then let *L* be *NL3*.

2) If *NL3* indicates NULL TERMINATED, then let *L* be the number of octets of PKTableName that precede the implementation-defined null character that terminates a C character string.

3) Otherwise, an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

Let *TBLVAL* be the first *L* octets of PKTableName.

xix) Case:

1) If the METADATA ID attribute of *S* is TRUE, then:

A) Case:

I) If the value of *NL1* is zero, then let *CATSTR* be a zero-length string.

II) Otherwise,

Case:

1) If `SUBSTRING(TRIM('CATVAL') FROM 1 FOR 1) = '"'` and if `SUBSTRING(TRIM('CATVAL') FROM CHAR_LENGTH(TRIM('CATVAL')) FOR 1) = '"'`, then let *TEMPSTR* be the value obtained from evaluating:

```
SUBSTRING ( TRIM('CATVAL') FROM 2
            FOR CHAR_LENGTH ( TRIM('CATVAL') ) - 2 )
```

and let *CATSTR* be the character string:

```
FK_TABLE_CAT = 'TEMPSTR' AND
```

2) Otherwise, let *CATSTR* be the character string:

```
UPPER(FK_TABLE_CAT) = UPPER('CATVAL') AND
```

B) Case:

I) If the value of *NL2* is zero, then let *SCHSTR* be a zero-length string.

II) Otherwise,

Case:

1) If `SUBSTRING(TRIM('SCHVAL') FROM 1 FOR 1) = '"'` and if `SUBSTRING(TRIM('SCHVAL') FROM CHAR_LENGTH(TRIM('SCHVAL')) FOR 1) = '"'`, then let *TEMPSTR* be the value obtained from evaluating:

```
SUBSTRING ( TRIM('SCHVAL') FROM 2
            FOR CHAR_LENGTH ( TRIM('SCHVAL') ) - 2 )
```

and let *SCHSTR* be the character string:

```
FK_TABLE_SCHEM = 'TEMPSTR' AND
```

2) Otherwise, let *SCHSTR* be the character string:

```
UPPER(FK_TABLE_SCHEM) = UPPER('SCHVAL') AND
```

C) Case:

I) If the value of *NL3* is zero, then let *TBLSTR* be a zero-length string.

II) Otherwise,

Case:

1) If `SUBSTRING(TRIM('TBLVAL') FROM 1 FOR 1) = '"'` and if `SUBSTRING(TRIM('TBLVAL') FROM CHAR_LENGTH(TRIM('TBLVAL')) FOR 1) = '"'`, then let *TEMPSTR* be the value obtained from evaluating:

```
SUBSTRING ( TRIM('TBLVAL') FROM 2
            FOR CHAR_LENGTH ( TRIM('TBLVAL') ) - 2 )
```

and let *TBLSTR* be the character string:

```
FK_TABLE_NAME = 'TEMPSTR' AND
```

2) Otherwise, let *TBLSTR* be the character string:

```
UPPER(FK_TABLE_NAME) = UPPER('TBLVAL') AND
```

2) Otherwise:

A) If the value of *NL1* is zero, then let *CATSTR* be a zero-length string; otherwise, let *CATSTR* be the character string:

```
FK_TABLE_CAT = 'CATVAL' AND
```

B) If the value of *NL2* is zero, then let *SCHSTR* be a zero-length string; otherwise, let *SCHSTR* be the character string:

```
FK_TABLE_SCHEM = 'SCHVAL' AND
```

C) If the value of *NL3* is zero, then let *TBLSTR* be a zero-length string; otherwise, let *TBLSTR* be the character string:

```
FK_TABLE_NAME = 'TBLVAL' AND
```

xx) Let *PRED* be the result of evaluating:

```
CATSTR || ' ' || SCHSTR || ' ' || TBLSTR || ' ' || 1=1
```

xxi) Let *STMT* be the character string:

```
SELECT *
FROM FOREIGN_KEYS_QUERY
WHERE PRED
ORDER BY FK_TABLE_CAT, FK_TABLE_SCHEM, FK_TABLE_NAME, ORDINAL_POSITION
```

xxii) ExecDirect is implicitly invoked with *S* as the value of StatementHandle, *STMT* as the value of StatementText, and the length of *STMT* as the value of TextLength.

c) If CHAR_LENGTH(*PKN*) ≠ 0 (zero) and CHAR_LENGTH(*FKN*) ≠ 0 (zero), then the result of the routine is implementation-defined.

6.24 FreeConnect

Function

Deallocate an SQL-connection.

Definition

```
FreeConnect (
    ConnectionHandle           IN   INTEGER )
    RETURNS SMALLINT
```

General Rules

1) Let *CH* be the value of ConnectionHandle.
2) FreeHandle is implicitly invoked with HandleType indicating CONNECTION HANDLE and with *CH* as the value of Handle.

6.25 FreeEnv

Function

Deallocate an SQL-environment.

Definition

```
FreeEnv (
    EnvironmentHandle           IN   INTEGER )
    RETURNS SMALLINT
```

General Rules

1) Let *EH* be the value of EnvironmentHandle.
2) FreeHandle is implicitly invoked with HandleType indicating ENVIRONMENT HANDLE and with *EH* as the value of Handle.

6.26 FreeHandle

Function

Free a resource.

Definition

```
FreeHandle (
    HandleType       IN   SMALLINT,
    Handle           IN   INTEGER )
    RETURNS SMALLINT
```

General Rules

1) Let *HT* be the value of HandleType and let *H* be the value of Handle.

2) If *HT* is not one of the code values in Table 14, "Codes used for SQL/CLI handle types", then an exception condition is raised: *CLI-specific condition — invalid handle*.

3) Case:

   a) If *HT* indicates ENVIRONMENT HANDLE, then:

      i) If *H* does not identify an allocated SQL-environment, then an exception condition is raised: *CLI-specific condition — invalid handle*.

      ii) Let *E* be the allocated SQL-environment identified by *H*.

      iii) The diagnostics area associated with *E* is emptied.

      iv) If an allocated SQL-connection is associated with *E*, then an exception condition is raised: *CLI-specific condition — function sequence error*.

      v) *E* is deallocated and all its resources are freed.

   b) If *HT* indicates CONNECTION HANDLE, then:

      i) If *H* does not identify an allocated SQL-connection, then an exception condition is raised: *CLI-specific condition — invalid handle*.

      ii) Let *C* be the allocated SQL-connection identified by *H*.

      iii) The diagnostics area associated with *C* is emptied.

      iv) If an established SQL-connection is associated with *C*, then an exception condition is raised: *CLI-specific condition — function sequence error*.

      v) *C* is deallocated and all its resources are freed.

   c) If *HT* indicates STATEMENT HANDLE, then:

      i) If *H* does not identify an allocated SQL-statement, then an exception condition is raised: *CLI-specific condition — invalid handle*.

ii) Let *S* be the allocated SQL-statement identified by *H*.

iii) The diagnostics area associated with *S* is emptied.

iv) Let *C* be the allocated SQL-connection with which *S* is associated and let *EC* be the established SQL-connection associated with *C*.

v) If *EC* is not the current SQL-connection, then the General Rules of Subclause 5.3, “Implicit set connection”, are applied with EC as *dormant SQL-connection*.

vi) If there is a deferred parameter number associated with *S*, then an exception condition is raised: *CLI-specific condition — function sequence error*.

vii) If there is an open cursor associated with *S*, then:

1) The open cursor associated with *S* is placed in the closed state and its copy of the select source is destroyed.

2) Any fetched row associated with *S* is removed from association with *S*.

viii) The automatically allocated CLI descriptor areas associated with *S* are deallocated and all their resources are freed.

ix) *S* is deallocated and all its resources are freed.

d) If *HT* indicates DESCRIPTOR HANDLE, then:

i) If *H* does not identify an allocated CLI descriptor area, then an exception condition is raised: *CLI-specific condition — invalid handle*.

ii) Let *D* be the allocated CLI descriptor area identified by *H*.

iii) The diagnostics area associated with *D* is emptied.

iv) Let *C* be the allocated SQL-connection with which *D* is associated and let *EC* be the established SQL-connection associated with *C*.

v) If *EC* is not the current SQL-connection, then the General Rules of Subclause 5.3, “Implicit set connection”, are applied with EC as *dormant SQL-connection*.

vi) The General Rules of Subclause 5.11, “Deferred parameter check”, are applied to *D* as the DESCRIPTOR AREA.

vii) Let *AT* be the value of the ALLOC_TYPE field of *D*.

viii) If *AT* indicates AUTOMATIC, then an exception condition is raised: *CLI-specific condition — invalid use of automatically-allocated descriptor handle*.

ix) Let *L1* be a list of allocated SQL-statements associated with *C* for which *D* is the current application row descriptor. For each allocated SQL-statement *S* in *L1*, the automatically-allocated application row descriptor associated with *S* becomes the current application row descriptor for *S*.

x) Let *L2* be a list of allocated SQL-statements associated with *C* for which *D* is the current application parameter descriptor. For each allocated SQL-statement *S* in *L2*, the automatically-allocated application parameter descriptor associated with *S* becomes the current application parameter descriptor for *S*.

xi) *D* is deallocated and all its resources are freed.

6.27 FreeStmt

Function

Deallocate an SQL-statement.

Definition

```
FreeStmt (
    StatementHandle          IN   INTEGER ,
    Option                   IN   SMALLINT )
    RETURNS SMALLINT
```

General Rules

1) Let *SH* be the value of StatementHandle and let *S* be the allocated SQL-statement identified by *SH*.

2) Let *OPT* be the value of Option.

3) If *OPT* is not one of the codes in Table 19, “Codes used for FreeStmt options”, then an exception condition is raised: *CLI-specific condition — invalid attribute identifier*.

4) Let *ARD* be the current application row descriptor for *S* and let *RC* be the value of the COUNT field of *ARD*.

5) Let *APD* be the current application parameter descriptor for *S* and let *PC* be the value of the COUNT field of *APD*.

6) Case:

   a) If *OPT* indicates CLOSE CURSOR and there is an open cursor associated with *S*, then:

      i) The open cursor associated with *S* is placed in the closed state and its copy of the select source is destroyed.

      ii) Any fetched row associated with *S* is removed from association with *S*.

   b) If *OPT* indicates FREE HANDLE, then FreeHandle is implicitly invoked with HandleType indicating STATEMENT HANDLE and with *SH* as the value of Handle.

   c) If *OPT* indicates UNBIND COLUMNS, then for each of the first *RC* item descriptor areas of *ARD*, the value of the DATA_POINTER field is set to zero.

   d) If *OPT* indicates UNBIND PARAMETERS, then for each of the first *PC* item descriptor areas of *APD*, the value of the DATA_POINTER field is set to zero.

   e) If *OPT* indicates REALLOCATE, then the following objects associated with *S* are destroyed:

      i) Any prepared statement.

      ii) Any cursor.

      iii) Any select source.

iv) Any executed statement.

and the original automatically allocated descriptors are associated with the allocated SQL-statement with their original default values as described in the General Rules of Subclause 6.3, “AllocHandle”.

6.28 GetConnectAttr

Function

Get the value of an SQL-connection attribute.

Definition

```
GetConnectAttr (
    ConnectionHandle     IN        INTEGER,
    Attribute            IN        INTEGER,
    Value                OUT       ANY,
    BufferLength         IN        INTEGER,
    StringLength         OUT       INTEGER )
    RETURNS SMALLINT
```

General Rules

1) Case:

   a) If ConnectionHandle does not identify an allocated SQL-connection, then an exception condition is raised: *CLI-specific condition — invalid handle*.

   b) Otherwise:

      i) Let *C* be the allocated SQL-connection identified by ConnectionHandle.

      ii) The diagnostics area associated with *C* is emptied.

2) Let *A* be the value of Attribute.

3) If *A* is not one of the code values in Table 17, “Codes used for connection attributes”, then an exception condition is raised: *CLI-specific condition — invalid attribute identifier*.

4) If *A* indicates POPULATE IPD, then

   Case:

   a) If there is no established SQL-connection associated with *C*, then an exception condition is raised: *connection exception — connection does not exist*.

   b) Otherwise:

      i) If POPULATE IPD for *C* is <u>*True*</u>, then Value is set to 1 (one).

      ii) If POPULATE IPD for *C* is <u>*False*</u>, then Value is set to 0 (zero).

5) If *A* indicates SAVEPOINT NAME, then:

   a) Let *BL* be the value of BufferLength.

   b) Let *AV* be the value of the SAVEPOINT NAME connection attribute.

c) The General Rules of Subclause 5.9, “Character string retrieval”, are applied with Value, *AV*, *BL*, and StringLength as TARGET, VALUE, TARGET OCTET LENGTH, and RETURNED OCTET LENGTH, respectively.

6) If *A* specifies an implementation-defined connection attribute, then

Case:

a) If the data type for the connection attribute is specified in Table 20, “Data types of attributes”, as INTEGER, then Value is set to the value of the implementation-defined connection attribute.

b) Otherwise:

i) Let *BL* be the value of BufferLength.

ii) Let *AV* be the value of the implementation-defined connection attribute.

iii) The General Rules of Subclause 5.9, “Character string retrieval”, are applied with Value, *AV*, *BL*, and StringLength as *TARGET*, *VALUE*, *TARGET OCTET LENGTH*, and *RETURNED OCTET LENGTH*, respectively.

6.29 GetCursorName

Function

Get a cursor name.

Definition

```
GetCursorName (
    StatementHandle  IN   INTEGER,
    CursorName       OUT  CHARACTER(L),
    BufferLength     IN   SMALLINT,
    NameLength       OUT  SMALLINT )
    RETURNS SMALLINT
```

where *L* has a maximum value equal to the implementation-defined maximum length of a variable-length character string.

General Rules

1) Let *S* be the allocated SQL-statement identified by StatementHandle.

2) Case:

   a) If there is no cursor name associated with *S*, then a unique implementation-dependent name that has the prefix 'SQLCUR' or the prefix 'SQL_CUR' becomes the cursor name associated with *S*; let *CN* be that associated cursor name.

   b) Otherwise, let *CN* be the cursor name associated with *S*.

3) Let *BL* be the value of BufferLength.

4) The General Rules of Subclause 5.9, "Character string retrieval", are applied with CursorName, *CN*, *BL*, and NameLength as *TARGET*, *VALUE*, *TARGET OCTET LENGTH*, and *RETURNED OCTET LENGTH*, respectively.

6.30 GetData

Function

Retrieve a column value.

Definition

```
GetData (
    StatementHandle      IN        INTEGER,
    ColumnNumber         IN        SMALLINT,
    TargetType           IN        SMALLINT,
    TargetValue          OUT       ANY,
    BufferLength         IN        INTEGER,
    StrLen_or_Ind        OUT       INTEGER )
    RETURNS SMALLINT
```

General Rules

1) Let *S* be the allocated SQL-statement identified by StatementHandle.

2) Case:

   a) If there is no fetched rowset associated with *S*, then an exception condition is raised: *CLI-specific condition — function sequence error*.

   b) If the fetched rowset associated with *S* is empty, then a completion condition is raised: *no data*, TargetValue and StrLen_or_Ind are set to implementation-dependent values, and no further rules of this Subclause are applied.

   c) Otherwise, let *R* be the fetched rowset associated with *S*.

3) Let *ARD* be the current application row descriptor for *S* and let *N* be the value of the TOP_LEVEL_COUNT field of *ARD*.

4) Let *AS* be the value of the ARRAY_SIZE field in the header of *ARD*. Let *P* be the value of the attribute CURRENT OF POSITION of *S*.

5) If *P* is greater than *AS*, the *P*-th row in *R* has not been fetched, or the value of the CURSOR SCROLLABLE attribute of *S* is NONSCROLLABLE and *AS* is greater than 1 (one), then an exception condition is raised: *CLI-specific condition — invalid cursor position*.

6) Let *FR* be the *P*-th row of *R*.

7) Let *D* be the degree of the table defined by the select source associated with *S*.

8) If *N* is less than zero, then an exception condition is raised: *dynamic SQL error — invalid descriptor count*.

9) Let *CN* be the value of ColumnNumber.

10) If *CN* is less than 1 (one) or greater than *D*, then an exception condition is raised: *dynamic SQL error — invalid descriptor index*.

11) If DATA_POINTER is non-zero for at least one of the first *N* item descriptor areas of *ARD* for which LEVEL is 0 (zero) and the value of TYPE is neither ROW, ARRAY, nor MULTISET, then let *BCN* be the column number associated with such an item descriptor area and let *HBCN* be the value of MAX(*BCN*). Otherwise, let *HBCN* be zero.

12) Let *IDA* be the item descriptor area of *ARD* specified by *CN*. If the value of TYPE in *IDA* is either ROW, ARRAY, or MULTISET, or if the LEVEL of *IDA* is greater than 0 (zero), then an exception condition is raised: *dynamic SQL error — invalid descriptor index*.

   NOTE 29 — GetData cannot be called to retrieve the data corresponding to a subordinate descriptor record such as, for example, from an individual field of a ROW type.

13) If *CN* is not greater than *HBCN*, then

   Case:

   a) If the DATA_POINTER field of *IDA* is not zero, then an exception condition is raised: *dynamic SQL error — invalid descriptor index*.

   b) If the DATA_POINTER field of *IDA* is zero, then it is implementation-defined whether an exception condition is raised: *dynamic SQL error — invalid descriptor index*.

      NOTE 30 — This implementation-defined feature determines whether columns before the highest bound column can be accessed by GetData.

14) If there is a fetched column number associated with *FR*, then let *FCN* be that column number; otherwise, let *FCN* be zero.

   NOTE 31 — "fetched column number" is the ColumnNumber value used with the previous invocation (if any) of the GetData routine with *FR*. See the General Rules later in this Subclause where this value is set.

15) Case:

   a) If *FCN* is greater than zero and *CN* is not greater than *FCN*, then it is implementation-defined whether an exception condition is raised: *dynamic SQL error — invalid descriptor index*.

      NOTE 32 — This implementation-defined feature determines whether GetData can only access columns in ascending column number order.

   b) If *FCN* is less than zero, then:

      i) Let *AFCN* be the absolute value of *FCN*.

      ii) Case:

         1) If *CN* is less than *AFCN*, then it is implementation-defined whether an exception condition is raised: *dynamic SQL error — invalid descriptor index*.

            NOTE 33 — This implementation-defined feature determines whether GetData can only access columns in ascending column number order.

         2) If *CN* is greater than *AFCN*, then let *FCN* be *AFCN*.

16) Let *T* be the value of TargetType.

17) Let *HL* be the programming language of the invoking host program. Let *operative data type correspondence table* be the data type correspondence table for *HL* as specified in Subclause 5.15, "SQL/CLI data type correspondences". Refer to the two columns of the operative data type correspondence table as the *SQL data type column* and the *host data type column*.

18) If either of the following is true, then an exception condition is raised: *CLI-specific condition — invalid data type in application descriptor*.

    a) *T* indicates neither DEFAULT nor ARD TYPE and is not one of the code values in Table 8, “Codes used for application data types in SQL/CLI”.

    b) *T* is one of the code values in Table 8, “Codes used for application data types in SQL/CLI”, but the row that contains the corresponding SQL data type in the SQL data type column of the operative data type correspondence table contains 'None' in the host data type column.

19) If *T* does not indicate ARD TYPE, then the data type of the <target specification> described by *IDA* is set to *T*.

20) Let *IRD* be the implementation row descriptor associated with *S*.

21) If the value of the TYPE field of *IDA* indicates DEFAULT, then:

    a) Let *CT*, *P*, and *SC* be the values of the TYPE, PRECISION, and SCALE fields, respectively, for the *CN*-th item descriptor area of *IRD* for which LEVEL is 0 (zero).

    b) The data type, precision, and scale of the <target specification> described by *IDA* are set to *CT*, *P*, and *SC*, respectively, for the purposes of this GetData invocation only.

22) If *IDA* is not valid as specified in Subclause 5.13, “Description of CLI item descriptor areas”, then an exception condition is raised: *dynamic SQL error — using clause does not match target specifications*.

23) Let *TT* be the value of the TYPE field of *IDA*.

24) Case:

    a) If *TT* indicates CHARACTER, then:

        i) Let *UT* be the code value corresponding to CHARACTER VARYING as specified in Table 7, “Codes used for implementation data types in SQL/CLI”.

        ii) Let *CL* be the implementation-defined maximum length for a CHARACTER VARYING data type.

    b) Otherwise, let *UT* be *TT* and let *CL* be zero.

25) Case:

    a) If *FCN* is less than zero, then

        Case:

        i) If *TT* does not indicate CHARACTER, CHARACTER LARGE OBJECT, BINARY, BINARY VARYING, or BINARY LARGE OBJECT, then *AFCN* becomes the fetched column number associated with the fetched row associated with *S* and an exception condition is raised: *dynamic SQL error — invalid descriptor index*.

        ii) Otherwise, let *FL*, *DV*, and *DL* be the fetched length, data value and data length, respectively, associated with *FCN* and let *TV* be the result of the <string value function>:

        ```
        SUBSTRING ( DV FROM (FL+1) )
        ```

    b) Otherwise:

i) Let *FL* be zero.

ii) Let *SDT* be the effective data type of the *CN*-th <select list> column as represented by the values of the TYPE, LENGTH, PRECISION, SCALE, DATETIME_INTERVAL_CODE, DATETIME_INTERVAL_PRECISION, CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, CHARACTER_SET_NAME, USER_DEFINED_TYPE_CATALOG, USER_DEFINED_TYPE_SCHEMA, USER_DEFINED_TYPE_NAME, SCOPE_CATALOG, SCOPE_SCHEMA, and SCOPE_NAME fields in the *CN*-th item descriptor area of *IRD*. Let *SV* be the value of the <select list> column, with data type *SDT*.

iii) If TYPE indicates USER-DEFINED TYPE, then let the most specific type of the *CN*-th <select list> column whose value is *SV* be represented by the values of the SPECIFIC_TYPE_CATALOG, SPECIFIC_TYPE_SCHEMA, and SPECIFIC_TYPE_NAME fields in the corresponding item descriptor area of *IRD*.

iv) Let *TDT* be the effective data type of the *CN*-th <target specification> as represented by the type *UT*, the length value *CL*, and the values of the PRECISION, SCALE, CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, CHARACTER_SET_NAME, USER_DEFINED_TYPE_CATALOG, USER_DEFINED_TYPE_SCHEMA, USER_DEFINED_TYPE_NAME, SCOPE_CATALOG, SCOPE_SCHEMA, and SCOPE_NAME fields of *IDA*.

v) Let *LTDT* be the data type on the last retrieval of the *CN*-th <target specification>, if any. If any of the following is true, then it is implementation-defined whether or not exception condition is raised: *dynamic SQL error — restricted data type attribute violation*.

   1) If *LTDT* and *TDT* both identify a binary large object type and only one of *LTDT* and *TDT* is a binary large object locator.

   2) If *LTDT* and *TDT* both identify a character large object type and only one of *LTDT* and *TDT* is a character large object locator.

   3) If *LTDT* and *TDT* both identify an array type and only one of *LTDT* and *TDT* is an array locator.

   4) If *LTDT* and *TDT* both identify a multiset type and only one of *LTDT* and *TDT* is a multiset locator.

   5) If *LTDT* and *TDT* both identify a user-defined type and only one of *LTDT* and *TDT* is a user-defined type locator.

vi) Case:

   1) If *TDT* is a locator type, then

      Case:

      A) If *SV* is not the null value, then a locator *L* that uniquely identifies *SV* is generated and the value *TV* of the *CN*-th <target specification> is set to an implementation-dependent four-octet value that represents *L*.

      B) Otherwise, the value *TV* of the *CN*-th <target specification> is the null value.

   2) If *SDT* and *TDT* are predefined data types, then

      Case:

A) If the <cast specification>

```
CAST ( SV AS TDT )
```

does not conform to the Syntax Rules of Subclause 6.12, “<cast specification>”, in ISO/IEC 9075-2, and there is an implementation-defined conversion from type *SDT* to type *TDT*, then that implementation-defined conversion is effectively performed, converting *SV* to type *TDT*, and the result is the value *TV* of the *CN*-th <target specification>.

B) Otherwise:

I) If the <cast specification>

```
CAST ( SV AS TDT )
```

does not conform to the Syntax Rules of Subclause 6.12, “<cast specification>”, in ISO/IEC 9075-2, then an exception condition is raised: *dynamic SQL error — restricted data type attribute violation*.

II) The <cast specification>

```
CAST ( SV AS TDT )
```

is effectively performed, and the result is the value *TV* of the *CN*-th <target specification>.

3) If *SDT* is a user-defined type and *TDT* is a predefined data type, then:

A) Let *DT* be the data type identified by *SDT*.

B) If the current SQL-session has a group name corresponding to the user-defined name of *DT*, then let *GN* be that group name; otherwise, let *GN* be the default transform group name associated with the current SQL-session.

C) The Syntax Rules of Subclause 9.19, “Determination of a from-sql function”, in ISO/IEC 9075-2, are applied with *DT* and *GN* as *TYPE* and *GROUP*, respectively.

Case:

I) If there is an applicable from-sql function, then let *FSF* be that from-sql function and let *FSFRT* be the <returns data type> of *FSF*.

Case:

1) If *FSFRT* is compatible with *TD*T, then the from-sql function *TSF* is effectively invoked with *SV* as its input parameter and the <return value> is the value *TV* of the *CN*-th <target specification>.

2) Otherwise, an exception condition is raised: *dynamic SQL error — restricted data type attribute violation*.

II) Otherwise, an exception condition is raised: *dynamic SQL error — data type transform function violation*.

26) *CN* becomes the fetched column number associated with the fetched row associated with *S*.

27) If *TV* is the null value, then

Case:

a) If StrLen_or_Ind is a null pointer, then an exception condition is raised: *data exception — null value, no indicator parameter*.

b) Otherwise, StrLen_or_Ind is set to the appropriate 'Code' for SQL NULL DATA in Table 27, “Miscellaneous codes used in CLI”, and the value of TargetValue is implementation-dependent.

28) Let *OL* be the value of BufferLength.

29) If null termination is *True* for the current SQL-environment, then let *NB* be the length in octets of a null terminator in the character set of the *i*-th bound target; otherwise let *NB* be 0 (zero).

30) If *TV* is not the null value, then:

a) StrLen_or_Ind is set to 0 (zero).

b) Case:

i) If *TT* does not indicate CHARACTER, CHARACTER LARGE OBJECT, BINARY, BINARY VARYING, or BINARY LARGE OBJECT, then TargetValue is set to *TV*.

ii) Otherwise:

1) If *TT* is CHARACTER or CHARACTER LARGE OBJECT, then:

A) If *TV* is a zero-length character string, then it is implementation-defined whether or not an exception condition is raised: *data exception — zero-length character string*.

B) The General Rules of Subclause 5.9, “Character string retrieval”, are applied with TargetValue, *TV*, *OL*, and StrLen_or_Ind as *TARGET*, *VALUE*, *OCTET LENGTH*, and *RETURNED OCTET LENGTH*, respectively.

2) If *TT* is BINARY, BINARY VARYING, or BINARY LARGE OBJECT, then the General Rules of Subclause 5.10, “Binary string retrieval”, are applied with TargetValue, *TV*, *OL*, and StrLen_or_Ind as *TARGET*, *VALUE*, *OCTET LENGTH*, and *RETURNED OCTET LENGTH*, respectively.

3) If *FCN* is not less than zero, then let *DV* be *TV* and let *DL* be the length of *TV* in octets.

4) Let *FL* be (*FL*+*OL*–*NB*).

5) If *FL* is less than *DL*, then –*CN* becomes the fetched column number associated with the fetched row associated with *S* and *FL*, *DV* and *DL* become the fetched length, data value, and data length, respectively, associated with the fetched column number.

6.31 GetDescField

Function

Get a field from a CLI descriptor area.

Definition

```
GetDescField (
    DescriptorHandle     IN       INTEGER,
    RecordNumber         IN       SMALLINT,
    FieldIdentifier      IN       SMALLINT,
    Value                OUT      ANY,
    BufferLength         IN       INTEGER,
    StringLength         OUT      INTEGER )
    RETURNS SMALLINT
```

General Rules

1) Let *D* be the allocated CLI descriptor area identified by DescriptorHandle and let *N* be the value of the COUNT field of *D*.

2) Let *FI* be the value of FieldIdentifier.

3) If *FI* is not one of the code values in Table 21, “Codes used for SQL/CLI descriptor fields”, then an exception condition is raised: *CLI-specific condition — invalid descriptor field identifier*.

4) Let *RN* be the value of RecordNumber.

5) Let *TYPE* be the value of the Type column in the row of Table 21, “Codes used for SQL/CLI descriptor fields”, that contains *FI*.

6) The General Rules of Subclause 5.11, “Deferred parameter check”, are applied to *D* as the DESCRIPTOR AREA.

7) If *TYPE* is 'ITEM', then:

   a) If *RN* is less than 1 (one), then an exception condition is raised: *dynamic SQL error — invalid descriptor index*.

   b) If *RN* is greater than *N*, then a completion condition is raised: *no data*.

8) If *D* is an implementation row descriptor, then let *S* be the allocated SQL-statement associated with *D*.

9) Let *MBR* be the value of the May Be Retrieved column in the row of Table 23, “Ability to retrieve SQL/CLI descriptor fields”, that contains *FI* and the column that contains the descriptor type *D*.

10) If *MBR* is 'PS' and there is no prepared or executed statement associated with *S*, then an exception condition is raised: *CLI-specific condition — associated statement is not prepared*.

11) If *MBR* is 'No', then an exception condition is raised: *CLI-specific condition — invalid descriptor field identifier*.

12) If *FI* indicates a descriptor field whose value is the initially undefined value created when the descriptor was created, then an exception condition is raised: *CLI-specific condition — invalid descriptor field identifier*.

13) Let *IDA* be the item descriptor area of *D* specified by *RN*.

14) If *TYPE* is 'HEADER', then header information from the descriptor area *D* is retrieved as follows.

    Case:

    a) If *FI* indicates COUNT, then the value retrieved is *N*.

    b) If *FI* indicates ALLOC_TYPE, then the value retrieved is the allocation type for *D*.

    c) If *FI* indicates an implementation-defined descriptor header field, then the value retrieved is the value of the implementation-defined descriptor header field identified by *FI*.

    d) Otherwise, if *FI* indicates a descriptor header field defined in Table 21, “Codes used for SQL/CLI descriptor fields”, then the value retrieved is the value of the descriptor header field identified by *FI*.

15) If *TYPE* is 'ITEM', then item information from the descriptor area *D* is retrieved as follows:

    Case:

    a) If *FI* indicates an implementation-defined descriptor item field, then the value retrieved is the value of the implementation-defined descriptor item field of *IDA* identified by *FI*.

    b) Otherwise, if *FI* indicates a descriptor item field defined in Table 21, “Codes used for SQL/CLI descriptor fields”, then the value retrieved is the value of the descriptor item field of *IDA* identified by *FI*.

16) Let *V* be the value retrieved.

17) If *FI* indicates a descriptor field whose row in Table 6, “Fields in SQL/CLI row and parameter descriptor areas”, contains a Data Type that is not CHARACTER VARYING, then Value is set to *V* and no further rules of this Subclause are applied.

18) Let *BL* be the value of BufferLength.

19) If *FI* indicates a descriptor field whose row in Table 6, “Fields in SQL/CLI row and parameter descriptor areas”, contains a Data Type that is CHARACTER VARYING, then the General Rules of Subclause 5.9, “Character string retrieval”, are applied with Value, *V*, *BL*, and StringLength as *TARGET*, *VALUE*, *TARGET OCTET LENGTH*, and *RETURNED OCTET LENGTH*, respectively.

6.32 GetDescRec

Function

Get commonly-used fields from a CLI descriptor area.

Definition

```
GetDescRec (
    DescriptorHandle     IN        INTEGER,
    RecordNumber         IN        SMALLINT,
    Name                 OUT       CHARACTER(L),
    BufferLength         IN        SMALLINT,
    NameLength           OUT       SMALLINT,
    Type                 OUT       SMALLINT,
    SubType              OUT       SMALLINT,
    Length               OUT       INTEGER,
    Precision            OUT       SMALLINT,
    Scale                OUT       SMALLINT,
    Nullable             OUT       SMALLINT )
    RETURNS SMALLINT
```

where *L* has a maximum value equal to the implementation-defined maximum length of a variable-length character string.

General Rules

1) Let *D* be the allocated CLI descriptor area identified by DescriptorHandle and let *N* be the value of the COUNT field of *D*.

2) The General Rules of Subclause 5.11, “Deferred parameter check”, are applied to *D* as the DESCRIPTOR AREA.

3) Let *RN* be the value of RecordNumber.

4) Case:

   a) If *RN* is less than 1 (one), then an exception condition is raised: *dynamic SQL error — invalid descriptor index*.

   b) Otherwise, if *RN* is greater than *N*, then a completion condition is raised: *no data*.

5) If *D* is an implementation row descriptor associated with an allocated SQL-statement *S* and there is no prepared or executed statement associated with *S*, then an exception condition is raised: *CLI-specific condition — associated statement is not prepared*.

6) Let *ITEM* be the <dynamic parameter specification> or <select list> column (or part thereof, if the item descriptor area of *D* is a subordinate descriptor) described by the item descriptor area of *D* specified by *RN*.

7) Let *BL* be the value of BufferLength.

8) Information is retrieved from *D*:

   a) If Type is not a null pointer, then Type is set to the value of the TYPE field of *ITEM*.

   b) If SubType is not a null pointer, then SubType is set to the value of the DATETIME_INTERVAL_CODE field of *ITEM*.

   c) If Length is not a null pointer, then Length is set to value of the OCTET_LENGTH field of *ITEM*.

   d) If Precision is not a null pointer, then Precision is set to the value of the PRECISION field of *ITEM*.

   e) If Scale is not a null pointer, then Scale is set to the value of the SCALE field of *ITEM*.

   f) If Nullable is not a null pointer, then Nullable is set to the value of the NULLABLE field of *ITEM*.

   g) If Name is not a null pointer, then

      Case:

      i) If null termination is *False* for the current SQL-environment and *BL* is zero, then no further rules of this Subclause are applied.

      ii) Otherwise:

         1) The value retrieved is the value of the NAME field of *ITEM*.

         2) Let *V* be the value retrieved.

         3) The General Rules of Subclause 5.9, "Character string retrieval", are applied with Name, *V*, *BL*, and NameLength as *TARGET*, *VALUE*, *TARGET OCTET LENGTH*, and *RETURNED OCTET LENGTH*, respectively.

6.33 GetDiagField

Function

Get information from a CLI diagnostics area.

Definition

```
GetDiagField (
    HandleType          IN        SMALLINT,
    Handle              IN        INTEGER,
    RecordNumber        IN        SMALLINT,
    DiagIdentifier      IN        SMALLINT,
    DiagInfo            OUT       ANY,
    BufferLength        IN        SMALLINT,
    StringLength        OUT       SMALLINT )
    RETURNS SMALLINT
```

General Rules

1) Let *HT* be the value of HandleType.

2) If *HT* is not one of the code values in Table 14, “Codes used for SQL/CLI handle types”, then an exception condition is raised: *CLI-specific condition — invalid handle*.

3) Case:

   a) If *HT* indicates ENVIRONMENT HANDLE and Handle does not identify an allocated SQL-environment, then an exception condition is raised: *CLI-specific condition — invalid handle*.

   b) If *HT* indicates CONNECTION HANDLE and Handle does not identify an allocated SQL-connection, then an exception condition is raised: *CLI-specific condition — invalid handle*.

   c) If *HT* indicates STATEMENT HANDLE and Handle does not identify an allocated SQL-statement, then an exception condition is raised: *CLI-specific condition — invalid handle*.

   d) If *HT* indicates DESCRIPTOR HANDLE and Handle does not identify an allocated CLI descriptor area, then an exception condition is raised: *CLI-specific condition — invalid handle*.

4) Let *DI* be the value of DiagIdentifier.

5) If *DI* is not one of the code values in Table 13, “Codes used for SQL/CLI diagnostic fields”, then an exception condition is raised: *CLI-specific condition — invalid attribute value*.

6) Let *TYPE* be the value of the Type column in the row that contains *DI* in Table 13, “Codes used for SQL/CLI diagnostic fields”.

7) Let *RN* be the value of RecordNumber.

8) Let *R* be the most recently executed CLI routine, other than GetDiagRec, GetDiagField, or Error, for which Handle was passed as the value of an input handle and let *N* be the number of status records generated by the execution of *R*.

NOTE 34 — The GetDiagRec, GetDiagField, and Error routines may cause exception or completion conditions to be raised, but they do not cause diagnostic information to be generated.

9) If *TYPE* is 'STATUS', then:

   a) If *RN* is less than 1 (one), then an exception condition is raised: *invalid condition number*.

   b) If *RN* is greater than *N*, then a completion condition is raised: *no data*, and no further rules of this Subclause are applied.

10) If *DI* indicates ROW_COUNT and *R* is neither Execute nor ExecDirect, then an exception condition is raised: *CLI-specific condition — invalid attribute identifier*.

11) If *TYPE* is 'HEADER', then header information from the diagnostics area associated with the resource identified by Handle is retrieved.

   a) If *DI* indicates NUMBER, then the value retrieved is *N*.

   b) If *DI* indicates DYNAMIC_FUNCTION, then

      Case:

      i) If no SQL-statement was being prepared or executed by *R*, then the value retrieved is a zero-length string.

      ii) Otherwise, the value retrieved is the character identifier of the SQL-statement being prepared or executed by *R*. The value DYNAMIC_FUNCTION values are specified in Table 32, “SQL-statement codes”, in ISO/IEC 9075-2.

         NOTE 35 — Additional valid DYNAMIC_FUNCTION values may be defined in other parts of ISO/IEC 9075.

   c) If *DI* indicates DYNAMIC_FUNCTION_CODE, then

      Case:

      i) If no SQL-statement was being prepared or executed by *R*, then the value retrieved is 0 (zero).

      ii) Otherwise, the value retrieved is the integer identifier of the SQL-statement being prepared or executed by *R*. The value DYNAMIC_FUNCTION_CODE values are specified in Table 32, “SQL-statement codes”, in ISO/IEC 9075-2.

         NOTE 36 — Additional valid DYNAMIC_FUNCTION_CODE values may be defined in other parts of ISO/IEC 9075.

   d) If *DI* indicates RETURNCODE, then the value retrieved is the code indicating the basic result of the execution of *R*. Subclause 4.2, “Return codes”, specifies the code values and their meanings.

      NOTE 37 — The value retrieved will never indicate **Invalid handle** or **Data needed**, since no diagnostic information is generated if this is the basic result of the execution of *R*.

   e) If *DI* indicates ROW_COUNT, the value retrieved is the number of rows affected as the result of executing a <delete statement: searched>, <insert statement>, <merge statement>, or <update statement: searched> as a direct result of the execution of the SQL-statement executed by *R*. Let *S* be the <delete statement: searched>, <insert statement>, <merge statement>, or <update statement: searched>. Let *T* be the table identified by the <table name> directly contained in *S*.

      Case:

      i) If *S* is an <insert statement>, then the value retrieved is the number of rows inserted into *T*.

ii) If *S* is a <merge statement>, then let *TR1* be the <target table> immediately contained in *S*, let *TR2* be the <table reference> immediately contained in *S*, and let *SC* be the <search condition> immediately contained in *S*. If <merge correlation name> is specified, let *MCN* be “AS <merge correlation name>”; otherwise, let *MCN* be a zero-length string.

Case:

1) If *S* contains a <merge when matched clause> and does not contain a <merge when not matched clause>, then the value retrieved is effectively derived by executing the statement:

```
SELECT COUNT (*)
FROM TR1 MCN, TR2
WHERE SC
```

before the execution of *S*.

2) If *S* contains a <merge when not matched clause> and does not contain a <merge when matched clause>, then the value retrieved is effectively derived by executing the statement:

```
( SELECT COUNT(*)
  FROM TR1 MCN
    RIGHT OUTER JOIN
       TR2
      ON SC )
-
( SELECT COUNT (*)
  FROM TR1 MCN, TR2
  WHERE SC )
```

before the execution of *S*.

3) If *S* contains both a <merge when matched clause> and a <merge when not matched clause>, then the value retrieved is effectively derived by executing the statement:

```
SELECT COUNT(*)
FROM TR1 MCN
  RIGHT OUTER JOIN
     TR2
    ON SC
```

before the execution of *S*.

iii) If *S* is a <delete statement: searched> or an <update statement: searched>, then

Case:

1) If *S* does not contain a <search condition>, then the value retrieved is the cardinality of *T* before the execution of *S*.

2) Otherwise, let *SC* be the <search condition> directly contained in *S*. The value retrieved is effectively derived by executing the statement:

```
SELECT COUNT(*)
FROM T
WHERE SC
```

before the execution of *S*.

The value retrieved following the execution by *R* of an SQL-statement that does not directly result in the execution of a <delete statement: searched>, <insert statement>, <merge statement>, or <update statement: searched> is implementation-dependent.

f) If *DI* indicates MORE, then the value retrieved is

Case:

i) If more conditions were raised during execution of *R* than have been stored in the diagnostics area, then 1 (one).

ii) If all the conditions that were raised during execution of *R* have been stored in the diagnostics area, then 0 (zero).

g) If *DI* indicates TRANSACTIONS_COMMITTED, then the value retrieved is the number of SQL-transactions that have been committed since the most recent time at which the diagnostics area for *HT* was emptied.

NOTE 38 — See the General Rules of Subclause 13.3, "<externally-invoked procedure>", and Subclause 13.4, "Calls to an <externally-invoked procedure>", in ISO/IEC 9075-2. TRANSACTIONS_COMMITTED indicates the number of SQL-transactions that were committed during the invocation of an external routine.

h) If *DI* indicates TRANSACTIONS_ROLLED_BACK, then the value retrieved is the number of SQL-transactions that have been rolled back since the most recent time at which the diagnostics area for *HT* was emptied.

NOTE 39 — See the General Rules of Subclause 13.3, "<externally-invoked procedure>", and Subclause 13.4, "Calls to an <externally-invoked procedure>", in ISO/IEC 9075-2. TRANSACTIONS_ROLLED_BACK indicates the number of SQL-transactions that were rolled back during the invocation of an external routine.

i) If *DI* indicates TRANSACTION_ACTIVE, then the value retrieved is 1 (one) if an SQL-transaction is currently active and is 0 (zero) if an SQL-transaction is not currently active.

NOTE 40 — TRANSACTION_ACTIVE indicates whether an SQL-transaction is active upon return from an external routine.

j) If *DI* indicates an implementation-defined diagnostics header field, then the value retrieved is the value of the implementation-defined diagnostics header field.

12) If *TYPE* is 'STATUS', then information from the *RN*-th status record in the diagnostics area associated with the resource identified by Handle is retrieved.

a) If *DI* indicates CONDITION_NUMBER, then the value retrieved is *RN*.

b) If *DI* indicates SQLSTATE, then the value retrieved is the SQLSTATE value corresponding to the status condition.

c) If *DI* indicates NATIVE_CODE, then the value retrieved is the implementation-defined native error code corresponding to the status condition.

d) If *DI* indicates MESSAGE_TEXT, then the value retrieved is

Case:

i) If the value of SQLSTATE corresponds to *external routine invocation exception*, *external routine exception*, or *warning*, then the message text item of the SQL-invoked routine that raised the exception condition.

ii) Otherwise, an implementation-defined character string.

NOTE 41 — An implementation may provide <space>s or a zero-length string or a character string that describes the status condition.

e) If *DI* indicates MESSAGE_LENGTH, then the value retrieved is the length in characters of the character string value of MESSAGE_TEXT corresponding to the status condition.

f) If *DI* indicates MESSAGE_OCTET_LENGTH, then the value retrieved is the length in octets of the character string value of MESSAGE_TEXT corresponding to the status condition.

g) If *DI* indicates CLASS_ORIGIN, then the value retrieved is the identification of the naming authority that defined the class value of the SQLSTATE value corresponding to the status condition. That value shall be 'ISO 9075' if the class value is fully defined in Subclause 24.1, “SQLSTATE”, in ISO/IEC 9075-2 or Subclause 5.12, “CLI-specific status codes”, and shall be an implementation-defined character string other than 'ISO 9075' for any implementation-defined class value.

h) If *DI* indicates SUBCLASS_ORIGIN, then the value retrieved is the identification of the naming authority that defined the subclass value of the SQLSTATE value corresponding to the status condition. That value shall be 'ISO 9075' if the subclass value is fully defined in Subclause 24.1, “SQLSTATE”, in ISO/IEC 9075-2, or Subclause 5.12, “CLI-specific status codes”, and shall be an implementation-defined character string other than 'ISO 9075' for any implementation-defined subclass value.

i) If *DI* indicates CURSOR_NAME, CONSTRAINT_CATALOG, CONSTRAINT_SCHEMA, CONSTRAINT_NAME, CATALOG_NAME, SCHEMA_NAME, TABLE_NAME, COLUMN_NAME, PARAMETER_MODE, PARAMETER_NAME, PARAMETER_ORDINAL_POSITION, ROUTINE_CATALOG, ROUTINE_SCHEMA, ROUTINE_NAME, SPECIFIC_NAME, TRIGGER_CATALOG, TRIGGER_SCHEMA, or TRIGGER_NAME, then the values retrieved are

   Case:

   i) If the value of SQLSTATE corresponds to *warning — cursor operation conflict*, then the value of CURSOR_NAME is the name of the cursor that caused the completion condition to be raised.

   ii) If the value of SQLSTATE corresponds to *integrity constraint violation*, *transaction rollback — integrity constraint violation*, or *triggered data change violation*, then:

      1) The values of CONSTRAINT_CATALOG and CONSTRAINT_SCHEMA are the <catalog name> and the <unqualified schema name> of the <schema name> of the schema containing the constraint or assertion. The value of CONSTRAINT_NAME is the <qualified identifier> of the constraint or assertion.

      2) Case:

         A) If the violated integrity constraint is a table constraint, then the values of CATALOG_NAME, SCHEMA_NAME, and TABLE_NAME are the <catalog name>, the <unqualified schema name> of the <schema name>, and the <qualified identifier> or <local table name>, respectively, of the table in which the table constraint is contained.

         B) If the violated integrity constraint is an assertion and if only one table referenced by the assertion has been modified as a result of executing the SQL-statement, then the values of CATALOG_NAME, SCHEMA_NAME, and TABLE_NAME are the <catalog name>, the <unqualified schema name> of the <schema name>, and the <qualified identifier> or <local table name>, respectively, of the modified table.

         C) Otherwise, the values of CATALOG_NAME, SCHEMA_NAME, and TABLE_NAME are <space>s.

If the value of TABLE_NAME identifies a declared local temporary table, then the value of CATALOG_NAME is <space>s and the value of SCHEMA_NAME is 'MODULE'.

iii) If the value of SQLSTATE corresponds to *syntax error or access rule violation*, then:

1) The values of CATALOG_NAME, SCHEMA_NAME, and TABLE_NAME are the <catalog name>, the <unqualified schema name> of the <schema name> of the schema that contains the table that caused the syntax error or the access rule violation and the <qualified identifier> or <local table name>, respectively. If TABLE_NAME refers to a declared local temporary table, then CATALOG_NAME is <space>s and SCHEMA_NAME contains 'MODULE'.

2) If the syntax error or the access rule violation was for an inaccessible column, then the value of COLUMN_NAME is the <column name> of that column. Otherwise, the value of COLUMN_NAME is <space>s.

iv) If the value of SQLSTATE corresponds to *invalid cursor state*, then the value of CURSOR_NAME is the name of the cursor that is in the invalid state.

v) If the value of SQLSTATE corresponds to *with check option violation*, then the values of CATALOG_NAME, SCHEMA_NAME, and TABLE_NAME are the <catalog name> and the <unqualified schema name> of the <schema name> of the schema that contains the view that caused the violation of the WITH CHECK OPTION, and the <qualified identifier> of that view, respectively.

vi) If the value of SQLSTATE does not correspond to *syntax error or access rule violation*, then:

1) If the values of CATALOG_NAME, SCHEMA_NAME, TABLE_NAME, and COLUMN_NAME identify a column for which no privileges are granted to the enabled authorization identifiers, then the value of COLUMN_NAME is replaced by a zero-length string.

2) If the values of CATALOG_NAME, SCHEMA_NAME, and TABLE_NAME identify a table for which no privileges are granted to the enabled authorization identifiers, then the values of CATALOG_NAME, SCHEMA_NAME, and TABLE_NAME are replaced by a zero-length string.

3) If the values of CONSTRAINT_CATALOG, CONSTRAINT_SCHEMA, and CONSTRAINT_NAME identify a <table constraint> for some table *T* and if no privileges for *T* are granted to the enabled authorization identifiers, then the values of CONSTRAINT_CATALOG, CONSTRAINT_SCHEMA, and CONSTRAINT_NAME are replaced by a zero-length string.

4) If the values of CONSTRAINT_CATALOG, CONSTRAINT_SCHEMA, and CONSTRAINT_NAME identify an assertion contained in some schema *S* and if the owner of *S* is not included in the set of enabled authorization identifiers, then the values of CONSTRAINT_CATALOG, CONSTRAINT_SCHEMA, and CONSTRAINT_NAME are replaced by a zero-length string.

vii) If the value of SQLSTATE corresponds to *triggered action exception*, to *transaction rollback — triggered action exception*, or to *triggered data change violation* that was caused by a trigger, then:

1) The values of TRIGGER_CATALOG and TRIGGER_SCHEMA are the <catalog name> and the <unqualified schema name>, respectively, of the <schema name> of the schema containing the trigger. The value of TRIGGER_NAME is the <qualified identifier> of the <trigger name> of the trigger.

2) The values of CATALOG_NAME, SCHEMA_NAME, and TABLE_NAME are the <catalog name>, the <unqualified schema name> of the <schema name>, and the <qualified identifier> of the <table name>, respectively, of the table on which the trigger is defined.

viii) If the value of SQLSTATE corresponds to *external routine invocation exception*, or to *external routine exception*, then:

1) The values of ROUTINE_CATALOG and ROUTINE_SCHEMA are the <catalog name> and the <unqualified schema name>, respectively,of the <schema name> of the schema containing the SQL-invoked routine.

2) The values of ROUTINE_NAME and SPECIFIC_NAME are the <identifier> of the <routine name> and the <identifier> of the <specific name> of the SQL-invoked routine, respectively.

3) Case:

A) If the condition is related to some parameter P_i of the SQL-invoked routine, then:

I) The value of PARAMETER_MODE is the <parameter mode> of P_i.

II) The value of PARAMETER_ORDINAL_POSITION is the value of i.

III) The value of PARAMETER_NAME is a zero-length string.

B) Otherwise:

I) The value of PARAMETER_MODE is a zero-length string.

II) The value of PARAMETER_ORDINAL_POSITION is 0 (zero).

III) The value of PARAMETER_NAME is a zero-length string.

ix) If the value of SQLSTATE corresponds to *data exception — numeric value out of range*, *data exception — invalid character value for cast*, *data exception — string data, right truncation*, *data exception — interval field overflow*, *integrity constraint violation*, or *warning — string data, right truncation*, and the condition was raised as the result of an assignment to an SQL parameter during an SQL-invoked routine invocation, then:

1) The values of ROUTINE_CATALOG and ROUTINE_SCHEMA are the <catalog name> and <unqualified schema name>, respectively, of the <schema name> of the schema containing the SQL-invoked routine.

2) The values of ROUTINE_NAME and SPECIFIC_NAME are the <identifier> of the <routine name> and the <identifier> of the <specific name>, respectively, of the SQL-invoked routine.

3) If the condition is related to some parameter P_i of the SQL-invoked routine, then:

A) The value of PARAMETER_MODE is the <parameter mode> of P_i.

B) The value of PARAMETER_ORDINAL_POSITION is the value of i.

C) If an <SQL parameter name> was specified for the SQL parameter when the SQL-invoked routine was created, then the value of PARAMETER_NAME is the <SQL parameter name> of that SQL parameter, P_i; otherwise, the value of PARAMETER_NAME is a zero-length string.

j) If *DI* indicates SERVER_NAME or CONNECTION_NAME, then the values retrieved are

Case:

i) If *R* is Connect, then the name of the SQL-server explicitly or implicitly referenced by *R* and the implementation-defined connection name associated with that SQL-server reference, respectively.

ii) If *R* is Disconnect, then the name of the SQL-server and the associated implementation-defined connection name, respectively, associated with the allocated SQL-connection referenced by *R*.

iii) If the status condition was caused by the application of the General Rules of Subclause 5.3, "Implicit set connection", then the name of the SQL-server and the implementation-defined connection name, respectively, associated with the dormant SQL-connection specified in the application of that Subclause.

iv) If the status condition was raised in an SQL-session, then the name of the SQL-server and the implementation-defined connection name, respectively, associated with the SQL-session in which the status condition was raised.

v) Otherwise, zero-length strings.

k) If *DI* indicates CONDITION_IDENTIFIER, then the value retrieved is

Case:

i) If the value of SQLSTATE corresponds to *unhandled user-defined exception*, then the <condition name> of the user-defined exception.

ii) Otherwise, a zero-length string.

l) If *FI* indicates ROW_NUMBER, then the value retrieved is the number of the row in the rowset to which this status record corresponds. If the status record does not correspond to any particular row, then the value retrieved is 0 (zero).

m) If *FI* indicates COLUMN_NUMBER, then the value retrieved is the number of the column to which this status record corresponds. If the status record does not correspond to any particular column, then the value retrieved is 0 (zero).

n) If *DI* indicates an implementation-defined diagnostics status field, then the value retrieved is the value of the implementation-defined diagnostics status field.

13) Let *V* be the value retrieved.

14) If *DI* indicates a diagnostics field whose row in Table 1, "Header fields in SQL/CLI diagnostics areas" or Table 2, "Status record fields in SQL/CLI diagnostics areas", contains a Data Type that is neither CHARACTER nor CHARACTER VARYING, then DiagInfo is set to *V* and no further rules of this Subclause are applied.

15) Let *BL* be the value of BufferLength.

16) If *BL* is not greater than zero, then an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

17) Let *L* be the length in octets of *V*.

18) If StringLength is not a null pointer, then StringLength is set to *L*.

19) Case:

a) If null termination is *False* for the current SQL-environment, then

Case:

i) If L is not greater than BL, then the first L octets of DiagInfo are set to V and the values of the remaining octets of DiagInfo are implementation-dependent.

ii) Otherwise, DiagInfo is set to the first BL octets of V.

b) Otherwise, let k be the number of octets in a null terminator in the character set of DiagInfo and let the phrase “implementation-defined null character that terminates a C character string” imply k octets, all of whose bits are 0 (zero).

Case:

i) If L is not greater than (BL–k), then the first (L+k) octets of DiagInfo are set to V concatenated with a single implementation-defined null character that terminates a C character string. The values of the remaining characters of DiagInfo are implementation-dependent.

ii) Otherwise, DiagInfo is set to the first (BL–k) octets of V concatenated with a single implementation-defined null character that terminates a C character string.

6.34 GetDiagRec

Function

Get commonly-used information from a CLI diagnostics area.

Definition

```
GetDiagRec (
    HandleType          IN        SMALLINT,
    Handle              IN        INTEGER,
    RecordNumber        IN        SMALLINT,
    Sqlstate            OUT       CHARACTER(5),
    NativeError         OUT       INTEGER,
    MessageText         OUT       CHARACTER(L),
    BufferLength        IN        SMALLINT,
    TextLength          OUT       SMALLINT )
    RETURNS SMALLINT
```

where *L* has a maximum value equal to the implementation-defined maximum length of a variable-length character string.

General Rules

1) Let *HT* be the value of HandleType.

2) If *HT* is not one of the code values in Table 14, "Codes used for SQL/CLI handle types", then an exception condition is raised: *CLI-specific condition — invalid handle*.

3) Case:

   a) If *HT* indicates ENVIRONMENT HANDLE and Handle does not identify an allocated SQL-environment, then an exception condition is raised: *CLI-specific condition — invalid handle*.

   b) If *HT* indicates CONNECTION HANDLE and Handle does not identify an allocated SQL-connection, then an exception condition is raised: *CLI-specific condition — invalid handle*.

   c) If *HT* indicates STATEMENT HANDLE and Handle does not identify an allocated SQL-statement, then an exception condition is raised: *CLI-specific condition — invalid handle*.

   d) If *HT* indicates DESCRIPTOR HANDLE and Handle does not identify an allocated CLI descriptor area, then an exception condition is raised: *CLI-specific condition — invalid handle*.

4) Let *RN* be the value of RecordNumber.

5) Let *R* be the most recently executed CLI routine, other than GetDiagRec, GetDiagField, or Error, for which Handle was passed as the value of an input handle and let *N* be the number of status records generated by the execution of *R*.

   NOTE 42 — The GetDiagRec, GetDiagField, and Error routines may cause exception or completion conditions to be raised, but they do not cause diagnostic information to be generated.

6) If *RN* is less than 1 (one), then an exception condition is raised: *invalid condition number*.

7) If *RN* is greater than *N*, then a completion condition is raised: *no data*, and no further rules of this Subclause are applied.

8) Let *BL* be the value of BufferLength.

9) Information from the *RN*-th status record in the diagnostics area associated with the resource identified by Handle is retrieved.

   a) If Sqlstate is not a null pointer, then Sqlstate is set to the SQLSTATE value corresponding to the status condition.

   b) If NativeError is not a null pointer, then NativeError is set to the implementation-defined native error code corresponding to the status condition.

   c) If MessageText is not a null pointer, then

      Case:

      i) If null termination is <u>*False*</u> for the current SQL-environment and *BL* is zero, then no further rules of this Subclause are applied.

      ii) Otherwise, an implementation-defined character string is retrieved. Let *MT* be the implementation-defined character string that is retrieved and let *L* be the length in octets of *MT*. If *BL* is not greater than zero, then an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*. If TextLength is not a null pointer, then TextLength is set to *L*.

         Case:

         1) If null termination is <u>*False*</u> for the current SQL-environment, then

            Case:

            A) If *L* is not greater than *BL*, then the first *L* octets of MessageText are set to *MT* and the values of the remaining octets of MessageText are implementation-dependent.

            B) Otherwise, MessageText is set to the first *BL* octets of *MT*.

         2) Otherwise, let *k* the number of octets in a null terminator in the character set of MessageText and let the phrase “implementation-defined null character that terminates a C character string” imply *k* octets, all of whose bits are 0 (zero).

            Case:

            A) If *L* is not greater than (*BL*–*k*), then the first (*L*+*k*) octets of MessageText are set to *MT* concatenated with a single implementation-defined null character that terminates a C character string. The values of the remaining characters of MessageText are implementation-dependent.

            B) Otherwise, MessageText is set to the first (*BL*–*k*) octets of *MT* concatenated with a single implementation-defined null character that terminates a C character string.

         NOTE 43 — An implementation may provide <space>s or a zero-length string or a character string that describes the status condition.

6.35 GetEnvAttr

Function

Get the value of an SQL-environment attribute.

Definition

```
GetEnvAttr (
    EnvironmentHandle    IN        INTEGER,
    Attribute            IN        INTEGER,
    Value                OUT       ANY,
    BufferLength         IN        INTEGER,
    StringLength         OUT       INTEGER )
    RETURNS SMALLINT
```

General Rules

1) Case:

   a) If EnvironmentHandle does not identify an allocated SQL-environment or if it identifies an allocated skeleton SQL-environment, then an exception condition is raised: *CLI-specific condition — invalid handle*.

   b) Otherwise:

      i) Let *E* be the allocated SQL-environment identified by EnvironmentHandle.

      ii) The diagnostics area associated with *E* is emptied.

2) Let *A* be the value of Attribute.

3) If *A* is not one of the code values in Table 16, “Codes used for environment attributes”, then an exception condition is raised: *CLI-specific condition — invalid attribute identifier*.

4) If *A* indicates NULL TERMINATION, then

   Case:

   a) If null termination for *E* is *True*, then Value is set to 1 (one).

   b) If null termination for *E* is *False*, then Value is set to 0 (zero).

5) If *A* specifies an implementation-defined environment attribute, then

   Case:

   a) If the data type for the environment attribute is specified in Table 20, “Data types of attributes”, as INTEGER, then Value is set to the value of the implementation-defined environment attribute.

   b) Otherwise:

      i) Let *BL* be the value of BufferLength.

ii) Let *AV* be the value of the implementation-defined environment attribute.

iii) The General Rules of Subclause 5.9, “Character string retrieval”, are applied with Value, *AV*, *BL*, and StringLength as *TARGET*, *VALUE*, *TARGET OCTET LENGTH*, and *RETURNED OCTET LENGTH*, respectively.

6.36 GetFeatureInfo

Function

Get information about features supported by the CLI implementation.

Definition

```
GetFeatureInfo (
    ConnectionHandle     IN        INTEGER,
    FeatureType          IN        CHARACTER(L1),
    FeatureTypeLength    IN        SMALLINT,
    FeatureId            IN        CHARACTER(L2),
    FeatureIdLength      IN        SMALLINT,
    SubFeatureId         IN        CHARACTER(L3),
    SubFeatureIdLength   IN        SMALLINT,
    Supported            OUT       SMALLINT )
    RETURNS SMALLINT
```

where *L1*, *L2*, and *L3* has a maximum value equal to the implementation-defined maximum length of a variable-length character string.

General Rules

1) Case:

   a) If ConnectionHandle does not identify an allocated SQL-connection, then an exception condition is raised: *CLI-specific condition — invalid handle*.

   b) Otherwise:

      i) Let *C* be the allocated SQL-connection identified by ConnectionHandle.

      ii) The diagnostics area associated with *C* is emptied.

2) Case:

   a) If there is no established SQL-connection associated with *C*, then an exception condition is raised: *connection exception — connection does not exist*.

   b) Otherwise, let *EC* be the established SQL-connection associated with *C*.

3) If *EC* is not the current SQL-connection, then the General Rules of Subclause 5.3, “Implicit set connection”, are applied with EC as *dormant SQL-connection*.

4) Let *FTL* be the value of FeatureTypeLength.

5) Case:

   a) If *FTL* is not negative, then let *L* be *FTL*.

   b) If *FTL* indicates NULL TERMINATED, then let *L* be the number of octets of FeatureType that precede the implementation-defined null character that terminates a C character string.

c) Otherwise, an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

6) Case:

a) If *L* is zero, then an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

b) Otherwise, let *FTV* be the first *L* octets of FeatureType and let *FT* be the value of

```
TRIM ( BOTH ' ' FROM 'FTV' )
```

7) If *FT* is other than 'FEATURE', 'SUBFEATURE', or 'PACKAGE', then an exception condition is raised: *CLI-specific condition — invalid attribute value*.

8) Let *FIL* be the value of FeatureIdIdLength.

9) Case:

a) If *FIL* is not negative, then let *L* be *FIL*.

b) If *FIL* indicates NULL TERMINATED, then let *L* be the number of octets of FeatureId that precede the implementation-defined null character that terminates a C character string.

c) Otherwise, an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

10) Case:

a) If *L* is zero, then an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

b) Otherwise, let *FIV* be the first *L* octets of FeatureId and let *FI* be the value of

```
TRIM ( BOTH ' ' FROM 'FIV' )
```

11) Case:

a) If *FT* is 'SUBFEATURE', then:

i) Let *SFIL* be the value of SubFeatureIdLength.

ii) Case:

1) If *SFIL* is not negative, then let *L* be *SFIL*.

2) If *SFIL* indicates NULL TERMINATED, then let *L* be the number of octets of SubFeatureId that precede the implementation-defined null character that terminates a C character string.

3) Otherwise, an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

iii) Case:

1) If *L* is zero, then an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

2) Otherwise, let *SFIV* be the first *L* octets of SubFeatureId and let *SFI* be the value of

```
TRIM ( BOTH ' ' FROM 'SFIV' )
```

b) Otherwise, let *SFI* be a character string consisting of a single space.

12) If there is no row in the INFORMATION_SCHEMA.SQL_FEATURES view with FEATURE_SUBFEATURE_PACKAGE_CODE equal to *FT*, FEATURE_ID equal to *FI*, and SUB_FEATURE_ID equal *SFI*, then an exception condition is raised: *CLI-specific condition — invalid attribute value*.

13) Let *SH* be an allocated statement handle on *C*.

14) Let *STMT* be the character string:

```
SELECT IS_SUPPORTED
FROM INFORMATION_SCHEMA.SQL_FEATURES
WHERE FEATURE_SUBFEATURE_PACKAGE_CODE = 'FT'
  AND FEATURE_ID = 'FI'
  AND SUB_FEATURE_ID = 'SFI'
```

15) Let *IS* be the single column value returned by the implicit invocation of ExecDirect with *SH* as the value of StatementHandle, *STMT* as the value of StatementText, and the length of *STMT* as the value of TextLength.

16) If any status condition, such as connection failure, is caused by the implicit execution of ExecDirect, then:

a) The status records returned by ExecDirect are returned on ConnectionHandle.

b) This invocation of GetFeatureInfo returns the same return code that was returned by the implicit invocation of ExecDirect and no further Rules of this Subclause are applied.

17) If the value of *IS* is 'YES', then Supported is set to 1 (one); otherwise, Supported is set to 0 (zero).

6.37 GetFunctions

Function

Determine whether a CLI routine is supported.

Definition

```
GetFunctions (
    ConnectionHandle        IN      INTEGER,
    FunctionId              IN      SMALLINT,
    Supported               OUT     SMALLINT )
    RETURNS SMALLINT
```

General Rules

1) Case:

   a) If ConnectionHandle does not identify an allocated SQL-connection, then an exception condition is raised: *CLI-specific condition — invalid handle*.

   b) Otherwise:

      i) Let *C* be the allocated SQL-connection identified by ConnectionHandle.

      ii) The diagnostics area associated with *C* is emptied.

2) Case:

   a) If there is no established SQL-connection associated with *C*, then an exception condition is raised: *connection exception — connection does not exist*.

   b) Otherwise, let *EC* be the established SQL-connection associated with *C*.

3) If *EC* is not the current SQL-connection, then the General Rules of Subclause 5.3, "Implicit set connection", are applied with EC as *dormant SQL-connection*.

4) Let *FI* be the value of FunctionId.

5) If *FI* is not one of the codes in Table 28, "Codes used to identify SQL/CLI routines", then an exception condition is raised: *CLI-specific condition — invalid FunctionId specified*.

6) If *FI* identifies a CLI routine that is supported by the implementation, then Supported is set to 1 (one); otherwise, Supported is set to 0 (zero). Table 28, "Codes used to identify SQL/CLI routines", specifies the codes used to identify the CLI routines defined in this part of ISO/IEC 9075.

6.38 GetInfo

This Subclause is modified by Subclause 20.3, "GetInfo", in ISO/IEC 9075-9.

Function

Get information about the implementation.

Definition

```
GetInfo (
    ConnectionHandle        IN      INTEGER,
    InfoType                IN      SMALLINT,
    InfoValue               OUT     ANY,
    BufferLength            IN      SMALLINT,
    StringLength            OUT     SMALLINT )
    RETURNS SMALLINT
```

General Rules

1) Case:

   a) If ConnectionHandle does not identify an allocated SQL-connection, then an exception condition is raised: *CLI-specific condition — invalid handle*.

   b) Otherwise:

      i) Let *C* be the allocated SQL-connection identified by ConnectionHandle.

      ii) The diagnostics area associated with *C* is emptied.

2) Case:

   a) If there is no established SQL-connection associated with *C*, then an exception condition is raised: *connection exception — connection does not exist*.

   b) Otherwise, let *EC* be the established SQL-connection associated with *C*.

3) If *EC* is not the current SQL-connection, then the General Rules of Subclause 5.3, "Implicit set connection", are applied with EC as *dormant SQL-connection*.

4) Several General Rules in this Subclause cause implicit invocation of ExecDirect. If any status condition, such as a connection failure, is caused by such implicit invocation of ExecDirect, then:

   a) The status records returned by ExecDirect are returned on ConnectionHandle.

   b) This invocation of GetInfo returns the same return code that was returned by the implicit invocation of ExecDirect and no further Rules of this Subclause are applied.

5) Let *IT* be the value of InfoType.

6) If *IT* is not one of the codes in Table 29, “Codes and data types for implementation information”, then an exception condition is raised: *CLI-specific condition — invalid information type*.

7) Let *SS* be the SQL-server associated with *EC*.

8) Refer to a component of the SQL-client that is responsible for communicating with one or more SQL-servers as a driver.

9) Let *SH* be an allocated statement handle on *C*.

10) Case:

a) If *IT* indicates any of the following:

— MAXIMUM COLUMN NAME LENGTH

— MAXIMUM COLUMNS IN GROUP BY

— MAXIMUM COLUMNS IN ORDER BY

— MAXIMUM COLUMNS IN SELECT

— MAXIMUM COLUMNS IN TABLE

— MAXIMUM CONCURRENT ACTIVITIES

— MAXIMUM CURSOR NAME LENGTH

— MAXIMUM DRIVER CONNECTIONS

— MAXIMUM IDENTIFIER LENGTH

— MAXIMUM SCHEMA NAME LENGTH

— MAXIMUM STATEMENT OCTETS DATA

— MAXIMUM STATEMENT OCTETS SCHEMA

— MAXIMUM STATEMENT OCTETS

— MAXIMUM TABLE NAME LENGTH

— MAXIMUM TABLES IN SELECT

— MAXIMUM USER NAME LENGTH

— MAXIMUM CATALOG NAME LENGTH

then:

i) Let *STMT* be the character string;

```
SELECT SUPPORTED_VALUE
FROM INFORMATION_SCHEMA.SQL_SIZING
WHERE SIZING_ID = IT
```

ii) Let *V* be the single column value returned by the implicit invocation of ExecDirect with *SH* as the value of StatementHandle, *STMT* as the value of StatementText, and the length of *STMT* as the value of TextLength.

b) If *IT* indicates any of the following:

— CATALOG NAME

— COLLATING SEQUENCE

— CURSOR COMMIT BEHAVIOR

— DATA SOURCE NAME

— DBMS NAME

— DBMS VERSION

— NULL COLLATION

— SEARCH PATTERN ESCAPE

— SERVER NAME

— SPECIAL CHARACTERS

then:

i) Let *STMT* be the character string;

```
SELECT CHARACTER_VALUE
FROM INFORMATION_SCHEMA.SQL_IMPLEMENTATION_INFO
WHERE IMPLEMENTATION_INFO_ID = IT
```

ii) Let *V* be the single column value returned by the implicit invocation of ExecDirect with *SH* as the value of StatementHandle, *STMT* as the value of StatementText, and the length of *STMT* as the value of TextLength.

c) If *IT* indicates any of the following:

— DEFAULT TRANSACTION ISOLATION

— IDENTIFIER CASE

— TRANSACTION CAPABLE

then:

i) Let *STMT* be the character string;

```
SELECT INTEGER_VALUE
FROM INFORMATION_SCHEMA.SQL_IMPLEMENTATION_INFO
WHERE IMPLEMENTATION_INFO_ID = IT
```

ii) Let *V* be the single column value returned by the implicit invocation of ExecDirect with *SH* as the value of StatementHandle, *STMT* as the value of StatementText, and the length of *STMT* as the value of TextLength.

d) If $IT \geq 21000$ and $IT \leq 24999$, or if $IT \geq 11000$ and $IT \leq 14999$, then:

i) Let *STMT* be the character string;

```
SELECT COALESCE (CHARACTER_VALUE, INTEGER_VALUE)
```

```
FROM INFORMATION_SCHEMA.SQL_IMPLEMENTATION_INFO
WHERE IMPLEMENTATION_INFO_ID = IT
```

ii) Let *V* be the single column value returned by the implicit invocation of ExecDirect with *SH* as the value of StatementHandle, *STMT* as the value of StatementText, and the length of *STMT* as the value of TextLength.

e) If *IT* ≥ 25000 and *IT* ≤ 29999, or if *IT* ≥ 15000 and *IT* ≤ 19999, then:

i) Let *STMT* be the character string;

```
SELECT SUPPORTED_VALUE
FROM INFORMATION_SCHEMA.SQL_SIZING
WHERE IMPLEMENTATION_INFO_ID = IT
```

ii) Let *V* be the single column value returned by the implicit invocation of ExecDirect with *SH* as the value of StatementHandle, *STMT* as the value of StatementText, and the length of *STMT* as the value of TextLength.

11) Let *BL* be the value of BufferLength.

12) Case:

a) If the data type of *V* is character string, then the General Rules of Subclause 5.9, “Character string retrieval”, are applied with InfoValue, *V*, *BL*, and StringLength as *TARGET*, *VALUE*, *TARGET LENGTH*, and *RETURNED LENGTH*, respectively.

b) Otherwise, InfoValue is set to *V*.

6.39 GetLength

Function

Retrieve the length of the string value represented by a Large Object locator.

Definition

```
GetLength(
    StatementHandle      IN        INTEGER,
    LocatorType          IN        SMALLINT,
    Locator              IN        INTEGER,
    StringLength         OUT       INTEGER,
    IndicatorValue       OUT       INTEGER )
    RETURNS SMALLINT
```

General Rules

1) Let *S* be the allocated SQL-statement identified by StatementHandle.

2) If there is a prepared statement associated with *S*, then an exception condition is raised: *CLI-specific condition — function sequence error*.

3) If the value of LocatorType is not that of either CHARACTER LARGE OBJECT LOCATOR or BINARY LARGE OBJECT LOCATOR from Table 8, “Codes used for application data types in SQL/CLI”, then an exception condition is raised: *CLI-specific condition — invalid attribute value*.

4) Let *LL* be the Large Object locator value in Locator.

5) If *LL* is not a valid Large Object locator, then an exception condition is raised: *locator exception — invalid specification*.

6) Let *TL* be the actual data type of the Large Object string on the server.

7) If the value of LocatorType is not consistent with *TL* (*e.g.*, a CHARACTER LARGE OBJECT LOCATOR for a BINARY LARGE OBJECT value), then an exception condition is raised: *dynamic SQL error — restricted data type attribute violation*.

8) Let *SV* be the string value that is represented by *LL*.

9) Case:

   a) If *SV* contains the null value, then

      Case:

      Case:

      i) If IndicatorValue is a null pointer, then an exception condition is raised: *data exception — null value, no indicator parameter*.

      ii) Otherwise:

1) IndicatorValue is set to the appropriate 'Code' for SQL NULL DATA in Table 27, “Miscellaneous codes used in CLI”.

2) The value of StringLength is implementation-dependent.

b) Otherwise:

i) IndicatorValue is set to 0 (zero).

ii) If *TL* is CHARACTER LARGE OBJECT, then StringLength is set to the length in characters of *SV*.

iii) If TL is BINARY LARGE OBJECT, then StringLength is set to the length in octets of *SV*.

6.40 GetParamData

Function

Retrieve the value of a dynamic output parameter.

Definition

```
GetParamData (
    StatementHandle      IN        INTEGER,
    ParameterNumber      IN        SMALLINT,
    TargetType           IN        SMALLINT,
    TargetValue          OUT       ANY,
    BufferLength         IN        INTEGER,
    StrLen_or_Ind        OUT       INTEGER )
    RETURNS SMALLINT
```

General Rules

1) Let *S* be the allocated SQL-statement identified by StatementHandle.

2) If there is no executed SQL-statement associated with *S*, then an exception condition is raised: *CLI-specific condition — function sequence error*; otherwise, let *P* be the SQL-statement that was prepared.

3) If *P* is not a <call statement>, then an exception condition is raised: *CLI-specific condition — function sequence error*.

4) Let *APD* be the current application parameter descriptor for *S* and let *N* be the value of the TOP_LEVEL_COUNT field of *APD*.

5) If *N* is less than zero, then an exception condition is raised: *dynamic SQL error — invalid descriptor count*.

6) Let *PN* be the value of ParameterNumber.

7) If *PN* is less than 1 (one) or greater than *N*, then an exception condition is raised: *dynamic SQL error — invalid descriptor index*.

8) If DATA_POINTER is non-zero for at least one of the first *N* item descriptor areas of *APD* for which the TYPE value is neither ROW, ARRAY, nor MULTISET, then let *BPN* be the parameter number associated with such an item descriptor area and let *HBPN* be the value of MAX(*BPN*). Otherwise, let *HBPN* be 0 (zero).

9) Let *IDA* be the item descriptor area of *APD* specified by *PN*. If the value of TYPE of *IDA* is either ROW, ARRAY, or MULTISET, or if LEVEL of *IDA* is greater than 0 (zero), then an exception condition is raised: *dynamic SQL error — invalid descriptor index*.

   NOTE 44 — GetParamData cannot be called to retrieve the data corresponding to a subordinate descriptor record such as, for example, from an individual field of a ROW type.

10) Let *IDA1* be the item descriptor area of *IPD* specified by *PN*.

11) Let *PM* be the value of PARAMETER_MODE in *IDA1*.

12) If *PM* is PARAM MODE IN then an exception condition is raised: *dynamic SQL error — invalid descriptor index*.

13) If *PN* is not greater than *HBPN*, then

    Case:

    a) If the DATA_POINTER field of *IDA* is not zero, then an exception condition is raised: *dynamic SQL error — invalid descriptor index*.

    b) If the DATA_POINTER field of *IDA* is zero, then it is implementation-defined whether an exception condition is raised: *dynamic SQL error — invalid descriptor index*.

       NOTE 45 — This implementation-defined feature determines whether parameters before the highest bound parameter can be accessed by GetParamData.

14) If there is a fetched parameter number associated with *S*, then let *FPN* be that parameter number; otherwise, let *FPN* be zero.

    NOTE 46 — "fetched parameter number" is the ParameterNumber value used with the previous invocation (if any) of the GetParamData routine with *S*. See the General Rules later in this Subclause where this value is set.

15) Case:

    a) If *FPN* is greater than zero and *PN* is not greater than *FPN*, then it is implementation-defined whether an exception condition is raised: *dynamic SQL error — invalid descriptor index*.

       NOTE 47 — This implementation-defined feature determines whether GetParam Data can only access parameters in ascending parameter number order.

    b) If *FPN* is less than zero, then:

       i) Let *AFPN* be the absolute value of *FPN*.

       ii) Case:

          1) If *PN* is less than *AFPN*, then it is implementation-defined whether an exception condition is raised: *dynamic SQL error — invalid descriptor index*.

             NOTE 48 — This implementation-defined feature determines whether GetParamData can only access parameters in ascending parameter number order.

          2) If *PN* is greater than *AFPN*, then let *FPN* be *AFPN*.

16) Let *T* be the value of TargetType.

17) Let *HL* be the programming language of the invoking host program. Let *operative data type correspondence table* be the data type correspondence table for *HL* as specified in Subclause 5.15, "SQL/CLI data type correspondences". Refer to the two columns of the operative data type correspondence table as the *SQL data type column* and the *host data type column*.

18) If either of the following is true, then an exception condition is raised: *CLI-specific condition — invalid data type in application descriptor*.

    a) *T* indicates neither DEFAULT nor APD TYPE and is not one of the code values in Table 8, "Codes used for application data types in SQL/CLI".

    b) *T* is one of the code values in Table 8, "Codes used for application data types in SQL/CLI", but the row that contains the corresponding SQL data type in the SQL data type column of the operative data type correspondence table contains 'None' in the host data type column.

19) If *T* does not indicate APD TYPE, then the data type of the <target specification> described by *IDA* is set to *T*.

20) Let *IPD* be the implementation parameter descriptor associated with *S*.

21) If the value of the TYPE field of *IDA* indicates DEFAULT, then:

   a) Let *PT*, *P*, and *SC* be the values of the TYPE, PRECISION, and SCALE fields, respectively, for the *PN*-th item descriptor area of *IPD* for which LEVEL is 0 (zero).

   b) The data type, precision, and scale of the <target specification> described by *IDA* are set to *PT*, *P*, and *SC*, respectively, for the purposes of this GetParamData invocation only.

22) If *IDA* is not valid as specified in Subclause 5.13, "Description of CLI item descriptor areas", then an exception condition is raised: *dynamic SQL error — using clause does not match target specifications*.

23) Let *TT* be the value of the TYPE field of *IDA*.

24) Case:

   a) If *TT* indicates CHARACTER, then:

      i) Let *UT* be the code value corresponding to CHARACTER VARYING as specified in Table 7, "Codes used for implementation data types in SQL/CLI".

      ii) Let *CL* be the implementation-defined maximum length for a CHARACTER VARYING data type.

   b) Otherwise, let *UT* be *TT* and let *CL* be zero.

25) Case:

   a) If *FPN* is less than zero, then

      Case:

      i) If *TT* does not indicate CHARACTER, CHARACTER LARGE OBJECT, BINARY, BINARY VARYING, or BINARY LARGE OBJECT, then *AFPN* becomes the *fetched parameter number* associated with *S* and an exception condition is raised: *dynamic SQL error — invalid descriptor index*.

      ii) Otherwise, let *FL*, *DV*, and *DL* be the fetched length, data value and data length, respectively, associated with *FPN* and let *TV* be the result of the <string value function>:

```
SUBSTRING (DV FROM (FL+1))
```

   b) Otherwise:

      i) Let *FL* be zero.

      ii) Let *SDT* be the effective data type of the *PCN*-th <select list> column as represented by the values of the TYPE, LENGTH, PRECISION, SCALE, DATETIME_INTERVAL_CODE, DATETIME_INTERVAL_PRECISION, CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, CHARACTER_SET_NAME, USER_DEFINED_TYPE_CATALOG, USER_DEFINED_TYPE_SCHEMA, and USER_DEFINED_TYPE_NAME fields in the *PN*-th item descriptor area of *IPD*. Let *SV* be the value of the parameter, with data type *SDT*.

iii) Let *TDT* be the effective data type of the *PN*-th <target specification> as represented by the type *UT*, the length value *CL*, and the values of the PRECISION, SCALE, CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, CHARACTER_SET_NAME, USER_DEFINED_TYPE_CATALOG, USER_DEFINED_TYPE_SCHEMA, and USER_DEFINED_TYPE_NAME fields of *IDA*.

iv) Case:

1) If *TDT* is a locator type, then

Case:

A) If *SV* is not the null value, then a locator *L* that uniquely identifies *SV* is generated and the value of *TV* of the *i*-th bound target is set to an implementation-dependent four-octet value that represents *L*.

B) Otherwise, the value *TV* of the *PN*-th <target specification> is the null value.

2) If *SDT* and *TDT* are predefined data types, then

Case:

A) If the <cast specification>

```
CAST ( SV AS TDT )
```

does not conform to the Syntax Rules of Subclause 6.12, "<cast specification>", in ISO/IEC 9075-2, and there is an implementation-defined conversion from type *SDT* to type *TDT*, then that implementation-defined conversion is effectively performed, converting *SV* to type *TDT*, and the result is the value *TV* of the *PN*-th <target specification>.

B) Otherwise:

I) If the <cast specification>

```
CAST ( SV AS TDT )
```

does not conform to the Syntax Rules of Subclause 6.12, "<cast specification>", in ISO/IEC 9075-2, then an exception condition is raised: *dynamic SQL error — restricted data type attribute violation*.

II) The <cast specification>

```
CAST ( SV AS TDT )
```

is effectively performed, and is the value *TV* of the *PN*-th <target specification>.

3) If *SDT* is a user-defined type and *TDT* is a predefined data type, then:

A) Let *DT* be the data type identified by *SDT*.

B) If the current SQL-session has a group name corresponding to the user-defined name of *DT*, then let *GN* be that group name; otherwise, let *GN* be the default transform group name associated with the current SQL-session.

C) The Syntax Rules of Subclause 9.19, "Determination of a from-sql function", in ISO/IEC 9075-2, are applied with *DT* and *GN* as *TYPE* and *GROUP*, respectively.

Case:

I) If there is an applicable from-sql function, then let *FSF* be that from-sql function and let *FSFRT* be the <returns data type> of *FSF*.

Case:

1) If *FSFPT* is compatible with *TDT*, then the from-sql function *TSF* is effectively invoked with *SV* as its input parameter and the <return value> is the value *TV* of the *CN*-th <target specification>.

2) Otherwise, an exception condition is raised: *dynamic SQL error — restricted data type attribute violation*.

II) Otherwise, an exception condition is raised: *dynamic SQL error — data type transform function violation*.

26) *PN* becomes the *fetched parameter number* associated with *S*.

27) If *TV* is the null value, then

Case:

a) If StrLen_or_Ind is a null pointer, then an exception condition is raised: *data exception — null value, no indicator parameter*.

b) Otherwise, StrLen_or_Ind is set to the appropriate 'Code' for SQL NULL DATA in Table 27, “Miscellaneous codes used in CLI”, and the value of TargetValue is implementation-dependent.

28) Let *OL* be the value of BufferLength.

29) If null termination is *True* for the current SQL-environment, then let *NB* be the length in octets of a null terminator in the character set of the *i*-th bound target; otherwise let *NB* be 0 (zero).

30) If *TV* is not the null value, then:

a) StrLen_or_Ind is set to 0 (zero).

b) Case:

i) If *TT* does not indicate CHARACTER, CHARACTER LARGE OBJECT, BINARY, BINARY VARYING, or BINARY LARGE OBJECT, then TargetValue is set to *TV*.

ii) Otherwise:

1) If *TT* is CHARACTER or CHARACTER LARGE OBJECT, then:

A) If *TV* is a zero-length character string, then it is implementation-defined whether or not an exception condition is raised: *data exception — zero-length character string*.

B) The General Rules of Subclause 5.9, “Character string retrieval”, are applied with TargetValue, *TV*, *OL*, and StrLen_or_Ind as *TARGET*, *VALUE*, *OCTET LENGTH*, and *RETURNED OCTET LENGTH*, respectively.

2) If *TT* is BINARY, BINARY VARYING, or BINARY LARGE OBJECT, then the General Rules of Subclause 5.10, “Binary string retrieval”, are applied with TargetValue, *TV*, *OL*, and StrLen_or_Ind as *TARGET*, *VALUE*, *OCTET LENGTH*, and *RETURNED OCTET LENGTH*, respectively.

3) If *FCN* is not less than zero, then let *DV* be *TV* and let *DL* be the length of *TV* in octets.

4) Let *FL* be (*FL*+*OL*–*NB*).

5) If *FL* is less than *DL*, then –*PN* becomes the *fetched parameter number* associated with the fetched parameter associated with *S* and *FL*, *DV* and *DL* become the fetched length, data value, and data length, respectively, associated with the fetched parameter number.

6.41 GetPosition

Function

Retrieve the starting position of a string value within another string value, where the second string value is represented by a Large Object locator.

Definition

```
GetPosition(
    StatementHandle      IN        INTEGER,
    LocatorType          IN        SMALLINT,
    SourceLocator        IN        INTEGER,
    SearchLocator        IN        INTEGER,
    SearchLiteral        IN        ANY,
    SearchLiteralLength  IN        INTEGER,
    FromPosition         IN        INTEGER,
    LocatedAt            OUT       INTEGER,
    IndicatorValue       OUT       INTEGER )
    RETURNS SMALLINT
```

General Rules

1) Let *S* be the allocated SQL-statement identified by StatementHandle.

2) If there is a prepared statement associated with *S*, then an exception condition is raised: *CLI-specific condition — function sequence error*.

3) If the value of LocatorType is not that of either CHARACTER LARGE OBJECT LOCATOR or BINARY LARGE OBJECT LOCATOR from Table 8, “Codes used for application data types in SQL/CLI”, then an exception condition is raised: *CLI-specific condition — invalid attribute identifier*.

4) Let *SRCL* be the Large Object locator value in SourceLocator.

5) If *SRCL* is not a valid Large Object locator, then an exception condition is raised: *locator exception — invalid specification*.

6) Let *SRCT* be the actual data type of the Large Object string on the server.

7) If the value of LocatorType is not consistent with *SRCT* (*e.g.*, a CHARACTER LARGE OBJECT LOCATOR for a BINARY LARGE OBJECT value), then an exception condition is raised: *dynamic SQL error — restricted data type attribute violation*.

8) Case:

   a) If *SRCL* represents the null value, then

      Case:

      i) If IndicatorValue is a null pointer, then an exception condition is raised: *data exception — null value, no indicator parameter*.

ii) Otherwise, IndicatorValue is set to the appropriate 'Code' for SQL NULL DATA in Table 27, "Miscellaneous codes used in CLI", the value of all other output arguments is implementation-dependent, and no further rules of this Subclause are applied.

b) Otherwise:

i) IndicatorValue is set to 0 (zero).

ii) Let *SRCV* be the actual value that is represented by *SRCL*.

9) Let *SLL* be the value of SearchLiteralLength.

10) Case:

a) If *SLL* is equal to zero, then:

i) Let *SCHL* be the Large Object locator value in SearchLocator.

ii) If *SCHL* is not a valid Large Object locator, then an exception condition is raised: *locator exception — invalid specification*.

iii) Let *SCHT* be the actual data type of the Large Object string on the server.

iv) If the value of LocatorType is not consistent with *SCHT*, then an exception condition is raised: *dynamic SQL error — restricted data type attribute violation*.

v) If *SCHL* represents the null value, then an exception condition is raised: *CLI-specific condition — invalid attribute value*.

vi) Let *SCHV* be the actual value that is represented by *SCHL*.

b) Otherwise,

Case:

i) If SearchLiteral is a null pointer, then an exception condition is raised: *CLI-specific condition — invalid attribute value*.

ii) Otherwise, let *SCHV* be the value of that literal.

11) Let *FP* be the value of FromPosition. Let *SRCVL* be the length of *SRCV* (in characters if *SRCV* is a character string and in octets if *SRCV* is a binary string).

12) If *FP* is less than 1 (one) or greater than *SRCVL*, then an exception condition is raised: *CLI-specific condition — invalid attribute value*.

13) If *FP* is greater than 1 (one), then let *SRCV* be the value of

```
SUBSTRING (SRCV FROM FP)
```

14) Case:

a) If *SRCV* contains a string *MV* of contiguous characters (if *SRCV* is a character string) or contiguous octets (if *SRCV* is a binary string) that is the same as the string of characters or octets (as appropriate) in *SCHV* then LocatedAt is set to the starting position (in characters or octets, as appropriate) of the first occurrence of *MV* within *SRCV*.

b) Otherwise, LocatedAt is set to 0 (zero).

6.42 GetSessionInfo

Function

Get information about <general value specification>s supported by the implementation.

Definition

```
GetSessionInfo(
    ConnectionHandle     IN        INTEGER,
    InfoType             IN        SMALLINT,
    InfoValue            OUT       ANY,
    BufferLength         IN        SMALLINT,
    StringLength         OUT       SMALLINT )
    RETURNS SMALLINT
```

General Rules

1) Case:

   a) If ConnectionHandle does not identify an allocated SQL-connection, then an exception condition is raised: *CLI-specific condition — invalid handle*.

   b) Otherwise:

      i) Let *C* be the allocated SQL-connection identified by ConnectionHandle.

      ii) The diagnostics area associated with *C* is emptied.

2) Case:

   a) If there is no established SQL-connection associated with *C*, then an exception condition is raised: *connection exception — connection does not exist*.

   b) Otherwise, let *EC* be the established SQL-connection associated with *C*.

3) If *EC* is not the current SQL-connection, then the General Rules of Subclause 5.3, “Implicit set connection”, are applied with EC as *dormant SQL-connection*.

4) Let *IT* be the value of InfoType.

5) If *IT* is not one of the codes in Table 30, “Codes and data types for session implementation information”, then an exception condition is raised: *CLI-specific condition — invalid information type*.

6) Let *GVS* be the value of <general value specification> in the same row as *IT* in Table 30, “Codes and data types for session implementation information”.

7) Let *SH* be an allocated statement handle on *C*.

8) Let *STMT* be the character string:

```
SELECT UNIQUE GVS
```

```
FROM INFORMATION_SCHEMA.TABLES   – Any table would do
WHERE 1 = 1                      – Any predicate that is TRUE would do
```

9) *V* is set to the single column value returned by the implicit invocation of ExecDirect with *SH* as the value of StatementHandle, *STMT* as the value of StatementText, and the length of *STMT* as the value of TextLength.

10) If any status condition, such as connection failure, is caused by the implicit invocation of ExecDirect, then:

   a) The status records returned by ExecDirect on *SH* are returned on ConnectionHandle.

   b) This invocation of GetSessionInfo returns the same return code that was returned by the implicit invocation of ExecDirect and no further Rules of this Subclause are applied.

11) Let *BL* be the value of BufferLength.

12) The General Rules of Subclause 5.9, “Character string retrieval”, are applied with InfoValue, *V*, *BL*, and StringLength as *TARGET*, *VALUE*, *TARGET OCTET LENGTH*, and *RETURNED OCTET LENGTH*, respectively.

6.43 GetStmtAttr

Function

Get the value of an SQL-statement attribute.

Definition

```
GetStmtAttr (
    StatementHandle      IN        INTEGER,
    Attribute            IN        INTEGER,
    Value                OUT       ANY,
    BufferLength         IN        INTEGER,
    StringLength         OUT       INTEGER )
    RETURNS SMALLINT
```

General Rules

1) Let *S* be the allocated SQL-statement identified by StatementHandle.

2) Let *A* be the value of Attribute.

3) If *A* is not one of the code values in Table 18, “Codes used for statement attributes”, then an exception condition is raised: *CLI-specific condition — invalid attribute identifier*.

4) Case:

   a) If *A* indicates APD_HANDLE, then Value is set to the handle of the current application parameter descriptor for *S*.

   b) If *A* indicates ARD_HANDLE, then Value is set to the handle of the current application row descriptor for *S*.

   c) If *A* indicates IPD_HANDLE, then Value is set to the handle of the implementation parameter descriptor associated with *S*.

   d) If *A* indicates IRD_HANDLE, then Value is set to the handle of the implementation row descriptor associated with *S*.

   e) If *A* indicates CURSOR SCROLLABLE, then

      Case:

      i) If the implementation supports scrollable cursors, then

         Case:

         1) If the value of the CURSOR SCROLLABLE attribute of *S* is NONSCROLLABLE, then Value is set to the code value for NONSCROLLABLE from Table 27, “Miscellaneous codes used in CLI”.

2) If the value of the CURSOR SCROLLABLE attribute of *S* is SCROLLABLE, then Value is set to the code value for SCROLLABLE from Table 27, “Miscellaneous codes used in CLI”.

ii) Otherwise, an exception condition is raised: *CLI-specific condition — optional feature not implemented*.

f) If *A* indicates CURSOR SENSITIVITY, then

Case:

i) If the implementation supports cursor sensitivity, then

Case:

1) If the value of the CURSOR SENSITIVITY attribute of *S* is ASENSITIVE, then Value is set to the code value for ASENSITIVE from Table 27, “Miscellaneous codes used in CLI”.

2) If the value of the CURSOR SENSITIVITY attribute of *S* is INSENSITIVE, then Value is set to the code value for INSENSITIVE from Table 27, “Miscellaneous codes used in CLI”.

3) If the value of the CURSOR SENSITIVITY attribute of *S* is SENSITIVE, then Value is set to the code value for SENSITIVE from Table 27, “Miscellaneous codes used in CLI”.

ii) Otherwise, an exception condition is raised: *CLI-specific condition — optional feature not implemented*.

g) If *A* indicates METADATA ID, then

Case:

i) If the METADATA ID attribute for *S* has been set by the SetStmtAttr routine, then Value is set to the code value of that attribute from Table 20, “Data types of attributes”.

ii) Otherwise, Value is set to the code value for FALSE from Table 27, “Miscellaneous codes used in CLI”.

h) If *A* indicates CURSOR HOLDABLE, then

Case:

i) If the implementation supports cursor holdability, then

Case:

1) If the value of the CURSOR HOLDABLE attribute of *S* is NONHOLDABLE, then the Value is set to the code value for NONHOLDABLE from Table 27, “Miscellaneous codes used in CLI”.

2) If the value of the CURSOR HOLDABLE attribute of *S* is HOLDABLE, then the Value is set to the code value for HOLDABLE from Table 27, “Miscellaneous codes used in CLI”.

3) Otherwise, an exception condition is raised: *CLI-specific condition — invalid attribute value*.

ii) Otherwise, an exception condition is raised: *CLI-specific condition — optional feature not implemented*.

i) If *A* indicates CURRENT OF POSITION, then

Case:

i) If there is no fetched rowset associated with *S*, then an exception condition is raised: *CLI-specific condition — invalid cursor state*.

ii) Otherwise, Value is set to the current position within the fetched rowset associated with *S*.

j) If *A* indicates NEST DESCRIPTOR, then

Case:

i) If the NEST DESCRIPTOR attribute for *S* has been set by the SetStmtAttr routine, then Value is set to the code value of that attribute from Table 20, “Data types of attributes”.

ii) Otherwise, VALUE is set to the code value for FALSE from Table 27, “Miscellaneous codes used in CLI”.

5) If *A* specifies an implementation-defined statement attribute, then

Case:

a) If the data type for the statement attribute is specified in Table 20, “Data types of attributes”, as INTEGER, then Value is set to the value of the implementation-defined statement attribute.

b) Otherwise:

i) Let *BL* be the value of BufferLength.

ii) Let *AV* be the value of the implementation-defined statement attribute.

iii) The General Rules of Subclause 5.9, “Character string retrieval”, are applied with Value, *AV*, *BL*, and StringLength as *TARGET*, *VALUE*, *TARGET OCTET LENGTH*, and *RETURNED OCTET LENGTH*, respectively.

6.44 GetSubString

Function

Either retrieve a portion of a string value that is represented by a Large Object locator or create a Large Object value at the server and retrieve a Large Object locator for that value.

Definition

```
GetSubString(
    StatementHandle      IN        INTEGER,
    LocatorType          IN        SMALLINT,
    SourceLocator        IN        INTEGER,
    FromPosition         IN        INTEGER,
    ForLength            IN        INTEGER,
    TargetType           IN        SMALLINT,
    TargetValue          OUT       ANY,
    BufferLength         IN        INTEGER,
    StringLength         OUT       INTEGER,
    IndicatorValue       OUT       INTEGER )
    RETURNS SMALLINT
```

General Rules

1) Let *S* be the allocated SQL-statement identified by StatementHandle.

2) If there is a prepared statement associated with *S*, then an exception condition is raised: *CLI-specific condition — function sequence error*.

3) If the value of LocatorType is not that of either CHARACTER LARGE OBJECT LOCATOR or BINARY LARGE OBJECT LOCATOR from Table 8, "Codes used for application data types in SQL/CLI", then an exception condition is raised: *CLI-specific condition — invalid attribute value*.

4) Let *SRCL* be the Large Object locator value in SourceLocator.

5) If *SRCL* is not a valid Large Object locator, then an exception condition is raised: *locator exception — invalid specification*.

6) Let *SRCT* be the actual data type of the Large Object string on the server.

7) If the value of LocatorType is not consistent with *SRCT* (*e.g.*, a CHARACTER LARGE OBJECT LOCATOR for a BINARY LARGE OBJECT value), then an exception condition is raised: *dynamic SQL error — restricted data type attribute violation*.

8) Let *TT* be the value of TargetType.

9) If *TT* is not equal to one of the values for CHARACTER, CHARACTER LARGE OBJECT, BINARY, BINARY VARYING, BINARY LARGE OBJECT, CHARACTER LARGE OBJECT LOCATOR, or BINARY LARGE OBJECT LOCATOR from Table 8, "Codes used for application data types in SQL/CLI", then an exception condition is raised: *CLI-specific condition — invalid attribute value*.

10) If *SRCL* is the null value, then

Case:

a) If IndicatorValue is a null pointer, then an exception condition is raised: *data exception — null value, no indicator parameter*.

b) Otherwise, IndicatorValue is set to the value of the 'Code' for SQL NULL DATA from Table 27, “Miscellaneous codes used in CLI”, the values of all other output arguments are implementation-dependent, and no further rules of this Subclause are applied.

11) Let *OL* be the value of BufferLength.

12) If *SRCL* is not the null value, then:

a) IndicatorValue is set to 0 (zero).

b) Let *SRCV* be the large object value that is represented by *SRCL*.

c) If *SRCV* is a character string, then let *SRCVL* be the length of *SRCV* in characters; if *SRCV* is a binary string, then let *SRCVL* be the length of *SRCV* in octets.

d) Let *FP* be the value of FromPosition and let *FL* be the value of ForLength.

e) If any of the following is true, then an exception condition is raised: *CLI-specific condition — invalid attribute value*.

i) *FP* is less than 1 (one).

ii) *FL* is less than 1 (one).

iii) *FP*+*FL*–1 is greater than *SRCVL*.

f) Let *RV* be the value of the string that starts at position *FP* and ends at position *FP*+*FL*–1 in *SRCV* (where the positions are in characters or octets, as appropriate).

g) Let *RVL* be the number of octets in *RV*.

h) Case:

i) If *TT* indicates CHARACTER or CHARACTER LARGE OBJECT, then:

1) If *TV* is a zero-length character string, then it is implementation-defined whether or not an exception condition is raised: *data exception — zero-length character string*.

2) The General Rules of Subclause 5.9, “Character string retrieval”, are applied with TargetValue, *RV*, *OL*, and *RVL* as *TARGET*, *VALUE*, *OCTET LENGTH* and *RETURNED OCTET LENGTH*, respectively.

ii) If *TT* indicates BINARY, BINARY VARYING, BINARY LARGE OBJECT, then the General Rules of Subclause 5.10, “Binary string retrieval”, are applied with TargetValue, *RV*, *OL*, and *RVL* as *TARGET*, *VALUE*, *OCTET LENGTH* and *RETURNED OCTET LENGTH*, respectively.

iii) Otherwise, set TargetValue to the value of a Large Object locator that represents the value *RV* at the server.

6.45 GetTypeInfo

Function

Get information about one or all of the predefined data types supported by the implementation.

Definition

```
GetTypeInfo (
    StatementHandle         IN      INTEGER,
    DataType                IN      SMALLINT )
    RETURNS SMALLINT
```

General Rules

1) Let *S* be the allocated SQL-statement identified by StatementHandle.

2) If an open cursor is associated with *S*, then an exception condition is raised: *invalid cursor state*.

3) Let *D* be the value of DataType.

4) If *D* is not the code value corresponding to ALL TYPES in Table 27, "Miscellaneous codes used in CLI", and is not one of the code values in Table 33, "Codes used for concise data types", then an exception condition is raised: *CLI-specific condition — invalid data type*.

5) Let *C* be the allocated SQL-connection with which *S* is associated.

6) Let *EC* be the established SQL-connection associated with *C* and let *SS* be the SQL-server associated with *EC*.

7) Let *TYPE_INFO* be a table, with a definition and description as specified below, that contains a row for each predefined data type supported by *SS*. For all supported predefined data types for which more than one name is supported, it is implementation-defined whether *TYPE_INFO* contains a single row or a row for each supported name.

```
CREATE TABLE TYPE_INFO (
    TYPE_NAME           CHARACTER VARYING(128) NOT NULL
      PRIMARY KEY,
    DATA_TYPE           SMALLINT               NOT NULL,
    COLUMN_SIZE         INTEGER,
    LITERAL_PREFIX      CHARACTER VARYING(128),
    LITERAL_SUFFIX      CHARACTER VARYING(128),
    CREATE_PARAMS       CHARACTER VARYING(128)
      CHARACTER SET SQL_TEXT,
    NULLABLE            SMALLINT               NOT NULL
      CHECK ( NULLABLE IN (0, 1, 2) ),
    CASE_SENSITIVE      SMALLINT               NOT NULL
      CHECK ( CASE_SENSITIVE IN (0, 1) ),
    SEARCHABLE          SMALLINT               NOT NULL
      CHECK ( SEARCHABLE IN (0, 1, 2, 3) ),
    UNSIGNED_ATTRIBUTE  SMALLINT
      CHECK ( UNSIGNED_ATTRIBUTE IN (0, 1)
```

```
            OR UNSIGNED_ATTRIBUTE IS NULL),
    FIXED_PREC_SCALE    SMALLINT NOT NULL
      CHECK ( FIXED_PREC_SCALE IN (0, 1)),
    AUTO_UNIQUE_VALUE   SMALLINT  NOT NULL
      CHECK ( AUTO_UNIQUE_VALUE IN (0, 1)),
    LOCAL_TYPE_NAME     CHARACTER VARYING(128)
      CHARACTER SET SQL_TEXT,
    MINIMUM_SCALE       INTEGER,
    MAXIMUM_SCALE       INTEGER,
    SQL_DATA_TYPE       SMALLINT              NOT NULL,
    SQL_DATETIME_SUB    SMALLINT
      CHECK ( SQL_DATETIME_SUB IN
            ( 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13 )
            OR SQL_DATETIME_SUB IS NULL),
    NUM_PREC_RADIX      INTEGER,
    INTERVAL_PRECISION  SMALLINT )
```

8) The description of the table TYPE_INFO is:

a) The value of TYPE_NAME is the name of the data type. If multiple names are supported for this data type and TYPE_INFO contains only a single row for this data type, then it is implementation-defined which of the names is in TYPE_NAME.

b) The value of DATA_TYPE is the code value for the predefined data type as defined in Table 33, “Codes used for concise data types”.

c) The value of COLUMN_SIZE is:

i) The null value if the data type has neither a length nor a precision.

ii) The maximum length in characters for a character string type.

iii) The maximum or fixed precision, as appropriate, for a numeric data type.

iv) The maximum or fixed length in positions, as appropriate, for a datetime or interval data type.

v) An implementation-defined value for an implementation-defined data type that has a length or a precision.

d) The value of LITERAL_PREFIX is the character string that shall precede the data type value when a <literal> of this data type is specified. The value of LITERAL_PREFIX is the null value if no such string is required.

e) The value of LITERAL_SUFFIX is the character string that shall follow the data type value when a <literal> of this data type is specified. The value of LITERAL_SUFFIX is the null value if no such string is required.

f) The value of CREATE_PARAMS is a comma-separated list of specifiable attributes for the data type in the order in which the attributes may be specified. The attributes <length>, <precision>, <scale>, and <time fractional seconds precision> appear in the list as LENGTH, PRECISION, SCALE, and PRECISION, respectively. The appearance of attributes in implementation-defined data types is implementation-defined.

g) The value of NULLABLE is 1 (one).

h) The value of CASE_SENSITIVE is 1 (one) if the data type is a character string type and the default collation for its implementation-defined implicit character set would result in a case sensitive comparison when two values with this data type are compared. Otherwise, the value of CASE_SENSITIVE is 0 (zero).

i) Refer to the <comparison predicate>, <between predicate>, <in predicate>, <null predicate>, <quantified comparison predicate>, and <match predicate> as the *basic predicates*. If the data type can be the data type of an operand in the <like predicate>, then let *V1* be 1 (one); otherwise let *V1* be 0 (zero). If the data type can be the data type of a column of a <row value constructor predicand> immediately contained in a basic predicate, then let *V2* be 2; otherwise let *V2* be 0 (zero). The value of SEARCHABLE is (*V1*+*V2*).

j) The value of UNSIGNED_ATTRIBUTE is

Case:

i) If the data type is unsigned, then 1 (one).

ii) If the data type is signed, then 0 (zero).

iii) If a sign is not applicable to the data type, then the null value.

k) The value of FIXED_PREC_SCALE is

Case:

i) If the data type is an exact numeric with a fixed precision and scale, then 1 (one).

ii) Otherwise, 0 (zero).

l) The value of AUTO_UNIQUE_VALUE is

Case:

i) If a column of this data type is set to a value unique among all rows of that column when a row is inserted, then 1 (one).

ii) Otherwise, 0 (zero).

m) The value of LOCAL_TYPE_NAME is an implementation-defined localized representation of the name of the data type, intended primarily for display purposes. The value of LOCAL_TYPE_NAME is the null value if a localized representation is not supported.

n) The value of MINIMUM_SCALE is:

i) The null value if the data type has neither a scale nor a fractional seconds precision.

ii) The minimum value of the scale for a data type that has a scale.

iii) The minimum value of the fractional seconds precision for a data type that has a fractional seconds precision.

o) The value of MAXIMUM_SCALE is:

i) The null value if the data type has neither a scale nor a fractional seconds precision.

ii) The maximum value of the scale for a data type that has a scale.

iii) The maximum value of the fractional seconds precision for a data type that has a fractional seconds precision.

p) The value of SQL_DATA_TYPE is the code value for the predefined data type as defined in Table 7, “Codes used for implementation data types in SQL/CLI”.

q) The value of SQL_DATETIME_SUB is

Case:

i) If the data type is a datetime type, then the code value for the datetime type as defined in Table 9, “Codes associated with datetime data types in SQL/CLI”.

ii) If the data type is an interval type, then the code value for the interval type as defined in Table 10, “Codes associated with <interval qualifier> in SQL/CLI”.

iii) Otherwise, the null value.

r) The value of NUM_PREC_RADIX is

Case:

i) If the value of PRECISION is the value of a precision, then the radix of that precision.

ii) Otherwise, the null value.

s) The value of SQL_INTERVAL_PRECISION is

Case:

i) If the data type is an interval type, then <interval leading field precision>.

ii) Otherwise, the null value.

9) Case:

a) If *D* is the code value corresponding to ALL TYPES in Table 27, “Miscellaneous codes used in CLI”, then let *P* be the character string

```
SELECT *
FROM TYPE_INFO
ORDER BY DATA_TYPE
```

b) Otherwise, let *P* be the character string

```
SELECT *
FROM TYPE_INFO
WHERE DATA_TYPE = d
```

10) ExecDirect is implicitly invoked with *S* as the value of StatementHandle, *P* as the value of StatementText, and the length of *P* as the value of TextLength.

6.46 MoreResults

Function

Determine whether there are more result sets available on a statement handle and, if there are, initialize processing for those result sets.

Definition

```
MoreResults (
    StatementHandle      IN       INTEGER )
    RETURNS SMALLINT
```

General Rules

1) Let *S* be the allocated SQL-statement identified by StatementHandle.

2) If there is no executed SQL-statement associated with *S*, then a completion condition is raised: *no data — no additional result sets returned*.

3) Case:

   a) If there is no cursor associated with *S* and there exists an implementation-defined capability to support that situation, then implementation-defined rules are evaluated and no further General Rules of this Subclause are evaluated.

   b) If there is no cursor associated with *S*, then an exception condition is raised: *CLI-specific condition — function sequence error*.

   c) Otherwise, let *CR* be the cursor associated with *S*.

4) If *CR* is currently open, then:

   a) *CR* is placed in the closed state.

   b) Any fetched row associated with *S* is removed from association with *S*.

5) Case:

   a) If there is another result set that was returned for the executed statement associated with *S*, then:

      i) Let *SS* be the <dynamic select statement> or <dynamic single row select statement> that was used to create the result set.

      ii) The General Rules of Subclause 5.5, "Implicit DESCRIBE USING clause", are applied with *SS* and *S* as *SOURCE* and *ALLOCATED STATEMENT*, respectively.

      iii) *CR* is opened on that result set and positioned before the first row.

      iv) A completion condition is raised: *successful completion*.

   b) Otherwise, a completion condition is raised: *no data — no additional result sets returned*.

6.47 NextResult

Function

Determine whether there are more result sets available on a statement handle and, if there are, initialize processing for the next result set on a separate statement handle.

Definition

```
NextResult (
    StatementHandle1     IN        INTEGER,
    StatementHandle2     IN        INTEGER )
    RETURNS SMALLINT
```

General Rules

1) Let *S1* be the allocated SQL-statement identified by StatementHandle1.

2) If there is no executed SQL-statement associated with *S1*, then a completion condition is raised: *no data — no additional result sets returned*.

3) Let *S2* be the allocated SQL-statement identified by StatementHandle2.

4) If there is a prepared statement associated with *S2*, then an exception condition is raised: *CLI-specific condition — function sequence error*.

5) Case:

   a) If there is another result set that was returned for the executed statement associated with *S1*, then:

      i) A cursor *CR* is associated with *S2*.

      ii) Let *SS* be the <dynamic select statement> or <dynamic single row select statement> that was used to create the result set.

      iii) The General Rules of Subclause 5.5, “Implicit DESCRIBE USING clause”, are applied with *SS* and *S2* as *SOURCE* and *ALLOCATED STATEMENT*, respectively.

      iv) *CR* is opened on that result set and positioned before the first row.

      v) A completion condition is raised: *successful completion*.

   b) Otherwise, a completion condition is raised: *no data — no additional result sets returned*.

6.48 NumResultCols

Function

Get the number of result columns.

Definition

```
NumResultCols (
    StatementHandle      IN        INTEGER,
    ColumnCount          OUT       SMALLINT )
    RETURNS SMALLINT
```

General Rules

1) Let *S* be the allocated SQL-statement identified by StatementHandle.

2) Case:

   a) If there is no prepared or executed statement associated with *S*, then an exception condition is raised: *CLI-specific condition — function sequence error*.

   b) Otherwise, let *D* be the implementation row descriptor associated with *S* and let *N* be the value of the TOP_LEVEL_COUNT field of *D*.

3) ColumnCount is set to *N*.

6.49 ParamData

Function

Process a deferred parameter value.

Definition

```
ParamData (
    StatementHandle     IN   INTEGER,
    Value               OUT  ANY )
    RETURNS SMALLINT
```

General Rules

1) Let *S* be the allocated SQL-statement identified by StatementHandle.

2) Case:

   a) If there is no deferred parameter number associated with *S*, then an exception condition is raised: *CLI-specific condition — function sequence error*.

   b) Otherwise, let *DPN* be the deferred parameter number associated with *S*.

3) Let *APD* be the current application parameter descriptor for *S* and let *N* be the value of the TOP_LEVEL_COUNT field of *APD*.

4) For each of the first *N* item descriptor areas *NIDA* in *APD*:

   a) If the OCTET_LENGTH_POINTER field of *NIDA* has the same non-zero value as the INDICATOR_POINTER field of *IDA*, then *SHARE* is true for *NIDA*; otherwise, *SHARE* is false for *NIDA*. Case:

      i) If *SHARE* is true for *NIDA* and the value of the commonly addressed host variable is the appropriate 'Code' for SQL NULL DATA in Table 27, “Miscellaneous codes used in CLI”, then *NULL* is true for *NIDA*.

      ii) If *SHARE* is false for *NIDA*, INDICATOR_POINTER is not zero, and the value of the host variable addressed by INDICATOR_POINTER is the appropriate 'Code' for SQL NULL DATA in Table 27, “Miscellaneous codes used in CLI”, then *NULL* is true for *NIDA*.

      iii) Otherwise, *NULL* is false for *NIDA*.

   b) If *NULL* is false for *NIDA*, OCTET_LENGTH_POINTER is not 0 (zero), and the value of the host variable addressed by OCTET_LENGTH_POINTER is the appropriate 'Code' for SQL NULL DATA in Table 27, “Miscellaneous codes used in CLI”, then *DEFERRED* is true for *NIDA*; otherwise, *DEFERRED* is false for *NIDA*.

5) For each item descriptor area for which *DEFERRED* is true in the first *N* item descriptor areas of *APD* for which LEVEL is 0 (zero), refer to the corresponding <dynamic parameter specification> value as a *deferred parameter value*.

6) Let *IDA* be the *DPN*-th item descriptor area of *APD* and let *PT* and *DP* be the values of the TYPE and DATA_POINTER fields, respectively, of *IDA*.

7) If there is no parameter value associated with *DPN*, then

   Case:

   a) If there is a DATA_POINTER value associated with *DPN*, then an exception condition is raised: *CLI-specific condition — function sequence error*.

   b) Otherwise:

      i) Value is set to *DP*.

      ii) *DP* becomes the DATA_POINTER value associated with *DPN*.

      iii) An exception condition is raised: *CLI-specific condition — dynamic parameter value needed*.

8) Let *IPD* be the implementation parameter descriptor associated with *S*.

9) Let *C* be the allocated SQL-connection with which *S* is associated.

10) Let *V* be the parameter value associated with *DPN*.

11) Case:

   a) If *V* is not the null value, then:

      i) Case:

         1) If *PT* indicates CHARACTER, then:

            A) Let *LO* be the parameter length associated with *DPN* and let *L* be the number of characters of *V* wholly contained in the first *LO* octets of *V*.

            B) If *L* exceeds the implementation-defined maximum length value for the CHARACTER data type, then an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

         2) If *PT* indicates CHARACTER LARGE OBJECT, then:

            A) Let *LO* be the parameter length associated with *DPN* and let *L* be the number of characters of *V* wholly contained in the first *LO* octets of *V*.

            B) If *L* exceeds the implementation-defined maximum length value for the CHARACTER LARGE OBJECT data type, then an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

         3) If *PT* indicates BINARY, then:

            A) Let *LO* be the parameter length associated with *DPN* and let *L* be the minimum of *LO* and the length of *V* in octets.

            B) If *L* exceeds the implementation-defined maximum length value for the BINARY data type, then an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

         4) If *PT* indicates BINARY VARYING, then:

A) Let *LO* be the parameter length associated with *DPN* and let *L* be the minimum of *LO* and the length of *V* in octets.

B) If *L* exceeds the implementation-defined maximum length value for the BINARY VARYING data type, then an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

5) If *PT* indicates BINARY LARGE OBJECT, then:

A) Let *LO* be the parameter length associated with *DPN* and let *L* be the minimum of *LO* and the length of *V* in octets.

B) If *L* exceeds the implementation-defined maximum length value for the BINARY LARGE OBJECT data type, then an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

6) Otherwise, let *L* be zero.

ii) Let *SV* be *V* with effective data type *SDT* as represented by the length value *L* and by the values of the TYPE, PRECISION, and SCALE fields of *IDA*.

b) Otherwise, let *SV* be the null value.

12) Let *TDT* be the effective data type of the *DPN*-th <dynamic parameter specification> as represented by the values of the TYPE, LENGTH, PRECISION, SCALE, DATETIME_INTERVAL_CODE, DATETIME_INTERVAL_PRECISION, CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, CHARACTER_SET_NAME, USER_DEFINED_TYPE_CATALOG, USER_DEFINED_TYPE_SCHEMA, USER_DEFINED_TYPE_NAME, SCOPE_CATALOG, SCOPE_SCHEMA, and SCOPE_NAME fields of the *DPN*-th item descriptor area of *IPD*.

13) Let *SDT* be the effective data type of the *DPN*-th parameter as represented by the values of the TYPE, LENGTH, PRECISION, SCALE, DATETIME_INTERVAL_CODE, DATETIME_INTERVAL_PRECISION, CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, CHARACTER_SET_NAME, USER_DEFINED_TYPE_CATALOG, USER_DEFINED_TYPE_SCHEMA, USER_DEFINED_TYPE_NAME, SCOPE_CATALOG, SCOPE_SCHEMA, and SCOPE_NAME fields in the corresponding item descriptor area of *APD*.

14) Case:

a) If *SDT* is a locator type, then let *TV* be the value *SV*.

b) If *SDT* and *TDT* are predefined types, then

i) Case:

1) If the <cast specification>

```
CAST ( SV AS TDT )
```

does not conform to the Syntax Rules of Subclause 6.12, "<cast specification>", in ISO/IEC 9075-2, and there is an implementation-defined conversion from type *SDT* to type *TDT*, then that implementation-defined conversion is effectively performed, converting *SV* to type *TDT*, and the result is the value *TV* of the *i*-th bound target.

2) Otherwise:

A) If the <cast specification>

```
CAST ( SV AS TDT )
```

does not conform to the Syntax Rules of Subclause 6.12, "<cast specification>", in ISO/IEC 9075-2, then an exception condition is raised: *dynamic SQL error — restricted data type attribute violation*.

B) If the <cast specification>

```
CAST ( SV AS TDT )
```

does not conform to the General Rules of Subclause 6.12, "<cast specification>", in ISO/IEC 9075-2, then an exception condition is raised in accordance with the General Rules of Subclause 6.12, "<cast specification>", in ISO/IEC 9075-2.

C) Let *TV* be the value obtained, with data type *TDT*, by effectively performing the <cast specification>

```
CAST ( SV AS TDT )
```

NOTE 49 — It is implementation-dependent whether the establishment of *TV* occurs at this time or during the preceding invocation of PutData.

ii) Let *UDT* be the effective data type of the actual *DPN*-th <dynamic parameter specification>, defined to be the data type represented by the values of the TYPE, LENGTH, PRECISION, SCALE, DATETIME_INTERVAL_CODE, DATETIME_INTERVAL_PRECISION, CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, CHARACTER_SET_NAME, SCOPE_CATALOG, SCOPE_SCHEMA, and SCOPE_NAME fields that would automatically be set in the *DPN*-th item descriptor area of *IPD* if POPULATE IPD was *True* for *C*.

iii) Case:

1) If the <cast specification>

```
CAST ( TV AS UDT )
```

does not conform to the Syntax Rules of Subclause 6.12, "<cast specification>", in ISO/IEC 9075-2, and there is an implementation-defined conversion from type *SDT* to type *UDT*, then that implementation-defined conversion is effectively performed, converting *SV* to type *UDT*, and the result is the value *TV* of the *i*-th bound target.

2) Otherwise:

A) If the <cast specification>

```
CAST ( TV AS UDT )
```

does not conform to the Syntax Rules of Subclause 6.12, "<cast specification>", in ISO/IEC 9075-2, then an exception condition is raised: *dynamic SQL error — restricted data type attribute violation*.

B) If the <cast specification>

```
CAST ( TV AS UDT )
```

does not conform to the General Rules of Subclause 6.12, “<cast specification>”, in ISO/IEC 9075-2, then an exception condition is raised in accordance with the General Rules of Subclause 6.12, “<cast specification>”, in ISO/IEC 9075-2.

C) The <cast specification>

```
CAST ( TV AS UDT )
```

is effectively performed and is the value of the *DPN*-th dynamic parameter.

15) Let *PN* be the parameter number associated with a deferred parameter value and let *HPN* be the value of MAX(*PN*).

16) If *DPN* is not equal to *HPN*, then:

a) Let *NPN* be the lowest value of *PN* for which $DPN < NPN \leq HPN$.

b) Let *DP* be the value of the DATA_POINTER field of the *NPN*-th item descriptor area of *APD* for which LEVEL is 0 (zero).

c) *NPN* becomes the deferred parameter number associated with *S* and *DP* becomes the DATA_POINTER value associated with the deferred parameter number.

d) An exception condition is raised: *CLI-specific condition — dynamic parameter value needed*.

17) If *DPN* is equal to *HPN*, then:

a) *DPN* is removed from association with *S*.

b) Case:

i) If there is a select source associated with *S*, then:

1) Let *SS* be the select source associated with *S*.

2) If the value of the CURSOR SCROLLABLE attribute of *S* is SCROLLABLE, then let *CT* be 'SCROLL'; otherwise, let *CT* be an empty string.

3) Case:

A) If the value of the CURSOR SENSITIVITY attribute of *S* is INSENSITIVE, then let *CS* be 'INSENSITIVE'.

B) If the value of the CURSOR SENSITIVITY attribute of *S* is SENSITIVE, then let *CS* be 'SENSITIVE'.

C) Otherwise, let *CS* be 'ASENSITIVE'.

4) If the value of the CURSOR HOLDABLE attribute of *S* is HOLDABLE, then let *CH* be 'WITH HOLD'; otherwise, let *CH* be an empty string.

5) Let *CN* be the name of the cursor associated with *S* and let *CR* be the following <declare cursor>:

```
DECLARE CN CS CT CURSOR CH
    FOR SS
```

6) A copy of *SS* is effectively created in which:

A) Each <dynamic parameter specification> is replaced by the value of the corresponding dynamic parameter.

B) Each <value specification> generally contained in *SS* that is CURRENT_USER, CURRENT_ROLE, SESSION_USER, SYSTEM_USER, CURRENT_CATALOG, CURRENT_SCHEMA, CURRENT_DEFAULT_TRANSFORM_GROUP, or CURRENT_TRANFORM_GROUP_FOR_TYPE <path-resolved user-defined type name> is replaced by the value resulting from evaluation of CURRENT_USER, CURRENT_ROLE, SESSION_USER, SYSTEM_USER, CURRENT_CATALOG, CURRENT_SCHEMA, CURRENT_DEFAULT_TRANSFORM_GROUP, or CURRENT_TRANFORM_GROUP_FOR_TYPE <path-resolved user-defined type name>, respectively, with all such evaluations effectively done at the same instant in time.

C) Each <datetime value function> generally contained in *SS* is replaced by the value resulting from evaluation of that <datetime value function>, with all evaluations effectively done at the same instant in time.

D) Each <value specification> generally contained in *S* that is CURRENT_PATH is replaced by the value resulting from evaluation of CURRENT_PATH, with all such evaluations effectively done at the same instant in time.

7) Let *T* be the table specified by the copy of *SS*.

8) A table descriptor for *T* is effectively created.

9) The General Rules of Subclause 14.1, “<declare cursor>”, in ISO/IEC 9075-2, are applied to *CR*.

10) Case:

A) If *CR* specifies INSENSITIVE, then a copy of *T* is effectively created and the cursor identified by *CN* is placed in the open state and its position is before the first row of the copy of *T*.

B) Otherwise, the cursor identified by *CN* is placed in the open state and its position is before the first row of *T*.

11) If *CR* specifies INSENSITIVE, and the implementation is unable to guarantee that significant changes will be invisible through *CR* during the SQL-transaction in which *CR* is opened and every subsequent SQL-transaction during which it may be held open, then an exception condition is raised: *cursor sensitivity exception — request rejected*.

12) If *CR* specifies SENSITIVE, and the implementation is unable to guarantee that significant changes will be visible through *CR* during the SQL-transaction in which *CR* is opened, then an exception condition is raised: *cursor sensitivity exception — request rejected*.

NOTE 50 — The visibility of significant changes through a sensitive holdable cursor during a subsequent SQL-transaction is implementation-defined.

13) Whether an implementation is able to disallow significant changes that would not be visible through a currently open cursor is implementation-defined.

ii) Otherwise:

1) Let *SS* be the statement source associated with *S*.

2) *SS* is removed from association with *S*.

3) Case:

A) If *SS* is a <preparable dynamic delete statement: positioned>, then:

I) Let *CR* be the cursor referenced by *SS*.

II) The General Rules in Subclause 20.22, "<preparable dynamic delete statement: positioned>", in ISO/IEC 9075-2 are applied to *SS*.

III) If the execution of *SS* deleted the current row of *CR*, then the effect on the fetched row, if any, associated with the allocated SQL-statement under which that current row was established, is implementation-defined.

B) If *SS* is a <preparable dynamic update statement: positioned>, then:

I) Let *CR* be the cursor referenced by *SS*.

II) All the General Rules in Subclause 20.24, "<preparable dynamic update statement: positioned>", in ISO/IEC 9075-2 apply to *SS*.

III) If the execution of *SS* updated the current row of *CR*, then the effect on the fetched row, if any, associated with the allocated SQL-statement under which that current row was established, is implementation-defined.

C) Otherwise, the results of the execution are the same as if the statement were contained in an <externally-invoked procedure> and executed; these are described in Subclause 10.4, "<routine invocation>", in ISO/IEC 9075-2.

4) If *SS* is a <call statement>, then the General Rules of Subclause 5.7, "Implicit CALL USING clause", are applied with *SS* and *S* as *SOURCE* and *ALLOCATED STATEMENT*, respectively.

c) Let *R* be the value of the ROW_COUNT field from the diagnostics area associated with *S*.

d) *R* becomes the row count associated with *S*.

e) If *P* executed successfully, then any executed statement associated with *S* is destroyed and *SS* becomes the executed statement associated with *S*.

6.50 Prepare

Function

Prepare a statement.

Definition

```
Prepare (
    StatementHandle      IN        INTEGER,
    StatementText        IN        CHARACTER(L),
    TextLength           IN        INTEGER )
    RETURNS SMALLINT
```

where *L* has a maximum value equal to the implementation-defined maximum length of a variable-length character string.

General Rules

1) Let *S* be the allocated SQL-statement identified by StatementHandle.

2) If an open cursor is associated with *S*, then an exception condition is raised: *invalid cursor state*.

3) Let *TL* be the value of TextLength.

4) Case:

   a) If *TL* is not negative, then let *L* be *TL*.

   b) If *TL* indicates NULL TERMINATED, then let *L* be the number of octets of StatementText that precede the implementation-defined null character that terminates a C character string.

   c) Otherwise, an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

5) Case:

   a) If *L* is zero, then an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

   b) Otherwise, let *P* be the first *L* octets of StatementText.

6) If *P* is a <preparable dynamic delete statement: positioned> or a <preparable dynamic update statement: positioned>, then let *CN* be the cursor name referenced by *P*. Let *C* be the allocated SQL-connection with which *S* is associated. If *CN* is not the name of a cursor associated with another allocated SQL-statement associated with *C*, then an exception condition is raised: *invalid cursor name*.

7) If one or more of the following are true, then an exception condition is raised: *syntax error or access rule violation*.

   a) *P* does not conform to the Format, Syntax Rules or Access Rules for a <preparable statement> or *P* is a <start transaction statement>, a <commit statement>, a <rollback statement>, or a <release savepoint statement>.

NOTE 51 — See Table 32, “SQL-statement codes”, in ISO/IEC 9075-2 for the list of <preparable statement>s. Other parts of ISO/IEC 9075 may have corresponding tables that define additional codes representing statements defined by those parts of ISO/IEC 9075.

b) *P* contains a <simple comment>.

c) *P* contains a <dynamic parameter specification> whose data type is undefined as determined by the rules specified in Subclause 20.6, “<prepare statement>”, in ISO/IEC 9075-2.

8) The data type of any <dynamic parameter specification> contained in *P* is determined by the rules specified in Subclause 20.6, “<prepare statement>”, in ISO/IEC 9075-2.

9) Let *DTGN* be the default transform group name and *TFL* be the list of user-defined type name—transform group name pairs used to identify the group of transform functions for every user-defined type that is referenced in *P*. *DTGN* and *TFL* are not affected by the execution of a <set transform group statement> after *P* is prepared.

10) The following objects associated with *S* are destroyed:

a) Any prepared statement.

b) Any cursor.

c) Any select source.

d) Any executed statement.

If a cursor associated with *S* is destroyed, then so are any prepared statements that reference that cursor.

11) *P* is prepared and the prepared statement is associated with *S*.

12) If *P* is a <dynamic select statement> or a <dynamic single row select statement>, then:

a) *P* becomes the select source associated with *S*.

b) If there is no cursor name associated with *S*, then a unique implementation-dependent name that has the prefix 'SQLCUR' or the prefix 'SQL_CUR' becomes the cursor name associated with *S*.

13) The General Rules of Subclause 5.5, “Implicit DESCRIBE USING clause”, are applied with *SS* and *S* as *SOURCE* and *ALLOCATED STATEMENT*, respectively.

14) The validity of a prepared statement in an SQL-transaction different from the one in which the statement was prepared is implementation-dependent.

6.51 PrimaryKeys

Function

Return a result set that contains a list of the column names that comprise the primary key for a single specified table stored in the information schemas of the connected data source.

Definition

```
PrimaryKeys (
    StatementHandle         IN INTEGER,
    CatalogName             IN CHARACTER(L1),
    NameLength1             IN SMALLINT,
    SchemaName              IN CHARACTER(L2),
    NameLength2             IN SMALLINT,
    TableName               IN CHARACTER(L3),
    NameLength3             IN SMALLINT )
    RETURNS SMALLINT
```

where each of *L1*, *L2*, and *L3* has a maximum value equal to the implementation-defined maximum length of a variable-length character string.

General Rules

1) Let *S* be the allocated SQL-statement identified by StatementHandle.

2) If an open cursor is associated with *S*, then an exception condition is raised: *invalid cursor state*.

3) Let *C* be the allocated SQL-connection with which *S* is associated.

4) Let *EC* be the established SQL-connection associated with *C* and let *SS* be the SQL-server on that connection.

5) Let *PRIMARY_KEYS_QUERY* be a table, with the definition:

```
CREATE TABLE PRIMARY_KEYS_QUERY (
    TABLE_CAT               CHARACTER VARYING(128),
    TABLE_SCHEM             CHARACTER VARYING(128) NOT NULL,
    TABLE_NAME              CHARACTER VARYING(128) NOT NULL,
    COLUMN_NAME             CHARACTER VARYING(128) NOT NULL,
    ORDINAL_POSITION        SMALLINT NOT NULL,
    PK_NAME                 CHARACTER VARYING(128) )
```

6) Let *PKS* represent the set of rows in *SS*'s Information Schema TABLE_CONSTRAINTS view where the value of CONSTRAINT_TYPE is 'PRIMARY KEY'.

7) Let *PK_COLS* represent the set of rows that define the columns within an individual primary key row in *PKS*. These rows are formed by a natural inner join on the values in the CONSTRAINT_CATALOG, CONSTRAINT_SCHEMA, and CONSTRAINT_NAME columns between a row in *PKS* and the matching row or rows in *SS*'s Information Schema KEY_COLUMN_USAGE view.

8) Let *PKS_COLS* represent the set of rows in the combination of all *PK_COLS* sets.

9) *PRIMARY_KEYS_QUERY* contains a row for each row in *PKS_COLS* where:

a) Let *SUP* be the value of Supported that is returned by the execution of GetFeatureInfo with FeatureType = 'FEATURE' and FeatureId = 'C041' (corresponding to the feature “Information Schema metadata constrained by privileges”).

b) Case:

i) If the value of *SUP* is 1 (one), then *PRIMARY_KEYS_QUERY* contains a row for each column of the primary key for a specific table in *SS*'s Information Schema TABLE_CONSTRAINTS view.

ii) Otherwise, *PRIMARY_KEYS_QUERY* contains a row for each column of the primary key for a specific table in *SS*'s Information Schema TABLE_CONSTRAINTS view in accordance with implementation-defined authorization criteria.

10) For each row of *PRIMARY_KEYS_QUERY*:

a) If the implementation does not support catalog names, then TABLE_CAT is set to the null value; otherwise, the value of TABLE_CAT in *PRIMARY_KEYS_QUERY* is the value of the TABLE_CATALOG column in *PKS*.

b) The value of TABLE_SCHEM in *PRIMARY_KEYS_QUERY* is the value of the TABLE_SCHEMA column in *PKS*.

c) The value of TABLE_NAME in *PRIMARY_KEYS_QUERY* is the value of the TABLE_NAME column in *PKS*.

d) The value of COLUMN_NAME in *PRIMARY_KEYS_QUERY* is the value of the COLUMN_NAME column in *PKS_COLS*.

e) The value of ORDINAL_POSITION in *PRIMARY_KEYS_QUERY* is the value of the ORDINAL_POSITION column in *PKS_COLS*.

f) The value of PK_NAME in *PRIMARY_KEYS_QUERY* is the value of the CONSTRAINT_NAME column in *PKS*.

11) Let *NL1*, *NL2*, and *NL3* be the values of NameLength1, NameLength2, and NameLength3, respectively.

12) Let *CATVAL*, *SCHVAL*, and *TBLVAL* be the values of CatalogName, SchemaName, and TableName, respectively.

13) If the METADATA ID attribute of *S* is TRUE, then:

a) If CatalogName is a null pointer and the value of the CATALOG NAME information type from Table 29, “Codes and data types for implementation information”, *Y*, then an exception condition is raised: *CLI-specific condition — invalid use of null pointer*.

b) If SchemaName is a null pointer, then an exception condition is raised: *CLI-specific condition — invalid use of null pointer*.

14) If TableName is a null pointer, then an exception condition is raised: *CLI-specific condition — invalid use of null pointer*.

15) If CatalogName is a null pointer, then *NL1* is set to zero. If SchemaName is a null pointer, then *NL2* is set to zero. If TableName is a null pointer, then *NL3* is set to zero.

16) Case:

a) If *NL1* is not negative, then let *L* be *NL1*.

b) If *NL1* indicates NULL TERMINATED, then let *L* be the number of octets of CatalogName that precede the implementation-defined null character that terminates a C character string.

c) Otherwise, an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

Let *CATVAL* be the first *L* octets of CatalogName.

17) Case:

a) If *NL2* is not negative, then let *L* be *NL2*.

b) If *NL2* indicates NULL TERMINATED, then let *L* be the number of octets of SchemaName that precede the implementation-defined null character that terminates a C character string.

c) Otherwise, an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

Let *SCHVAL* be the first *L* octets of SchemaName.

18) Case:

a) If *NL3* is not negative, then let *L* be *NL3*.

b) If *NL3* indicates NULL TERMINATED, then let *L* be the number of octets of TableName that precede the implementation-defined null character that terminates a C character string.

c) Otherwise, an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

Let *TBLVAL* be the first *L* octets of TableName.

19) Case:

a) If the METADATA ID attribute of *S* is TRUE, then:

i) Case:

1) If the value of *NL1* is zero, then let *CATSTR* be a zero-length string.

2) Otherwise,

Case:

A) If `SUBSTRING(TRIM('CATVAL') FROM 1 FOR 1) = '"'` and if `SUBSTRING(TRIM('CATVAL') FROM CHAR_LENGTH(TRIM('CATVAL')) FOR 1) = '"'`, then let *TEMPSTR* be the value obtained from evaluating:

```
SUBSTRING(TRIM('CATVAL') FROM 2
  FOR CHAR_LENGTH(TRIM('CATVAL')) - 2)
```

and let *CATSTR* be the character string:

```
TABLE_CAT = 'TEMPSTR' AND
```

B) Otherwise, let *CATSTR* be the character string:

```
UPPER(TABLE_CAT) = UPPER('CATVAL') AND
```

ii) Case:

1) If the value of *NL2* is zero, then let *SCHSTR* be a zero-length string.

2) Otherwise,

Case:

A) If `SUBSTRING(TRIM('SCHVAL') FROM 1 FOR 1) = '"'` and if `SUBSTRING(TRIM('SCHVAL') FROM CHAR_LENGTH(TRIM('SCHVAL')) FOR 1) = '"'`, then let *TEMPSTR* be the value obtained from evaluating:

```
SUBSTRING(TRIM('SCHVAL') FROM 2
  FOR CHAR_LENGTH(TRIM('SCHVAL')) - 2)
```

and let *SCHSTR* be the character string:

```
TABLE_SCHEM = 'TEMPSTR' AND
```

B) Otherwise, let *SCHSTR* be the character string:

```
UPPER(TABLE_SCHEM) = UPPER('SCHVAL') AND
```

iii) Case:

1) If the value of *NL3* is zero, then let *TBLSTR* be a zero-length string.

2) Otherwise,

Case:

A) If `SUBSTRING(TRIM('TBLVAL') FROM 1 FOR 1) = '"'` and if `SUBSTRING(TRIM('TBLVAL') FROM CHAR_LENGTH(TRIM('TBLVAL')) FOR 1) = '"'`, then let *TEMPSTR* be the value obtained from evaluating:

```
SUBSTRING(TRIM('TBLVAL') FROM 2
  FOR CHAR_LENGTH(TRIM('TBLVAL')) - 2)
```

and let *TBLSTR* be the character string:

```
TABLE_NAME = 'TEMPSTR' AND
```

B) Otherwise, let *TBLSTR* be the character string:

```
UPPER(TABLE_NAME) = UPPER('TBLVAL') AND
```

b) Otherwise,

i) If the value of *NL1* is zero, then let *CATSTR* be a zero-length string; otherwise, let *CATSTR* be the character string:

```
TABLE_CAT = 'CATVAL' AND
```

ii) If the value of *NL2* is zero, then let *SCHSTR* be a zero-length string; otherwise, let *SCHSTR* be the character string:

```
TABLE_SCHEM = 'SCHVAL' AND
```

iii) If the value of *NL3* is zero, then let *TBLSTR* be a zero-length string; otherwise, let *TBLSTR* be the character string:

```
TABLE_NAME = 'TBLVAL' AND
```

20) Let *PRED* be the result of evaluating:

```
CATSTR || ' ' || SCHSTR || ' ' || TBLSTR || ' ' || 1=1
```

21) Let *STMT* be the character string:

```
SELECT *
FROM PRIMARY_KEYS_QUERY
WHERE PRED
ORDER BY TABLE_CAT, TABLE_SCHEM, TABLE_NAME, ORDINAL_POSITION
```

22) ExecDirect is implicitly invoked with *S* as the value of StatementHandle, *STMT* as the value of StatementText, and the length of *STMT* as the value of TextLength.

6.52 PutData

Function

Provide a deferred parameter value.

Definition

```
PutData (
    StatementHandle     IN   INTEGER,
    Data                IN   ANY,
    StrLen_or_Ind       IN   INTEGER )
    RETURNS SMALLINT
```

General Rules

1) Let *S* be the allocated SQL-statement identified by StatementHandle.

2) Case:

   a) If there is no deferred parameter number associated with *S*, then an exception condition is raised: *CLI-specific condition — function sequence error*.

   b) Otherwise, let *DPN* be the deferred parameter number associated with *S*.

3) If there is no DATA_POINTER value associated with DPN, then an exception condition is raised: *CLI-specific condition — function sequence error*.

4) Let *APD* be the current application parameter descriptor for *S*.

5) Let *PT* be the value of the TYPE field of the *DPN*-th item descriptor area of *APD* for which LEVEL is 0 (zero).

6) Let *IV* be the value of StrLen_or_Ind.

7) If there is a parameter value associated with *DPN* and *PT* does not indicate CHARACTER, CHARACTER LARGE OBJECT, BINARY, BINARY VARYING, or BINARY LARGE OBJECT, then an exception is raised: *CLI-specific condition — non-string data cannot be sent in pieces*.

8) Case:

   a) If *IV* is the appropriate 'Code' for SQL NULL DATA in Table 27, “Miscellaneous codes used in CLI”, then let *V* be the null value.

   b) If *PT* indicates CHARACTER or CHARACTER LARGE OBJECT, then:

      i) Case:

         1) If *IV* is not negative, then let *L* be *IV*.

         2) If *IV* indicates NULL TERMINATED, then let *L* be the number of octets in the characters of Data that precede the implementation-defined null character that terminates a C character string.

3) Otherwise, an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

ii) Let *V* be the first *L* octets of Data.

c) If *PT* indicates BINARY, BINARY VARYING, or BINARY LARGE OBJECT, then:

i) Case:

1) If *IV* is not negative, then let *L* be *IV*.

2) Otherwise, an exception condition is raised: *CLI-specific condition — invalid attribute value*.

ii) Let *V* be the first *L* octets of Data.

d) Otherwise, let *V* be the value of Data.

9) If *V* is not a valid value of the data type indicated by *PT*, then an exception condition is raised: *dynamic SQL error — using clause does not match dynamic parameter specifications*.

10) If there is no parameter value associated with *DPN*, then:

a) *V* becomes the parameter value associated with *DPN*.

b) If *V* is not the null value and *PT* indicates CHARACTER, CHARACTER LARGE OBJECT, BINARY, BINARY VARYING, or BINARY LARGE OBJECT, then *L* becomes the parameter length associated with *DPN*.

11) If there is a parameter value associated with *DPN*, then

Case:

a) If *V* is the null value, then:

i) *DPN* is removed from association with *S*.

ii) Any statement source associated with *S* is removed from association with *S*.

iii) An exception condition is raised: *CLI-specific condition — attempt to concatenate a null value*.

b) Otherwise:

i) Let *PV* be the parameter value associated with *DPN*.

ii) Case:

1) If *PV* is the null value, then:

A) *DPN* is removed from association with *S*.

B) Any statement source associated with *S* is removed from association with *S*.

C) An exception condition is raised: *CLI-specific condition — attempt to concatenate a null value*.

2) Otherwise:

A) Let *PL* be the parameter length associated with *DPN*.

B) Let *NV* be the result of the <string value function>

```
PV || V
```

C) *NV* becomes the parameter value associated with *DPN* and (*PL*+*L*) becomes the parameter length associated with *DPN*.

6.53 RowCount

Function

Get the row count.

Definition

```
RowCount (
    StatementHandle     IN         INTEGER,
    RowCount            OUT        INTEGER )
    RETURNS SMALLINT
```

General Rules

1) Let *S* be the allocated SQL-statement identified by StatementHandle.
2) If there is no executed statement associated with *S*, then an exception condition is raised: *CLI-specific condition — function sequence error*.
3) RowCount is set to the value of the row count associated with *S*.

6.54 SetConnectAttr

Function

Set the value of an SQL-connection attribute.

Definition

```
SetConnectAttr(
    ConnectionHandle    IN   INTEGER,
    Attribute           IN   INTEGER,
    Value               IN   ANY,
    StringLength        IN   INTEGER )
    RETURNS SMALLINT
```

General Rules

1) Case:

   a) If ConnectionHandle does not identify an allocated SQL-connection, then an exception condition is raised: *CLI-specific condition — invalid handle*.

   b) Otherwise:

      i) Let *C* be the allocated SQL-connection identified by ConnectionHandle.

      ii) The diagnostics area associated with *C* is emptied.

2) Let *A* be the value of Attribute.

3) If *A* is not one of the code values in Table 17, “Codes used for connection attributes”, or if *A* is one of the code values in Table 17, “Codes used for connection attributes”, but the row that contains *A* contains 'No' in the 'May be set' column, then an exception condition is raised: *CLI-specific condition — invalid attribute identifier*.

4) If *A* indicates SAVEPOINT NAME, then:

   a) Let *SL* be the value of StringLength.

   b) Case:

      i) If *SL* is not negative, then let *L* be *SL*.

      ii) If *SL* indicates NULL TERMINATED, then let *L* be the number of octets of Value that precede the implementation-defined null character that terminates a C character string.

      iii) Otherwise, an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

   c) The SAVEPOINT NAME attribute of *C* is set to the first *L* octets of Value.

5) If *A* specifies an implementation-defined connection attribute, then

Case:

a) If the data type for the connection attribute is specified as INTEGER in Table 20, “Data types of attributes”, then the connection attribute is set to the value of Value.

b) Otherwise:

   i) Let *SL* be the value of StringLength.

   ii) Case:

      1) If *SL* is not negative, then let *L* be *SL*.

      2) If *SL* indicates NULL TERMINATED, then let *L* be the number of octets of Value that precede the implementation-defined null character that terminates a C character string.

      3) Otherwise, an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

   iii) The connection attribute is set to the first *L* octets of Value.

6.55 SetCursorName

Function

Set a cursor name.

Definition

```
SetCursorName (
    StatementHandle     IN   INTEGER,
    CursorName          IN   CHARACTER(L),
    NameLength          IN   SMALLINT )
    RETURNS SMALLINT
```

where *L* has a maximum value equal to the implementation-defined maximum length of a variable-length character string.

General Rules

1) Let *S* be the allocated SQL-statement identified by StatementHandle.

2) If an open cursor is associated with *S*, then an exception condition is raised: *invalid cursor state*.

3) Let *NL* be the value of NameLength.

4) Case:

   a) If *NL* is not negative, then let *L* be *NL*.

   b) If *NL* indicates NULL TERMINATED, then let *L* be the number of octets of CursorName that precede the implementation-defined null character that terminates a C character string.

   c) Otherwise, an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

5) Case:

   a) If *L* is zero, then an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

   b) Otherwise, let *N* be the number of whole characters in the first *L* octets of CursorName and let *NO* be the number of octets occupied by those *N* characters. If $NO \neq L$, then an exception condition is raised: *invalid cursor name*; otherwise, let *CV* be the first *L* octets of CursorName and let *TCN* be the value of

      ```
      TRIM ( BOTH ' ' FROM 'CV' )
      ```

6) Let *ML* be the maximum length in characters allowed for an <identifier> as specified in the Syntax Rules of Subclause 5.4, “Names and identifiers”, in ISO/IEC 9075-2, and let *TCNL* be the length in characters of *TCN*.

7) Case:

a) If *TCNL* is greater than *ML*, then *CN* is set to the first *ML* characters of *TCN* and a completion condition is raised: *warning — string data, right truncation*.

b) Otherwise, *CN* is set to *TCN*.

8) If *CN* does not conform to the Format and Syntax Rules of an <identifier>, then an exception condition is raised: *invalid cursor name*.

9) Let *C* be the allocated SQL-connection with which *S* is associated and let *SC* be the <search condition>:

```
CN LIKE 'SQL\_CUR%' ESCAPE '\' OR CN LIKE 'SQLCUR%'
```

If *SC* is *True* or if *CN* is identical to the value of any cursor name associated with an allocated SQL-statement associated with *C*, then an exception condition is raised: *invalid cursor name*.

10) *CN* becomes the cursor name associated with *S*.

6.56 SetDescField

Function

Set a field in a CLI descriptor area.

Definition

```
SetDescField (
    DescriptorHandle     IN        INTEGER,
    RecordNumber         IN        SMALLINT,
    FieldIdentifier      IN        SMALLINT,
    Value                IN        ANY,
    BufferLength         IN        INTEGER )
    RETURNS SMALLINT
```

General Rules

1) Let *D* be the allocated CLI descriptor area identified by DescriptorHandle and let *N* be the value of the COUNT field of *D*.

2) The General Rules of Subclause 5.11, “Deferred parameter check”, are applied to *D* as the DESCRIPTOR AREA.

3) Let *FI* be the value of FieldIdentifier.

4) If *FI* is not one of the code values in Table 21, “Codes used for SQL/CLI descriptor fields”, then an exception condition is raised: *CLI-specific condition — invalid descriptor field identifier*.

5) Case:

   a) If the ALLOC_TYPE field of descriptor *D* is USER and *D* is not being used as the current ARD or current APD of any statement handle, then let *DT* be ARD.

   b) Otherwise, let *DT* be the type of the descriptor *D*.

6) Let *MBS* be the value of the May Be Set column in the row of Table 22, “Ability to set SQL/CLI descriptor fields”, that contains *FI* and in the column that contains the descriptor type *DT*.

7) If *MBS* is 'No', then an exception condition is raised: *CLI-specific condition — invalid descriptor field identifier*.

8) Let *RN* be the value of RecordNumber.

9) Let *TYPE* be the value of the Type column in the row of Table 21, “Codes used for SQL/CLI descriptor fields”, that contains *FI*.

10) If *TYPE* is 'ITEM' and *RN* is less than 1 (one), then an exception condition is raised: *dynamic SQL error — invalid descriptor index*.

11) Let *IDA* be the item descriptor area of *D* specified by *RN*.

12) If an exception condition is raised in any of the following General Rules, then all fields of *IDA* for which specific values were provided in the invocation of SetDescField are set to implementation-dependent values and the value of COUNT for *D* is unchanged.

13) Information is set in *D*:

Case:

a) If *FI* indicates COUNT, then

   Case:

   i) If the memory requirements to manage the CLI descriptor area cannot be satisfied, then an exception condition is raised: *CLI-specific condition — memory allocation error*.

   ii) Otherwise, the count of the number of item descriptor areas is set to the value of Value.

b) If *FI* indicates ARRAY_SIZE, then the value of the ARRAY_SIZE header field of descriptor *D* is set to Value.

c) If *FI* indicates ARRAY_STATUS_POINTER, then the value of the ARRAY_STATUS_POINTER header field of descriptor *D* is set to the address of Value. If Value is a null pointer, then the address is set to 0 (zero).

d) If *FI* indicates ROWS_PROCESSED_POINTER, then the value of the ROWS_PROCESSED_POINTER header field of descriptor *D* is set to the address of Value. If Value is a null pointer, then the address is set to 0 (zero).

e) If *FI* indicates OCTET_LENGTH_POINTER, then the value of the OCTET_LENGTH_POINTER field of *IDA* is set to the address of Value.

f) If *FI* indicates DATA_POINTER, then the value of the DATA_POINTER field of *IDA* is set to the address of Value. If Value is a null pointer, then the address is set to 0 (zero).

g) If *FI* indicates INDICATOR_POINTER, then the value of the INDICATOR_POINTER field of *IDA* is set to the address of Value.

h) If *FI* indicates RETURNED_CARDINALITY_POINTER, then the value fo the RETURNED_CARDINALITY_POINTER field of *IDA* is set to the address of Value.

i) If *FI* indicates CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, or CHARACTER_SET_NAME, then:

   i) Let *BL* be the value of BufferLength.

   ii) Case:

      1) If *BL* is not negative, then let *L* be *BL*.

      2) If *BL* indicates NULL TERMINATED, then let *L* be the number of octets of Value that precedes the implementation-defined null character that terminates a C character string.

      3) Otherwise, an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

   iii) Case:

1) If *L* is zero, then an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

2) Otherwise, let *FV* be the first *l* octets of Value and let *TFV* be the value of

```
TRIM ( BOTH ' ' FROM 'FV' )
```

iv) Let *ML* be the maximum length in characters allowed for an <identifier> as specified in the Syntax Rules of Subclause 5.4, "Names and identifiers", in ISO/IEC 9075-2, and let *TFVL* be the length in characters of *TFV*.

v) Case:

1) If *TFVL* is greater than *ML*, then *FV* is set to the first *ML* characters of *TFV* and a completion condition is raised: *warning — string data, right truncation*.

2) Otherwise, *FV* is set to *TFV*.

vi) Case:

1) If *FI* indicates CHARACTER_SET_CATALOG and *FV* does not conform to the Format and Syntax Rules of an <identifier>, then an exception condition is raised: *invalid catalog name*.

2) If *FI* indicates CHARACTER_SET_SCHEMA and *FV* does not conform to the Format and Syntax Rules of an <identifier>, then an exception condition is raised: *invalid schema name*.

3) If *FI* indicates CHARACTER_SET_NAME and *FV* does not conform to the Format and Syntax Rules of an <identifier>, then an exception condition is raised: *invalid character set name*.

vii) The value of the field of *IDA* identified by *FI* is set to the value of *FV*.

j) Otherwise, the value of the field of *IDA* identified by *FI* is set to the value of Value.

14) If *FI* indicates LEVEL, then:

a) If *RI* is 1 (one) and value is not 0 (zero), then an exception condition is raised: *dynamic SQL error — invalid LEVEL value*.

b) If *RI* is greater than 1 (one), then let *PIDA* be *IDA*'s immediately preceding item descriptor area and let *K* be its LEVEL value.

i) If Value is *K*+1 and TYPE in *PIDA* does not indicate ROW, ARRAY, ARRAY LOCATOR, MULTISET, or MULTISET LOCATOR, then an exception condition is raised: *dynamic SQL error — invalid LEVEL value*.

ii) If Value is greater than *K*+1, then an exception condition is raised: *dynamic SQL error — invalid LEVEL value*.

iii) If value is less than *K*+1, then let $OIDA_i$ be the *i*-th item descriptor area to which *PIDA* is subordinate and whose TYPE field indicates ROW. Let NS_i be the number of immediately subordinate descriptor areas of $OIDA_i$ between $OIDA_i$ and *IDA*, and let D_i be the value of DEGREE of $OIDA_i$.

1) For each $OIDA_i$ whose LEVEL value is greater than *V*, if D_i is not equal to NS_i, then an exception condition is raised: *dynamic SQL error — invalid LEVEL value*.

2) If *K* is not 0 (zero), then let $OIDA_i$ be the $OIDA_j$ whose LEVEL value is *K*. If there exists no such $OIDA_j$ or D_j is not greater than NS_j, then an exception condition is raised: *dynamic SQL error — invalid LEVEL value*.

c) The value of LEVEL in *IDA* is set to Value.

15) If *TYPE* is 'ITEM' and *RN* is greater than *N*, then the COUNT field of *D* is set to *RN*.

16) If *FI* indicates TYPE, LENGTH, OCTET_LENGTH, PRECISION, SCALE, DATETIME_INTERVAL_CODE, DATETIME_INTERVAL_PRECISON, PARAMETER_MODE, PARAMETER_ORDINAL_POSITION, PARAMETER_SPECIFIC_CATALOG, PARAMETER_SPECIFIC_SCHEMA, PARAMETER_SPECIFIC_NAME, CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, CHARACTER_SET_NAME, USER_DEFINED_TYPE_CATALOG, USER_DEFINED_TYPE_SCHEMA, USER_DEFINED_TYPE_NAME, SCOPE_CATALOG, SCOPE_SCHEMA, or SCOPE_NAME, then the DATA_POINTER field of *IDA* is set to zero.

17) If *FI* indicates DATA_POINTER, and Value is not a null pointer, and *IDA* is not consistent as specified in Subclause 5.13, “Description of CLI item descriptor areas”, then an exception condition is raised: *CLI-specific condition — inconsistent descriptor information*.

18) Let *V* be the value of Value.

19) If *FI* indicates TYPE, then:

a) All the other fields of *IDA* are set to implementation-dependent values.

b) Case:

i) If *V* indicates CHARACTER, CHARACTER VARYING or CHARACTER LARGE OBJECT then the CHARACTER_SET_CATALOG, CHARACTER_SET_SCHEMA, and CHARACTER_SET_NAME fields of *IDA* are set to the values for the default character set name for the SQL-session and the LENGTH field of *IDA* is set to the maximum possible length in characters of the indicated data type.

ii) If *V* indicates BINARY, BINARY VARYING, or BINARY LARGE OBJECT, then the LENGTH field of *IDA* is set to the maximum possible length in octets of the indicated data type.

iii) If *V* indicates a <datetime type>, then the PRECISION field of *IDA* is set to 0 (zero).

iv) If *V* indicates INTERVAL, then the DATETIME_INTERVAL_PRECISION field of *IDA* is set to 2.

v) If *V* indicates NUMERIC or DECIMAL, then the SCALE field of *IDA* is set to 0 (zero) and the PRECISION field of *IDA* is set to the implementation-defined default value for the precision of the NUMERIC or DECIMAL data types, respectively.

vi) If *V* indicates SMALLINT, INTEGER, or BIGINT, then the SCALE field of *IDA* is set to 0 (zero) and the PRECISION field of *IDA* is set to the implementation-defined value for the precision of the SMALLINT, INTEGER, or BIGINT data types, respectively.

vii) If *V* indicates FLOAT, then the PRECISION field of *IDA* is set to the implementation-defined default value for the precision of the FLOAT data type.

viii) If *V* indicates REAL or DOUBLE PRECISION, then the PRECISION field of *IDA* is set to the implementation-defined value for the precision of the REAL or DOUBLE PRECISION data types, respectively.

ix) If *V* indicates an implementation-defined data type, then an implementation-defined set of fields of *IDA* are set to implementation-defined default values.

x) Otherwise, an exception condition is raised: *CLI-specific condition — invalid data type*.

20) If *FI* indicates DATETIME_INTERVAL_CODE and the TYPE field of *IDA* indicates a <datetime type>, then:

a) All the fields of *IDA* other than DATETIME_INTERVAL_CODE and TYPE are set to implementation-dependent values.

b) Case:

i) If *V* indicates DATE, TIME, or TIME WITH TIME ZONE, then the PRECISION field of *IDA* is set to 0 (zero).

ii) If *V* indicates TIMESTAMP or TIMESTAMP WITH TIME ZONE, then the PRECISION field of *IDA* is set to 6.

21) If *FI* indicates DATETIME_INTERVAL_CODE and the TYPE field of *IDA* indicates INTERVAL, then the DATETIME_INTERVAL_PRECISION field of IDA is set to 2 and

a) If *V* indicates DAY TO SECOND, HOUR TO SECOND, MINUTE TO SECOND, or SECOND, then the PRECISION field of *IDA* is set to 6.

b) Otherwise, the PRECISION field of *IDA* is set to 0 (zero).

22) Restrictions on the differences allowed between implementation and application parameter descriptors are implementation-defined, except as specified in the General Rules of Subclause 5.6, “Implicit EXECUTE USING and OPEN USING clauses”, in the General Rules of Subclause 5.7, “Implicit CALL USING clause”, and in the General Rules of Subclause 6.49, “ParamData”. Restrictions on the differences between the implementation and application row descriptors are implementation-defined, except as specified in the General Rules of Subclause 5.8, “Implicit FETCH USING clause”, and the General Rules of Subclause 6.30, “GetData”.

6.57 SetDescRec

Function

Set commonly-used fields in a CLI descriptor area.

Definition

```
SetDescRec (
    DescriptorHandle     IN        INTEGER,
    RecordNumber         IN        SMALLINT,
    Type                 IN        SMALLINT,
    SubType              IN        SMALLINT,
    Length               IN        INTEGER,
    Precision            IN        SMALLINT,
    Scale                IN        SMALLINT,
    Data                 DEF       ANY,
    StringLength         DEF       INTEGER,
    Indicator            DEF       INTEGER )
    RETURNS SMALLINT
```

General Rules

1) Let *D* be the allocated CLI descriptor area identified by DescriptorHandle and let *N* be the value of the COUNT field of *D*.

2) The General Rules of Subclause 5.11, "Deferred parameter check", are applied to *D* as the DESCRIPTOR AREA.

3) If *D* is an implementation row descriptor, then an exception condition is raised: *CLI-specific condition — cannot modify an implementation row descriptor*.

4) Let *RN* be the value of RecordNumber.

5) If *RN* is less than 1 (one), then an exception condition is raised: *dynamic SQL error — invalid descriptor index*.

6) If *RN* is greater than *N*, then

   Case:

   a) If the memory requirements to manage the larger CLI descriptor area cannot be satisfied, then an exception condition is raised: *CLI-specific condition — memory allocation error*.

   b) Otherwise, the COUNT field of *D* is set to *RN*.

7) Let *IDA* be the item descriptor area of *D* specified by *RN*.

8) Information is set in *D* as follows:

   a) The data type, precision, scale, and datetime data type of the item described by *IDA* are set to the values of Type, Precision, Scale, and SubType, respectively.

b) Case:

i) If *D* is an implementation parameter descriptor, then the length (in characters or positions, as appropriate) of the item described by *IDA* is set to the value of Length.

ii) Otherwise, the length in octets of the item described by *IDA* is set to the value of Length.

c) If StringLength is not a null pointer, then the address of the host variable that is to provide the length of the item described by *IDA*, or that is to receive the returned length in octets of the item described by *IDA*, is set to the address of StringLength.

d) The address of the host variable that is to provide a value for the item described by *IDA*, or that is to receive a value for the item described by *IDA*, is set to the address of Data. If Data is a null pointer, then the address is set to 0 (zero).

e) If Indicator is not a null pointer, then the address of the <indicator variable> associated with the item described by *IDA* is set to the address of Indicator.

9) If Data is not a null pointer and *IDA* is not consistent as specified in Subclause 5.13, "Description of CLI item descriptor areas", then an exception condition is raised: *CLI-specific condition — inconsistent descriptor information*.

10) If an exception condition is raised, then all fields of *IDA* for which specific values were provided in the invocation of SetDescRec are set to implementation-dependent values and the value of the COUNT field of *D* is unchanged.

11) Restrictions on the differences allowed between implementation and application parameter descriptors are implementation-defined, except as specified in the General Rules of Subclause 5.6, "Implicit EXECUTE USING and OPEN USING clauses", in the General Rules of Subclause 5.7, "Implicit CALL USING clause", and in the General Rules of Subclause 6.49, "ParamData". Restrictions on the differences between the implementation and application row descriptors are implementation-defined, except as specified in the General Rules of Subclause 5.8, "Implicit FETCH USING clause", and the General Rules of Subclause 6.30, "GetData".

6.58 SetEnvAttr

Function

Set the value of an SQL-environment attribute.

Definition

```
SetEnvAttr (
    EnvironmentHandle    IN        INTEGER,
    Attribute            IN        INTEGER,
    Value                IN        ANY,
    StringLength         IN        INTEGER )
    RETURNS SMALLINT
```

General Rules

1) Case:

   a) If EnvironmentHandle does not identify an allocated SQL-environment or if it identifies an allocated skeleton SQL-environment, then an exception condition is raised: *CLI-specific condition — invalid handle*.

   b) Otherwise:

      i) Let *E* be the allocated SQL-environment identified by EnvironmentHandle.

      ii) The diagnostics area associated with *E* is emptied.

2) If there are any allocated SQL-connections associated with *E*, then an exception condition is raised: *CLI-specific condition — attribute cannot be set now*.

3) Let *A* be the value of Attribute.

4) If *A* is not one of the code values in Table 16, “Codes used for environment attributes”, then an exception condition is raised: *CLI-specific condition — invalid attribute identifier*.

5) If *A* indicates NULL TERMINATION, then

   Case:

   a) If Value indicates TRUE, then null termination for *E* is set to *True*.

   b) If Value indicates FALSE, then null termination for *E* is set to *False*.

   c) Otherwise, an exception condition is raised: *CLI-specific condition — invalid attribute value*.

6) If *A* specifies an implementation-defined environment attribute, then

   Case:

   a) If the data type for the environment attribute is specified as INTEGER in Table 20, “Data types of attributes”, then the environment attribute is set to the value of Value.

b) Otherwise:

i) Let *SL* be the value of StringLength.

ii) Case:

1) If *SL* is not negative, then let *L* be *SL*.

2) If *SL* indicates NULL TERMINATED, then let *L* be the number of octets of Value that precede the implementation-defined null character that terminates a C character string.

3) Otherwise, an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

iii) The environment attribute is set to the first *L* octets of Value.

6.59 SetStmtAttr

Function

Set the value of an SQL-statement attribute.

Definition

```
SetStmtAttr (
    StatementHandle      IN       INTEGER,
    Attribute            IN       INTEGER,
    Value                IN       ANY,
    StringLength         IN       INTEGER )
    RETURNS SMALLINT
```

General Rules

1) Let *S* be the allocated SQL-statement identified by StatementHandle.

2) Let *A* be the value of Attribute.

3) If *A* is not one of the code values in Table 18, “Codes used for statement attributes”, or if *A* is one of the code values in Table 18, “Codes used for statement attributes”, but the row that contains *A* contains 'No' in the 'May be set' column, then an exception condition is raised: *CLI-specific condition — invalid attribute identifier*.

4) Let *V* be the value of Value.

5) Case:

   a) If *A* indicates APD_HANDLE, then:

      i) Case:

         1) If *V* does not identify an allocated CLI descriptor area, then an exception condition is raised: *CLI-specific condition — invalid attribute value*.

         2) Otherwise, let *DA* be the CLI descriptor area identified by *V* and let *AT* be the value of the ALLOC_TYPE field for *DA*.

      ii) Case:

         1) If *AT* indicates AUTOMATIC but *DA* is not the application parameter descriptor associated with *S*, then an exception condition is raised: *CLI-specific condition — invalid use of automatically-allocated descriptor handle*.

         2) Otherwise, *DA* becomes the current application parameter descriptor for *S*.

   b) If *A* indicates ARD_HANDLE, then:

      i) Case:

1) If *V* does not identify an allocated CLI descriptor area, then an exception condition is raised: *CLI-specific condition — invalid attribute value*.

2) Otherwise, let *DA* be the CLI descriptor area identified by *V* and let *AT* be the value of the ALLOC_TYPE field for *DA*.

ii) Case:

1) If *AT* indicates AUTOMATIC but *DA* is not the application row descriptor associated with *S*, then an exception condition is raised: *CLI-specific condition — invalid use of automatically-allocated descriptor handle*.

2) Otherwise, *DA* becomes the current application row descriptor for *S*.

c) If *A* indicates CURSOR SCROLLABLE, then

Case:

i) If the implementation supports scrollable cursors, then:

1) If an open cursor is associated with *S*, then an exception condition is raised: *CLI-specific condition — attribute cannot be set now*.

2) Case:

A) If *V* indicates NONSCROLLABLE, then the CURSOR SCROLLABLE attribute of *S* is set to NONSCROLLABLE.

B) If *V* indicates SCROLLABLE, then the CURSOR SCROLLABLE attribute of *S* is set to SCROLLABLE.

C) Otherwise, an exception condition is raised: *CLI-specific condition — invalid attribute value*.

ii) Otherwise, an exception condition is raised: *CLI-specific condition — optional feature not implemented*.

d) If *A* indicates CURSOR SENSITIVITY, then

Case:

i) If the implementation supports cursor sensitivity, then

Case:

1) If an open cursor is associated with *S*, then an exception condition is raised: *CLI-specific condition — attribute cannot be set now*.

2) Case:

A) If *V* indicates ASENSITIVE, then the CURSOR SENSITIVITY attribute of *S* is set to ASENSITIVE.

B) If *V* indicates INSENSITIVE, then the CURSOR SENSITIVITY attribute of *S* is set to INSENSITIVE.

C) If *V* indicates SENSITIVE, then the CURSOR SENSITIVITY attribute of *S* is set to SENSITIVE.

D) Otherwise, an exception condition is raised: *CLI-specific condition — invalid attribute value*.

ii) Otherwise, an exception condition is raised: *CLI-specific condition — optional feature not implemented*.

e) If *A* indicates METADATA ID, then

Case:

i) If *V* indicates FALSE, then the METADATA ID attribute of *S* is set to FALSE.

ii) If *V* indicates TRUE, then the METADATA ID attribute of *S* is set to TRUE.

iii) Otherwise, an exception condition is raised: *CLI-specific condition — invalid attribute value*.

f) If *A* indicates CURSOR HOLDABLE, then

Case:

i) If the implementation supports cursor holdability, then

Case:

1) If an open cursor is associated with *S*, then an exception condition is raised: *CLI-specific condition — attribute cannot be set now*.

2) Case:

A) If *V* indicates NONHOLDABLE, then the CURSOR HOLDABLE attribute of *S* is set to NONHOLDABLE.

B) If *V* indicates HOLDABLE, then the CURSOR HOLDABLE attribute of *S* is set to HOLDABLE.

C) Otherwise, an exception condition is raised: *CLI-specific condition — invalid attribute value*.

ii) Otherwise, an exception condition is raised: *CLI-specific condition — optional feature not implemented*.

g) If *A* indicates CURRENT OF POSITION, then

Case:

i) If there is no open cursor associated with *S*, then an exception condition is raised: *CLI-specific condition — Invalid cursor state*.

ii) If *V* is greater than the ARRAY_SIZE field of the application row descriptor associated with *S*, then an exception condition is raised: *CLI-specific condition — row value out of range*.

iii) If the value of the CURSOR SCROLLABLE attribute of *S* is NONSCROLLABLE, then an exception condition is raised: *CLI-specific condition — invalid cursor position*.

iv) Otherwise, the current row within the fetched rowset associated with *S* is set to *V*.

h) If *A* indicates NEST DESCRIPTOR, then

Case:

i) If there is a prepared statement associated with StatementHandle, then an exception condition is raised: *CLI-specific condition — function sequence error*.

ii) Otherwise,

Case:

1) If *V* indicates FALSE, then the NEST DESCRIPTOR attribute of *S* is set to FALSE.

2) If *V* indicates TRUE, then the NEST DESCRIPTOR attribute of *S* is set to TRUE.

3) Otherwise, an exception condition is raised: *CLI-specific condition — invalid attribute value*.

6) If *A* specifies an implementation-defined statement attribute, then

Case:

a) If the data type for the statement attribute is specified as INTEGER in Table 20, "Data types of attributes", then the statement attribute is set to the value of Value.

b) Otherwise:

i) Let *SL* be the value of StringLength.

ii) Case:

1) If *SL* is not negative, then let *L* be *SL*.

2) If *SL* indicates NULL TERMINATED, then let *L* be the number of octets of Value that precede the implementation-defined null character that terminates a C character string.

3) Otherwise, an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

iii) The statement attribute is set to the first *L* octets of Value.

6.60 SpecialColumns

Function

Return a result set that contains a list of columns the combined values of which can uniquely identify any row within a single specified table described by the Information Schemas of the connected data source.

Definition

```
SpecialColumns (
    StatementHandle         IN INTEGER,
    IdentifierType          IN SMALLINT,
    CatalogName             IN CHARACTER(L1),
    NameLength1             IN SMALLINT,
    SchemaName              IN CHARACTER(L2),
    NameLength2             IN SMALLINT,
    TableName               IN CHARACTER(L3),
    NameLength3             IN SMALLINT,
    Scope                   IN SMALLINT,
    Nullable                IN SMALLINT )
    RETURNS SMALLINT
```

where each of *L1*, *L2*, and *L3* has a maximum value equal to the implementation-defined maximum length of a variable-length character string.

General Rules

1) Let *S* be the allocated SQL-statement identified by StatementHandle.

2) If an open cursor is associated with *S*, then an exception condition is raised: *invalid cursor state*.

3) Let *C* be the allocated SQL-connection with which *S* is associated.

4) Let *EC* be the established SQL-connection associated with *C* and let *SS* be the SQL-server on that connection.

5) Let *SPECIAL_COLUMNS_QUERY* be a table, with the definition:

```
CREATE TABLE SPECIAL_COLUMNS_QUERY (
    SCOPE                   SMALLINT,
    COLUMN_NAME             CHARACTER VARYING(128) NOT NULL,
    DATA_TYPE               SMALLINT NOT NULL,
    TYPE_NAME               CHARACTER VARYING(128) NOT NULL,
    COLUMN_SIZE             INTEGER,
    BUFFER_LENGTH           INTEGER,
    DECIMAL_DIGITS          SMALLINT,
    PSEUDO_COLUMN           SMALLINT )
```

6) *SPECIAL_COLUMNS_QUERY* contains a row for each column that is part of a set of columns that can be used to best uniquely identify a row within the tables listed in *SS*'s Information Schema TABLES view. Some tables may not have such a set of columns. Some tables may have more than one such set, in which case it is implementation-dependent as to which set of columns is chosen. It is implementation-dependent as to whether a column identified for a given table is a pseudo-column.

a) Let *SUP* be the value of Supported that is returned by the execution of GetFeatureInfo with FeatureType = 'FEATURE' and FeatureId = 'C041' (corresponding to the feature “Information Schema metadata constrained by privileges”).

b) Case:

i) If the value of *SUP* is 1 (one), then Table 29, “Codes and data types for implementation information”, is 'Y', then *SPECIAL_COLUMNS_QUERY* contains a row for each identifying column in *SS*'s Information Schema COLUMNS view and each implementation-dependent pseudo-column.

ii) Otherwise, *SPECIAL_COLUMNS_QUERY* contains a row for each identifying column in *SS*'s Information Schema COLUMNS view and each implementation-dependent pseudo-column in accordance with implementation-defined authorization criteria.

7) If the value of IdentifierType is other than the code for BEST ROWID in Table 39, “Column types and scopes used with SpecialColumns”, or an implementation-defined extension to that table, then an exception condition is raised: *CLI-specific condition — column type out of range*.

8) If the value of Scope is other than the code SCOPE CURRENT ROW, SCOPE TRANSACTION, or SCOPE SESSION in Table 39, “Column types and scopes used with SpecialColumns”, or an implementation-defined extension to that table, then an exception condition is raised: *CLI-specific condition — scope out of range*.

9) If the value of Nullable is other than the code for NO NULLS or NULLABLE in Table 39, “Column types and scopes used with SpecialColumns”, then an exception condition is raised: *CLI-specific condition — nullable type out of range*.

10) For each row of *SPECIAL_COLUMNS_QUERY*:

a) The value of SCOPE in *SPECIAL_COLUMNS_QUERY* is either the code for one of SCOPE CURRENT ROW, SCOPE TRANSACTION, or SCOPE SESSION from Table 39, “Column types and scopes used with SpecialColumns”, or it is an implementation-defined value, determined as follows:

Case:

i) If the value that uniquely identifies a row is only guaranteed to be valid while positioned on that row, then the code is that for SCOPE CURRENT ROW.

ii) If the value that uniquely identifies a row is only guaranteed to be valid for the current transaction, then the code is that for SCOPE TRANSACTION.

iii) If the value that uniquely identifies a row is only guaranteed to be valid for the current SQL-session, then the code is that for SCOPE SESSION.

iv) Otherwise, the value is implementation-defined.

b) The value of COLUMN_NAME in *SPECIAL_COLUMNS_QUERY* is the value of the COLUMN_NAME column in the COLUMNS view.

c) The value of DATA_TYPE in *SPECIAL_COLUMNS_QUERY* is derived from the values of the DATA_TYPE and INTERVAL_TYPE columns in the COLUMNS view as follows:

Case:

i) If the value of DATA_TYPE in the COLUMNS view is 'INTERVAL', then the value of DATA_TYPE in (SPECIAL_COLUMNS_QUERY) is the appropriate Code from Table 33, “Codes used for concise data types”, that matches the interval specified in the INTERVAL_TYPE column in the COLUMNS view.

ii) Otherwise, the value of DATA_TYPE in *SPECIAL_COLUMNS_QUERY* is the appropriate Code from Table 33, “Codes used for concise data types”, that matches the interval specified in the DATA_TYPE column in the COLUMNS view.

d) The value of TYPE_NAME in *SPECIAL_COLUMNS_QUERY* is an implementation-defined value that is the character string by which the data type is known at the data source.

e) The value of COLUMN_SIZE in *SPECIAL_COLUMNS_QUERY* is

Case:

i) If the value of DATA_TYPE in the COLUMNS view is 'CHARACTER', 'CHARACTER VARYING', 'CHARACTER LARGE OBJECT', 'BINARY', 'BINARY VARYING', or 'BINARY LARGE OBJECT', then the value is that of the CHARACTER_MAXIMUM_LENGTH in the same row of the COLUMNS view.

ii) If the value of DATA_TYPE in the COLUMNS view is 'DECIMAL' or 'NUMERIC', then the value is that of the NUMERIC_PRECISION column in the same row of the COLUMNS view.

iii) If the value of DATA_TYPE in the COLUMNS view is 'SMALLINT', 'INTEGER', 'BIGINT', 'FLOAT', 'REAL', or 'DOUBLE PRECISION', then the value is implementation-defined.

iv) If the value of DATA_TYPE in the COLUMNS view is 'DATE', 'TIME', 'TIMESTAMP', 'TIME WITH TIME ZONE', or 'TIMESTAMP WITH TIME ZONE', then the value of COLUMN_SIZE is that derived from SR 33), in Subclause 6.1, “<data type>”, of ISO/IEC 9075-2, where the value of <time fractional seconds precision> is the value of the NUMERIC_PRECISION column in the same row of the COLUMNS view.

v) If the value of DATA_TYPE in the COLUMNS view is 'INTERVAL', then the value of COLUMN_SIZE is that derived from the General Rules of Subclause 10.1, “<interval qualifier>”, of ISO/IEC 9075-2, where:

1) The value of <interval qualifier> is the value of the INTERVAL_TYPE column in the same row of the COLUMNS view.

2) The value of <interval leading field precision> is the value of the INTERVAL_PRECISION column in the same row of the COLUMNS view.

3) The value of <interval fractional seconds precision> is the value of the NUMERIC_PRECISION column in the same row of the COLUMNS view.

vi) If the value of DATA_TYPE in the COLUMNS view is 'REF', then the value is the length in octets of the reference type.

vii) Otherwise, the value is implementation-dependent.

f) The value of BUFFER_LENGTH in *SPECIAL_COLUMNS_QUERY* is implementation-defined.

NOTE 52 — The purpose of BUFFER_LENGTH is to record the number of octets transferred for the column with a Fetch routine, a FetchScroll routine, or a GetData routine when the TYPE field in the application row descriptor indicates DEFAULT. This length excludes any null terminator.

g) The value of DECIMAL_DIGITS in *SPECIAL_COLUMNS_QUERY* is:

Case:

i) If the value of DATA_TYPE in the COLUMNS view is one of 'DATE', 'TIME', 'TIMESTAMP', 'TIME WITH TIME ZONE', or 'TIMESTAMP WITH TIME ZONE', then the value of DECIMAL_DIGITS in *SPECIAL_COLUMNS_QUERY* is the value of the DATETIME_PRECISION column in the COLUMNS view.

ii) If the value of DATA_TYPE in the COLUMNS view is one of 'NUMERIC', 'DECIMAL', 'SMALLINT', 'INTEGER', or 'BIGINT', then the value of DECIMAL_DIGITS in *SPECIAL_COLUMNS_QUERY* is the value of the NUMERIC_SCALE column in the COLUMNS view.

iii) Otherwise, the value of DECIMAL_DIGITS in *SPECIAL_COLUMNS_QUERY* is the null value.

h) The value of PSEUDO_COLUMN in *SPECIAL_COLUMNS_QUERY* is the code for one of PSEUDO UNKNOWN, NOT PSEUDO, or PSEUDO from Table 39, “Column types and scopes used with SpecialColumns”. The algorithm used to set this value is implementation-dependent.

11) Let *NL1*, *NL2*, and *NL3* be the values of NameLength1, NameLength2, and NameLength3, respectively.

12) Let *CATVAL*, *SCHVAL*, *TBLVAL*, *SCPVAL*, and *NULVAL* be the values of CatalogName, SchemaName, and TableName, Scope, and Nullable respectively.

13) If the METADATA ID attribute of *S* is TRUE, then:

a) If CatalogName is a null pointer and the value of the CATALOG NAME information type from Table 29, “Codes and data types for implementation information”, is 'Y', then an exception condition is raised: *CLI-specific condition — invalid use of null pointer*.

b) If SchemaName is a null pointer, then an exception condition is raised: *CLI-specific condition — invalid use of null pointer*.

14) If TableName is a null pointer, then an exception condition is raised: *CLI-specific condition — invalid use of null pointer*.

15) If CatalogName is a null pointer, then *NL1* is set to zero. If SchemaName is a null pointer, then *NL2* is set to zero. If TableName is a null pointer, then *NL3* is set to zero.

16) Case:

a) If *NL1* is not negative, then let *L* be *NL1*.

b) If *NL1* indicates NULL TERMINATED, then let *L* be the number of octets of CatalogName that precede the implementation-defined null character that terminates a C character string.

c) Otherwise, an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

Let *CATVAL* be the first *L* octets of CatalogName.

17) Case:

a) If *NL2* is not negative, then let *L* be *NL2*.

b) If *NL2* indicates NULL TERMINATED, then let *L* be the number of octets of SchemaName that precede the implementation-defined null character that terminates a C character string.

c) Otherwise, an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

Let *SCHVAL* be the first *L* octets of SchemaName.

18) Case:

a) If *NL3* is not negative, then let *L* be *NL3*.

b) If *NL3* indicates NULL TERMINATED, then let *L* be the number of octets of TableName that precede the implementation-defined null character that terminates a C character string.

c) Otherwise, an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

Let *TBLVAL* be the first *L* octets of TableName.

19) Case:

a) If the METADATA ID attribute of *S* is TRUE, then:

i) Case:

1) If the value of *NL1* is zero, then let *CATSTR* be a zero-length string.

2) Otherwise,

Case:

A) If `SUBSTRING(TRIM('CATVAL') FROM 1 FOR 1) = '"'` and if `SUBSTRING(TRIM('CATVAL') FROM CHAR_LENGTH(TRIM('CATVAL')) FOR 1) = '"'`, then let *TEMPSTR* be the value obtained from evaluating:

```
SUBSTRING(TRIM('CATVAL') FROM 2
  FOR CHAR_LENGTH(TRIM('CATVAL')) - 2)
```

and let *CATSTR* be the character string:

```
TABLE_CAT = 'TEMPSTR' AND
```

B) Otherwise, let *CATSTR* be the character string:

```
UPPER(TABLE_CAT) = UPPER('CATVAL') AND
```

ii) Case:

1) If the value of *NL2* is zero, then let *SCHSTR* be a zero-length string.

2) Otherwise,

Case:

A) If `SUBSTRING(TRIM('SCHVAL') FROM 1 FOR 1) = '"'` and if `SUBSTRING(TRIM('SCHVAL') FROM CHAR_LENGTH(TRIM('SCHVAL')) FOR 1) = '"'`, then let *TEMPSTR* be the value obtained from evaluating:

```
SUBSTRING(TRIM('SCHVAL') FROM 2
  FOR CHAR_LENGTH(TRIM('SCHVAL')) - 2)
```

and let *SCHSTR* be the character string:

```
TABLE_SCHEM = 'TEMPSTR' AND
```

B) Otherwise, let *SCHSTR* be the character string:

```
UPPER(TABLE_SCHEM) = UPPER('SCHVAL') AND
```

iii) Case:

1) If the value of *NL3* is zero, then let *TBLSTR* be a zero-length string.

2) Otherwise,

Case:

A) If `SUBSTRING(TRIM('TBLVAL') FROM 1 FOR 1) = '"'` and if `SUBSTRING(TRIM('TBLVAL') FROM CHAR_LENGTH(TRIM('TBLVAL')) FOR 1) = '"'`, then let *TEMPSTR* be the value obtained from evaluating:

```
SUBSTRING(TRIM('TBLVAL') FROM 2
  FOR CHAR_LENGTH(TRIM('TBLVAL')) - 2)
```

and let *TBLSTR* be the character string:

```
TABLE_NAME = 'TEMPSTR' AND
```

B) Otherwise, let *TBLSTR* be the character string:

```
UPPER(TABLE_NAME) = UPPER('TBLVAL') AND
```

b) Otherwise:

i) If the value of *NL1* is zero, then let *CATSTR* be a zero-length string; otherwise, let *CATSTR* be the character string:

```
TABLE_CAT = 'CATVAL' AND
```

ii) If the value of *NL2* is zero, then let *SCHSTR* be a zero-length string; otherwise, let SCHSTR be the character string:

```
TABLE_SCHEM = 'SCHVAL' AND
```

iii) If the value of *NL3* is zero, then let *TBLSTR* be a zero-length string; otherwise, let *TBLSTR* be the character string:

```
TABLE_NAME = 'TBLVAL' AND
```

20) Let the value of *SCPSTR* be the character string:

```
SCOPE >= SCPVAL
```

21) Let *PRED* be the result of evaluating:

```
CATSTR || ' ' || SCHSTR || ' ' || TBLSTR || ' ' || SCPSTR
```

22) Case:

a) If NULVAL is equal to the code for NO NULLS in Table 27, “Miscellaneous codes used in CLI”, and any of the rows selected by the above query would describe a column for which the value of IS_NULLABLE column in the COLUMNS view is 'YES', then let *STMT* be the character string:

```
SELECT *
FROM SPECIAL_COLUMNS_QUERY
WHERE 1 = 2  – select no rows
ORDER BY SCOPE
```

b) Otherwise, let *STMT* be the character string:

```
SELECT *
FROM SPECIAL_COLUMNS_QUERY
WHERE PRED
ORDER BY SCOPE
```

23) ExecDirect is implicitly invoked with *S* as the value of StatementHandle, *STMT* as the value of Statement-Text, and the length of *STMT* as the value of TextLength.

6.61 StartTran

Function

Explicitly start an SQL-transaction and set its characteristics.

Definition

```
StartTran (
    HandleType              IN SMALLINT,
    Handle                  IN INTEGER,
    AccessMode              IN INTEGER,
    IsolationLevel          IN INTEGER )
    RETURNS SMALLINT
```

General Rules

1) Let *HT* be the value of HandleType and let *H* be the value of Handle.

2) If *HT* is not one of the code values in Table 14, "Codes used for SQL/CLI handle types", then an exception condition is raised: *CLI-specific condition — invalid handle*.

3) Case:

   a) If *HT* indicates STATEMENT HANDLE, then

      Case:

      i) If *H* does not identify an allocated SQL-statement, then an exception condition is raised: *CLI-specific condition — invalid handle*.

      ii) Otherwise, an exception condition is raised: *CLI-specific condition — invalid attribute identifier*.

   b) If *HT* indicates DESCRIPTOR HANDLE, then

      Case:

      i) If *H* does not identify an allocated CLI descriptor area, then an exception condition is raised: *CLI-specific condition — invalid handle*.

      ii) Otherwise, an exception condition is raised: *CLI-specific condition — invalid attribute identifier*.

   c) If *HT* indicates CONNECTION HANDLE, then

      Case:

      i) If *H* does not identify an allocated SQL-connection, then an exception condition is raised: *CLI-specific condition — invalid handle*.

      ii) Otherwise:

         1) Let *C* be the allocated SQL-connection identified by *H*.

         2) The diagnostics area associated with *C* is emptied.

3) Case:

A) If there is no established SQL-connection associated with *C*, then an exception condition is raised: *connection exception — connection does not exist*.

B) Otherwise, let *EC* be the established SQL-connection associated with *C*.

4) If *C* has an associated established SQL-connection that is active, then let *L1* be a list containing *EC*; otherwise, let *L1* be an empty list.

d) If *HT* indicates ENVIRONMENT HANDLE, then

Case:

i) If *H* does not identify an allocated SQL-environment or if it identifies an allocated SQL-environment that is a skeleton SQL-environment, then an exception condition is raised: *CLI-specific condition — invalid handle*.

ii) Otherwise:

1) Let *E* be the allocated SQL-environment identified by *H*.

2) The diagnostics area associated with *E* is emptied.

3) Let *L* be a list of the allocated SQL-connections associated with *E*. Let *L1* be a list of the allocated SQL-connections in *L* that have an associated established SQL-connection that is active.

4) If an SQL-transaction is currently active on any of the SQL-connections contained in *L1*, then an exception condition is raised: *invalid transaction state — active SQL-transaction*.

5) Let *AM* be the value for AccessMode. If *AM* is not one of the codes in Table 32, “Values for TRANSACTION ACCESS MODE with StartTran”, then an exception condition is raised: *CLI-specific condition — invalid attribute identifier*.

6) Let *IL* be the value for IsolationLevel. If *IL* is not one of the codes in Table 31, “Values for TRANSACTION ISOLATION OPTION with StartTran”, then an exception condition is raised: *CLI-specific condition — invalid attribute identifier*.

7) Let *TXN* be the SQL-transaction that is started by this invocation of the StartTran routine.

8) If READ ONLY is specified by *AM*, then the access mode of *TXN* is set to read-only. If READ WRITE is specified by *AM*, then the access mode of *TXN* is set to read-write.

9) The isolation level of *TXN* is set to an implementation-defined isolation level that will not exhibit any of the phenomena that the isolation level indicated by *TIL* would not exhibit, as specified in Table 8, “SQL-transaction isolation levels and the three phenomena”, in ISO/IEC 9075-2.

10) *TXN* is started in each SQL-connection contained in *L1*.

6.62 TablePrivileges

Function

Return a result set that contains a list of the privileges held on the tables whose names adhere to the requested pattern(s) within tables described by the Information Schemas of the connected data source.

Definition

```
TablePrivileges (
    StatementHandle       IN     INTEGER,
    CatalogName           IN     CHARACTER(L1),
    NameLength1           IN     SMALLINT,
    SchemaName            IN     CHARACTER(L2),
    NameLength2           IN     SMALLINT,
    TableName             IN     CHARACTER(L3),
    NameLength3           IN     SMALLINT )
    RETURNS SMALLINT
```

where each of *L1*, *L2*, and *L3* has a maximum value equal to the implementation-defined maximum length of a variable-length character string.

General Rules

1) Let *S* be the allocated SQL-statement identified by StatementHandle.

2) If an open cursor is associated with *S*, then an exception condition is raised: *invalid cursor state*.

3) Let *C* be the allocated SQL-connection with which *S* is associated.

4) Let *EC* be the established SQL-connection associated with *C* and let *SS* be the SQL-server on that connection.

5) Let *TABLE_PRIVILEGES_QUERY* be a table, with the definition:

```
CREATE TABLE TABLE_PRIVILEGES_QUERY (
    TABLE_CAT          CHARACTER VARYING(128),
    TABLE_SCHEM        CHARACTER VARYING(128) NOT NULL,
    TABLE_NAME         CHARACTER VARYING(128) NOT NULL,
    GRANTOR            CHARACTER VARYING(128) NOT NULL,
    GRANTEE            CHARACTER VARYING(128) NOT NULL,
    PRIVILEGE          CHARACTER VARYING(128) NOT NULL,
    IS_GRANTABLE       CHARACTER VARYING(3) NOT NULL,
    WITH_HIERARCHY     CHARACTER VARYING(254) NOT NULL )
```

6) *TABLE_PRIVILEGES_QUERY* contains a row for each privilege in *SS*'s Information Schema TABLE_PRIVILEGES view where:

   a) Let *SUP* be the value of Supported that is returned by the execution of GetFeatureInfo with FeatureType = 'FEATURE' and FeatureId = 'C041' (corresponding to the feature “Information Schema metadata constrained by privileges”).

   b) Case:

i) If the value of *SUP* is 1 (one), then *TABLE_PRIVILEGES_QUERY* contains a row for each privilege in *SS*'s Information Schema TABLE_PRIVILEGES view.

ii) Otherwise, *TABLE_PRIVILEGES_QUERY* contains a row for each privilege in *SS*'s Information Schema TABLE_PRIVILEGES view that meets implementation-defined authorization criteria.

7) For each row of *TABLE_PRIVILEGES_QUERY*:

a) If the implementation does not support catalog names, then TABLE_CAT is the null value; otherwise, the value of TABLE_CAT in *TABLE_PRIVILEGES_QUERY* is the value of the TABLE_CATALOG column in the TABLE_PRIVILEGES view in the Information Schema.

b) The value of TABLE_SCHEM in *TABLE_PRIVILEGES_QUERY* is the value of the TABLE_SCHEMA column in the TABLE_PRIVILEGES view.

c) The value of TABLE_NAME in *TABLE_PRIVILEGES_QUERY* is the value of the TABLE_NAME column in the TABLE_PRIVILEGES view.

d) The value of GRANTOR in *TABLE_PRIVILEGES_QUERY* is the value of the GRANTOR column in the TABLE_PRIVILEGES view.

e) The value of GRANTEE in *TABLE_PRIVILEGES_QUERY* is the value of the GRANTEE column in the TABLE_PRIVILEGES view.

f) The value of PRIVILEGE in *TABLE_PRIVILEGES_QUERY* is the value of the PRIVILEGE_TYPE column in the TABLE_PRIVILEGES view.

g) The value of IS_GRANTABLE in *TABLE_PRIVILEGES_QUERY* is the value of the IS_GRANTABLE column in the TABLE_PRIVILEGES view.

h) The value of WITH_HIERARCHY in *TABLE_PRIVILEGES_QUERY* is the value of the WITH_HIERARCHY column in the TABLE_PRIVILEGES veiw.

8) Let *NL1*, *NL2*, and *NL3* be the values of NameLength1, NameLength2, and NameLength3, respectively.

9) Let *CATVAL*, *SCHVAL*, and *TBLVAL* be the values of CatalogName, SchemaName, and TableName, respectively.

10) If the METADATA ID attribute of *S* is TRUE, then:

a) If CatalogName is a null pointer and the value of the CATALOG NAME information type from Table 29, “Codes and data types for implementation information”, is 'Y', then an exception condition is raised: *CLI-specific condition — invalid use of null pointer*.

b) If SchemaName is a null pointer or if TableName is a null pointer, then an exception condition is raised: *CLI-specific condition — invalid use of null pointer*.

11) If CatalogName is a null pointer, then *NL1* is set to zero. If SchemaName is a null pointer, then *NL2* is set to zero. If TableName is a null pointer, then *NL3* is set to zero.

12) Case:

a) If *NL1* is not negative, then let *L* be *NL1*.

b) If *NL1* indicates NULL TERMINATED, then let *L* be the number of octets of CatalogName that precede the implementation-defined null character that terminates a C character string.

c) Otherwise, an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

Let *CATVAL* be the first *L* octets of CatalogName.

13) Case:

a) If *NL2* is not negative, then let *L* be *NL2*.

b) If *NL2* indicates NULL TERMINATED, then let *L* be the number of octets of SchemaName that precede the implementation-defined null character that terminates a C character string.

c) Otherwise, an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

Let *SCHVAL* be the first *L* octets of SchemaName.

14) Case:

a) If *NL3* is not negative, then let *L* be *NL3*.

b) If *NL3* indicates NULL TERMINATED, then let *L* be the number of octets of TableName that precede the implementation-defined null character that terminates a C character string.

c) Otherwise, an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

Let *TBLVAL* be the first *L* octets of TableName.

15) Case:

a) If the METADATA ID attribute of *S* is TRUE, then:

i) Case:

1) If the value of *NL1* is zero, then let *CATSTR* be a zero-length string.

2) Otherwise:

Case:

A) If `SUBSTRING(TRIM('CATVAL') FROM 1 FOR 1) = '"'` and if `SUBSTRING(TRIM('CATVAL') FROM CHAR_LENGTH(TRIM('CATVAL')) FOR 1) = '"'`, then let *TEMPSTR* be the value obtained from evaluating:

```
SUBSTRING(TRIM('CATVAL') FROM 2
  FOR CHAR_LENGTH(TRIM('CATVAL')) - 2)
```

and let *CATSTR* be the character string:

```
TABLE_CAT = 'TEMPSTR' AND
```

B) Otherwise, let *CATSTR* be the character string:

```
UPPER(TABLE_CAT) = UPPER('CATVAL') AND
```

ii) Case:

1) If the value of *NL2* is zero, then let *SCHSTR* be a zero-length string.

2) Otherwise:

Case:

A) If `SUBSTRING(TRIM('`*`SCHVAL`*`') FROM 1 FOR 1) = '"'` and if `SUBSTRING(TRIM('`*`SCHVAL`*`') FROM CHAR_LENGTH(TRIM('`*`SCHVAL`*`')) FOR 1) = '"'`, then let *TEMPSTR* be the value obtained from evaluating:

```
SUBSTRING(TRIM('SCHVAL') FROM 2
  FOR CHAR_LENGTH(TRIM('SCHVAL')) - 2)
```

and let *SCHSTR* be the character string:

```
TABLE_SCHEM = 'TEMPSTR' AND
```

B) Otherwise, let *SCHSTR* be the character string:

```
UPPER(TABLE_SCHEM) = UPPER('SCHVAL') AND
```

iii) Case:

1) If the value of *NL3* is zero, then let *TBLSTR* be a zero-length string.

2) Otherwise:

Case:

A) If `SUBSTRING(TRIM('`*`TBLVAL`*`') FROM 1 FOR 1) = '"'` and if `SUBSTRING(TRIM('`*`TBLVAL`*`') FROM CHAR_LENGTH(TRIM('`*`TBLVAL`*`')) FOR 1) = '"'`, then let *TEMPSTR* be the value obtained from evaluating:

```
SUBSTRING(TRIM('TBLVAL') FROM 2
  FOR CHAR_LENGTH(TRIM('TBLVAL')) - 2)
```

and let *TBLSTR* be the character string:

```
TABLE_NAME = 'TEMPSTR' AND
```

B) Otherwise, let *TBLSTR* be the character string:

```
UPPER(TABLE_NAME) = UPPER('TBLVAL') AND
```

b) Otherwise:

i) Let *SPC* be the Code value from Table 29, “Codes and data types for implementation information”, that corresponds to the Information Type SEARCH PATTERN ESCAPE in that same table.

ii) Let *ESC* be the value of InfoValue that is returned by the execution of GetInfo() with the value of InfoType set to *SPC*.

iii) If the value of *NL1* is zero, then let *CATSTR* be a zero-length string; otherwise, let *CATSTR* be the character string:

```
TABLE_CAT = 'CATVAL' AND
```

iv) If the value of *NL2* is zero, then let *SCHSTR* be a zero-length string; otherwise, let *SCHSTR* be the character string:

```
TABLE_SCHEM LIKE 'SCHVAL' ESCAPE 'ESC' AND
```

v) If the value of *NL3* is zero, then let *TBLSTR* be a zero-length string; otherwise, let *TBLSTR* be the character string:

```
TABLE_NAME LIKE 'TBLVAL' ESCAPE 'ESC' AND
```

16) Let *PRED* be the result of evaluating:

```
CATSTR || ' ' || SCHSTR || ' ' || TBLSTR || ' ' || 1=1
```

17) Let *STMT* be the character string:

```
SELECT *
FROM TABLE_PRIVILEGES_QUERY
WHERE PRED
ORDER BY TABLE_CAT, TABLE_SCHEM, TABLE_NAME, PRIVILEGE
```

18) ExecDirect is implicitly invoked with *S* as the value of StatementHandle, *STMT* as the value of Statement-Text, and the length of *STMT* as the value of TextLength.

6.63 Tables

Function

Based on the specified selection criteria, return a result set that contains information about tables described by the Information Schema of the connected data source.

Definition

```
Tables (
    StatementHandle       IN     INTEGER,
    CatalogName           IN     CHARACTER(L1),
    NameLength1           IN     SMALLINT,
    SchemaName            IN     CHARACTER(L2),
    NameLength2           IN     SMALLINT,
    TableName             IN     CHARACTER(L3),
    NameLength3           IN     SMALLINT,
    TableType             IN     CHARACTER(L4),
    NameLength4           IN     SMALLINT )
    RETURNS SMALLINT
```

where each of *L1*, *L2*, *L3*, and *L4* has a maximum value equal to the implementation-defined maximum length of a variable-length character string.

General Rules

1) Let *S* be the allocated SQL-statement identified by StatementHandle.

2) If an open cursor is associated with *S*, then an exception condition is raised: *invalid cursor state*.

3) Let *C* be the allocated SQL-connection with which *S* is associated.

4) Let *EC* be the established SQL-connection associated with *C* and let *SS* be the SQL-server on that connection.

5) Let *TABLES_QUERY* be a table with the definition:

```
CREATE TABLE TABLES_QUERY (
    TABLE_CAT         CHARACTER VARYING(128),
    TABLE_SCHEM       CHARACTER VARYING(128),
    TABLE_NAME        CHARACTER VARYING(128),
    TABLE_TYPE        CHARACTER VARYING(254),
    REMARKS           CHARACTER VARYING(254),
    SELF_REF_COLUMN   CHARACTER VARYING(128),
    REF_GENERATION    CHARACTER VARYING(254),
    UDT_CAT           CHARACTER VARYING(128),
    UDT_SCHEM         CHARACTER VARYING(128),
    UDT_NAME          CHARACTER VARYING(128),
    UNIQUE (TABLE_CAT, TABLE_SCHEM, TABLE_NAME) )
```

6) *TABLES_QUERY* contains a row for each table described by *SS*'s Information Schema TABLES view where:

a) Let *SUP* be the value of Supported that is returned by the execution of GetFeatureInfo with FeatureType = 'FEATURE' and FeatureId = 'C041' (corresponding to the feature “Information Schema metadata constrained by privileges”).

b) Case:

i) If the value of *SUP* is 1 (one), then *TABLES_QUERY* contains a row for each row describing a table in *SS*'s Information Schema TABLES view for which the connected UserName has selection privileges.

ii) Otherwise, *TABLES_QUERY* contains a row for each row describing a table in *SS*'s Information Schema TABLES view that meets implementation-defined authorization criteria.

7) The description of the table *TABLES_QUERY* is:

a) The value of TABLE_CAT in *TABLES_QUERY* is the value of the TABLE_CATALOG column in the TABLES view. If *SS* does not support catalog names, then TABLE_CAT is set to the null value.

b) The value of TABLE_SCHEM in *TABLES_QUERY* is the value of the TABLE_SCHEMA column in the TABLES view. The value of TABLE_NAME in *TABLES_QUERY* is the value of the TABLE_NAME column in the TABLES view.

c) The value of TABLE_TYPE in *TABLES_QUERY* is determined by the values of the TABLE_TYPE column in the TABLES view.

Case:

i) If the value of TABLE_TYPE in the TABLES view is 'VIEW', then

Case:

1) If the defined view is within the Information Schema itself, then the value of TABLE_TYPE in *TABLES_QUERY* is set to 'SYSTEM TABLE''.

2) Otherwise, the value of TABLE_TYPE in *TABLES_QUERY* is set to 'VIEW'.

ii) If the value of TABLE_TYPE in the TABLES view is 'BASE TABLE', then the value of TABLE_TYPE in *TABLES_QUERY* is set to 'TABLE'.

iii) If the value of TABLE_TYPE in the TABLES view is 'GLOBAL TEMPORARY' or 'LOCAL TEMPORARY', then the value of TABLE_TYPE in *TABLES_QUERY* is set to that value.

iv) Otherwise, the value of TABLE_TYPE in *TABLES_QUERY* is an implementation-defined value.

d) The value of REMARKS in *TABLES_QUERY* is an implementation-defined description of the table.

e) The value of SELF_REF_COLUMN in *TABLES_QUERY* is the value of the SELF_REFERENCING_COLUMN_NAME column in the TABLES view.

f) The value of REF_GENERATION in *TABLES_QUERY* is the value of the REFERENCE_GENERATION column in the TABLES view.

g) The value of UDT_CAT in *TABLES_QUERY* is the value of the USER_DEFINED_TYPE_CATALOG column in the TABLES view.

h) The value of UDT_SCHEMA in *TABLES_QUERY* is the value of the USER_DEFINED_TYPE_SCHEMA column in the TABLES view.

i) The value of UDT_NAME in *TABLES_QUERY* is the value of the USER_DEFINED_TYPE_NAME column in the TABLES view.

8) Let *NL1*, *NL2*, *NL3*, and *NL4* be the values of NameLength1, NameLength2, NameLength3, and NameLength4, respectively.

9) Let *CATVAL*, *SCHVAL*, *TBLVAL*, and *TYPVAL* be the values of CatalogName, SchemaName, TableName, and TableType, respectively.

10) If the METADATA ID attribute of *S* is TRUE, then:

a) If CatalogName is a null pointer and the value of the CATALOG NAME information type from Table 29, “Codes and data types for implementation information”, is 'Y', then an exception condition is raised: *CLI-specific condition — invalid use of null pointer*.

b) If SchemaName is a null pointer or if TableName is a null pointer, then an exception condition is raised: *CLI-specific condition — invalid use of null pointer*.

11) If CatalogName is a null pointer, then *NL1* is set to zero. If SchemaName is a null pointer, then *NL2* is set to zero. If TableName is a null pointer, then *NL3* is set to zero. If TableType is a null pointer, then *NL4* is set to zero.

12) Case:

a) If *NL1* is not negative, then let *L* be *NL1*.

b) If *NL1* indicates NULL TERMINATED, then let *L* be the number of octets of CatalogName that precede the implementation-defined null character that terminates a C character string.

c) Otherwise, an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

Let *CATVAL* be the first *L* octets of CatalogName.

13) Case:

a) If *NL2* is not negative, then let *L* be *NL2*.

b) If *NL2* indicates NULL TERMINATED, then let *L* be the number of octets of SchemaName that precede the implementation-defined null character that terminates a C character string.

c) Otherwise, an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

Let *SCHVAL* be the first *L* octets of SchemaName.

14) Case:

a) If *NL3* is not negative, then let *L* be *NL3*.

b) If *NL3* indicates NULL TERMINATED, then let *L* be the number of octets of TableName that precede the implementation-defined null character that terminates a C character string.

c) Otherwise, an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

Let *TBLVAL* be the first *L* octets of TableName.

15) Case:

a) If *NL4* is not negative, then let *L* be *NL4*.

b) If *NL4* indicates NULL TERMINATED, then let *L* be the number of octets of TableType that precede the implementation-defined null character that terminates a C character string.

c) Otherwise, an exception condition is raised: *CLI-specific condition — invalid string length or buffer length*.

Let *TYPVAL* be the first *L* octets of ColumnName.

16) Case:

a) If the METADATA ID attribute of *S* is TRUE, then:

i) Case:

1) If the value of *NL1* is zero, then let *CATSTR* be a zero-length string.

2) Otherwise,

Case:

A) If `SUBSTRING(TRIM('CATVAL') FROM 1 FOR 1) = '"'` and if `SUBSTRING(TRIM('CATVAL') FROM CHAR_LENGTH(TRIM('CATVAL')) FOR 1) = '"'`, then let *TEMPSTR* be the value obtained from evaluating:

```
SUBSTRING ( TRIM('CATVAL') FROM 2
  FOR CHAR_LENGTH ( TRIM('CATVAL') ) - 2 )
```

and let *CATSTR* be the character string:

```
TABLE_CAT = 'TEMPSTR' AND
```

B) Otherwise, let *CATSTR* be the character string:

```
UPPER(TABLE_CAT) = UPPER('CATVAL') AND
```

ii) Case:

1) If the value of *NL2* is zero, then let *SCHSTR* be a zero-length string.

2) Otherwise,

Case:

A) If `SUBSTRING(TRIM('SCHVAL') FROM 1 FOR 1) = '"'` and if `SUBSTRING(TRIM('SCHVAL') FROM CHAR_LENGTH(TRIM('SCHVAL')) FOR 1) = '"'`, then let *TEMPSTR* be the value obtained from evaluating:

```
SUBSTRING ( TRIM('SCHVAL') FROM 2
  FOR CHAR_LENGTH ( TRIM('SCHVAL') ) - 2 )
```

and let *SCHSTR* be the character string:

```
TABLE_SCHEM = 'TEMPSTR' AND
```

B) Otherwise, let *SCHSTR* be the character string:

```
UPPER(TABLE_SCHEM) = UPPER('SCHVAL') AND
```

iii) Case:

1) If the value of *NL3* is zero, then let *TBLSTR* be a zero-length string.

2) Otherwise,

Case:

A) If `SUBSTRING(TRIM('TBLVAL') FROM 1 FOR 1) = '"'` and if `SUBSTRING(TRIM('TBLVAL') FROM CHAR_LENGTH(TRIM('TBLVAL')) FOR 1) = '"'`, then let *TEMPSTR* be the value obtained from evaluating:

```
SUBSTRING ( TRIM('TBLVAL') FROM 2
            FOR CHAR_LENGTH ( TRIM('TBLVAL') ) - 2 )
```

and let *TBLSTR* be the character string:

```
TABLE_NAME = 'TEMPSTR' AND
```

B) Otherwise, let *TBLSTR* be the character string:

```
UPPER(TABLE_NAME) = UPPER('TBLVAL') AND
```

b) Otherwise:

i) Let *SPC* be the Code value from Table 29, “Codes and data types for implementation information”, that corresponds to the Information Type SEARCH PATTERN ESCAPE in that same table.

ii) Let *ESC* be the value of InfoValue that is returned by the execution of GetInfo() with the value of InfoType set to *SPC*.

iii) If the value of *NL1* is zero, then let *CATSTR* be a zero-length string; otherwise, let *CATSTR* be the character string:

```
TABLE_CAT = 'CATVAL' AND
```

iv) If the value of *NL2* is zero, then let *SCHSTR* be a zero-length string; otherwise, let *SCHSTR* be the character string:

```
TABLE_SCHEM LIKE 'SCHVAL' ESCAPE 'ESC' AND
```

v) If the value of *NL3* is zero, then let *TBLSTR* be a zero-length string; otherwise, let *TBLSTR* be the character string:

```
TABLE_NAME LIKE 'TBLVAL' ESCAPE 'ESC' AND
```

17) Case:

a) If the value of *NL4* is zero, then let *TYPSTR* be a zero-length string.

b) Otherwise,

i) TableType is a comma-separated list of one or more types of tables that are to be returned in the result set. Each value may optionally be enclosed within <quote> characters. The types are 'TABLE', 'VIEW', 'GLOBAL TEMPORARY', 'LOCAL TEMPORARY', and 'SYSTEM TABLE'.

NOTE 53 — These types are mutually exclusive; for instance, 'TABLE' includes only user-created base tables and 'SYSTEM TABLE' includes only views from the Information Schema. Implementation-defined types may also be specified.

ii) Let *N* be the number of comma-separated values specified within TableType.

iii) Let *TT* be the set of comma-separated values TT_i, 1 (one) ≤ *i* ≤ *N*, specified within TableType.

iv) *TYPSTR* is a string that is the predicate required to select the requested types of tables from *TABLES_QUERY*:

```
TABLE_TYPE = '''' || TRIM(TT1) || '''' OR
TABLE_TYPE = '''' || TRIM(TT2) || '''' OR
...
TABLE_TYPE = '''' || TRIM(TTN) || ''''
```

18) Let *PRED* be the result of evaluating:

```
CATSTR || ' ' || SCHSTR || ' ' || TBLSTR || ' ' || TYPSTR || ' ' || 1=1
```

19) Case:

a) If the value of *CATVAL* is the value in the 'Value' column for ALL CATALOGS in Table 38, “Special parameter values”, and both *SCHVAL* and *TBLVAL* are zero-length strings, then let *STMT* be the character string:

```
SELECT DISTINCT TABLE_CAT,
                CAST (NULL AS VARCHAR(128)),
                CAST (NULL AS VARCHAR(128)),
                CAST (NULL AS VARCHAR(254)),
                CAST (NULL AS VARCHAR(254))
FROM TABLES_QUERY
ORDER BY TABLE_CAT
```

NOTE 54 — All tables qualify for selection and no privileges are required for access to the underlying TABLES view.

b) If the value of *SCHVAL* is the value in the 'Value' column for ALL SCHEMAS in Table 38, “Special parameter values”, and both *CATVAL* and *TBLVAL* are zero-length strings, then let *STMT* be the character string:

```
SELECT DISTINCT CAST (NULL AS VARCHAR(128)),
                TABLE_SCHEM,
                CAST (NULL AS VARCHAR(128)),
                CAST (NULL AS VARCHAR(254)),
                CAST (NULL AS VARCHAR(254))
FROM TABLES_QUERY
ORDER BY TABLE_SCHEM
```

NOTE 55 — All tables qualify for selection and no privileges are required for access to the underlying TABLES view.

c) If the value of *TYPVAL* is the value in the 'Value' column for ALL TYPES in Table 38, “Special parameter values”, and *CATVAL*, *SCHVAL*, and *TBLVAL* are zero-length strings, then let *STMT* be the character string:

```
SELECT DISTINCT CAST (NULL AS VARCHAR(128)),
                CAST (NULL AS VARCHAR(128)),
                CAST (NULL AS VARCHAR(128)),
                TABLE_TYPE,
                CAST (NULL AS VARCHAR(254))
FROM TABLES_QUERY
ORDER BY TABLE_TYPE
```

NOTE 56 — All tables qualify for selection and no privileges are required for access to the underlying TABLES view.

d) Otherwise, let *STMT* be the character string:

```
SELECT *
FROM TABLES_QUERY
WHERE PRED
ORDER BY TABLE_TYPE, TABLE_CAT, TABLE_SCHEM, TABLE_NAME
```

20) ExecDirect is implicitly invoked with *S* as the value of StatementHandle, *STMT* as the value of StatementText, and the length of *STMT* as the value of TextLength.

(Blank page)

7 Definition Schema

This Clause modifies Clause 6, "Definition Schema", in ISO/IEC 9075-11.

7.1 SQL_IMPLEMENTATION_INFO base table

This Subclause modifies Subclause 6.44, "SQL_IMPLEMENTATION_INFO base table", in ISO/IEC 9075-11.

Function

The SQL_IMPLEMENTATION_INFO base table has one row for each implementation information item defined by ISO/IEC 9075.

Definition

```
No additional Definition.
```

Description

1) Insert this description Some IMPLEMENTATION_INFO_ID values assigned by ISO/IEC 9075 have been assigned for backwards compatibility with ISO/IEC 9075-3:1995. All other values assigned by ISO/IEC 9075 are in the range 21000 through 24999, inclusive.

2) Insert this description Implementation-defined items that are represented in this table shall have an IMPLEMENTATION_INFO_ID value that is in the range 11000 through 14999, inclusive.

Table Population

The implementation shall effectively populate the SQL_IMPLEMENTATION_INFO base table with an <insert statement> that is equivalent to the <insert statement> shown below; the <insert statement> shown below provides values only for certain columns and implicitly assigns the null value to other columns of the table.

The implementation effectively populates the table so that, for each row containing information about some facility that the implementation supports, either the INTEGER_VALUE column or the CHARACTER_VALUE column is set to a value that specifies the requisite information about that supported facility. For all information items that the implementation does not support, both the INTEGER_VALUE and the CHARACTER_VALUE column have the null value. The COMMENTS column may be set to any value deemed appropriate by the implementation, or it may be set to the null value. The following INSERT statement specifies values for the COMMENTS column that may be used by an implementation if it so chooses.

— Let *CAT* be 'Y' if the SQL-implementation supports catalog names and 'N' otherwise.

— Let *COLL* be the default collation name for the SQL-implementation.

— Let *CCB* be an integer representing the default cursor commit behavior of the SQL-implementation: 0 (zero) if the SQL-implementation closes cursors and deleted prepared statements, 1 (one) if it closes cursors and retains prepared statements, and 2 if it leaves cursors open and retains prepared statements.

— Let *DSN* be the connection name used in a CONNECT statement when connecting to the SQL-implementation.

— Let *DBMS* be the name of the SQL-implementation software (*e.g.*, the product name).

— Let *VER* be a character representation of the version of the implementation software comprising two digits, a period, two more digits, another period, four more digits, and optionally a sequence of characters.

— Let *DTI* be a number indicating the default transaction isolation level of the SQL-implementation: 1 (one) for READ UNCOMMITED, 2 for READ COMMITTED, 3 for REPEATABLE READ, and 4 for SERIALIZABLE.

— Let *IDC* be a number indicating the treatment of identifiers when stored in the SQL-implementation's metadata: 1 (one) indicates they are stored in all upper case characters, 2 that they are stored in all lower case characters, 3 that they are stored in mixed case and that they are case sensitive, and 4 that they are stored in mixed case but are case insensitive.

— Let *NCOL* be a number indicating how nulls are collated: 0 (zero) indicates that nulls are collated higher than non-null values, and 1 (one) indicates that they are collated lower than non-null values.

— Let *SERV* be the SQL server name used by a CONNECT statement when connecting to the SQL-implementation.

— Let *SPEC* be a character string containing all special characters that are allowed in nondelimited identifiers.

— Let *TXC* be a number indicating the transaction capabilities of the SQL-implementation: 0 (zero) indicates that transactions are not supported, 1 (one) that they are supported only for DML statements, 2 that they are supported for both DML and DDL statements, 3 that they are supported only for DML statements and that an implicit COMMIT occurs before any DDL statements are executed, and 4 that they are supported only for DML statements and that DDL statements are ignored for the purposes of transactions.

```
INSERT INTO sql_implementation_info ( implementation_info_id,
                                      implementation_info_name,
                                      integer_value,
                                      character_value,
                                      comments )
  VALUES ( 10003, 'CATALOG NAME',
             NULL, 'CAT',
             'CHAR: ''Y'' if supported, otherwise ''N''' ),
           ( 10004, 'COLLATING SEQUENCE',
             NULL, 'COLL',
             'CHAR: default collation name' ),
           ( 23, 'CURSOR COMMIT BEHAVIOR',
             CCB, NULL,
             'INT: 0: close cursors & delete prepared stmts
                   1: close cursors & retain prepared stmts
```

```
       2: leave cursors open & retain stmts' ),
( 2, 'DATA SOURCE NAME',
  NULL, 'DSN',
  'CHAR: <connection name> on CONNECT statement' ),
( 17, 'DBMS NAME',
  NULL, 'DBMS',
  'CHAR: Name of the implementation software' ),
( 18, 'DBMS VERSION',
  NULL, 'VER',
  'CHAR: Version of the implementation software
         The format is:
           <part1>.<part2>.<part3>[<part4>]
         where:
           <part1> ::= <digit><digit>

           <part2> ::= <digit><digit>

           <part3> ::= <digit><digit><digit><digit>
           <part4> ::= <character representation>' ),
( 26, 'DEFAULT TRANSACTION ISOLATION',
  DTI, NULL,
  'INT: 1: READ UNCOMMITTED
        2: READ COMMITTED
        3: REPEATABLE READ
        4: SERIALIZABLE' ),
( 28, 'IDENTIFIER CASE',
  IDC, NULL,
  'The case in which identifiers are stored in the Definition Schema
  INT: 1: stored in upper case
       2: stored in lower case
       3: stored in mixed case - case sensitive
       4: stored in mixed case - case insensitive' ),
( 85, 'NULL COLLATION',
  NCOL, NULL,
  'INT: 0: nulls higher than non-nulls
        1: nulls lower than non-nulls' ),
( 13, 'SERVER NAME',
  NULL, 'SERV',
  'CHAR: <SQL server name> on CONNECT statement' ),
( 94, 'SPECIAL CHARACTERS',
  NULL, 'SPEC',
  'CHAR: All special chars OK in non-delimited ids' ),
( 46, 'TRANSACTION CAPABLE',
  TXC, NULL,
  'INT: 0: not supported
        1: DML only - error if DDL
        2: both DML and DDL
        3: DML only - commit before DDL
        4: DML only - ignore DDL' );
```

7.2 SQL_SIZING base table

This Subclause is modified by Subclause 25.11, "SQL_SIZING base table", in ISO/IEC 9075-9.

This Subclause modifies Subclause 6.45, "SQL_SIZING base table", in ISO/IEC 9075-11.

Function

The SQL_SIZING base table has one row for each sizing item defined by ISO/IEC 9075.

Definition

```
No additional Definition.
```

Description

1) Insert this description Some SIZING_ID values assigned by ISO/IEC 9075 have been assigned for backwards compatibility with ISO/IEC 9075-3:1995. All other values assigned by ISO/IEC 9075 are in the range 25000 through 29999, inclusive.

2) Insert this description Implementation-defined items that are represented in this table shall have a SIZING_ID value that is in the range 15000 through 19999, inclusive.

Table Population

The implementation shall effectively populate the SQL_SIZING base table with an <insert statement> that is equivalent to the <insert statement> shown below; the <insert statement> shown below provides values only for certain columns and implicitly assigns the null value to other columns of the table.

The implementation effectively populates the table so that, for each row containing information about some facility that the implementation supports, the SUPPORTED_VALUE column is set to a value that specifies the requisite information about that supported facility. For all information items that the implementation does not support, the SUPPORTED_VALUE column has the null value. The COMMENTS column may be set to any value deemed appropriate by the implementation, or it may be set to the null value.

```
INSERT INTO sql_sizing ( sizing_id, sizing_name, comments )
  VALUES ( 34, 'MAXIMUM CATALOG NAME LENGTH',
                 'Length in characters' ),
           ( 30, 'MAXIMUM COLUMN NAME LENGTH',
                 'Length in characters' ),
           ( 97, 'MAXIMUM COLUMNS IN GROUP BY', NULL ),
           ( 99, 'MAXIMUM COLUMNS IN ORDER BY', NULL ),
           ( 100, 'MAXIMUM COLUMNS IN SELECT,
                  'Max number of expressions in <select list>' ),
           ( 101, 'MAXIMUM COLUMNS IN TABLE', NULL ),
           ( 1, 'MAXIMUM CONCURRENT ACTIVITIES',
```

```
     'Max number of SQL-statements currently active' ),
( 31, 'MAXIMUM CURSOR NAME LENGTH',
      'Length in characters' ),
( 0, 'MAXIMUM DRIVER CONNECTIONS',
     'Max number of SQL-connections currently established' ),
( 10005, 'MAXIMUM IDENTIFIER LENGTH',
         'Length in characters;
          If different for some objects, set to smallest max' ),
( 32, 'MAXIMUM SCHEMA NAME LENGTH,
      'Length in characters' ),
( 20000, 'MAXIMUM STATEMENT OCTETS',
         'Max length in octets of <SQL statement variable>' ),
( 20001, 'MAXIMUM STATEMENT OCTETS DATA',
         'Max length in octets of <SQL data statement>' ),
( 20002, 'MAXIMUM STATEMENT OCTETS SCHEMA',
         'Max length in octets of SQL <schema definition>' ),
( 35, 'MAXIMUM TABLE NAME LENGTH',
      'Max length in chars of low order table name part' ),
( 106, 'MAXIMUM TABLES IN SELECT',
       'Max number of table names in FROM clause' ),
( 107, 'MAXIMUM USER NAME LENGTH',
       'Length in characters for a <user identifier> of an SQL-session' ),
( 25000, 'MAXIMUM CURRENT DEFAULT TRANSFORM GROUP LENGTH',
         'Length in characters' ),
( 25001, 'MAXIMUM CURRENT TRANSFORM GROUP LENGTH',
         'Length in characters' ),
( 25002, 'MAXIMUM CURRENT PATH LENGTH',
         'Length in characters' ),
( 25003, 'MAXIMUM CURRENT ROLE LENGTH',
         'Length in characters' ),
( 25004, 'MAXIMUM SESSION USER LENGTH',
         'Length in characters' ),
( 25005, 'MAXIMUM SYSTEM USER LENGTH',
         'Length in characters' );
```

(Blank page)

8 Conformance

8.1 Claims of conformance to SQL/CLI

In addition to the requirements of ISO/IEC 9075-1, Clause 8, “Conformance”, a claim of conformance to this part of ISO/IEC 9075 shall:

1) Claim conformance to at least one of:
   - Feature C001, “CLI routine invocation in Ada”
   - Feature C002, “CLI routine invocation in C”
   - Feature C003, “CLI routine invocation in COBOL”
   - Feature C004, “CLI routine invocation in Fortran”
   - Feature C005, “CLI routine invocation in MUMPS”
   - Feature C006, “CLI routine invocation in Pascal”
   - Feature C007, “CLI routine invocation in PL/I”

8.2 Additional conformance requirements for SQL/CLI

A claim of conformance to this part of ISO/IEC 9075 implies that a conforming SQL/CLI implementation shall correctly process all SQL language that is conforming SQL language in terms of the claim of conformance to ISO/IEC 9075.

NOTE 57 — This includes all optional features to which conformance is claimed.

NOTE 58 — Certain facilities specified in this part of ISO/IEC 9075 are closely related to specific facilities specified in ISO/IEC 9075-2; such facilities specified in this part of ISO/IEC 9075 are not supported unless the corresponding facilities in ISO/IEC 9075-2 are supported. The relationships between the facilities specified in this part of ISO/IEC 9075 and the corresponding facilities in ISO/IEC 9075-2 are not specified, but are inferable.

For example, provision of the GetPosition, GetSubstring, and GetLength routines specified in this part of ISO/IEC 9075 is dependent on support of the LARGE OBJECT data types specified in ISO/IEC 9075-2.

8.3 Implied feature relationships of SQL/CLI

Table 47 — Implied feature relationships of SQL/CLI

Feature ID	Feature Name	Implied Feature ID	Implied Feature Name
(none)			

Annex A
(informative)

Typical header files

This Annex is modified by Annex E, "Typical header files", in ISO/IEC 9075-9.

A.1 C header file SQLCLI.H

This Subclause is modified by Subclause E.1, "C Header File SQLCLI.H", in ISO/IEC 9075-9.

Here is a typical SQLCLI.H file. C applications include this file by containing the following statement:

```
#include "sqlcli.h"
```

The following file contains C language function prototypes for the SQL/CLI routines.

```
/* sqlcli.h  Header File for SQL CLI.
 * The actual header file shall contain at least the information
 * specified here, except that the comments may vary.
                                                               */
/* API declaration data types                                  */
typedef unsigned char   SQLCHAR;
typedef void *          SQLPOINTER;
typedef unsigned char   SQLCLOB;
typedef long            SQLCLOB_LOCATOR;
typedef unsigned char   SQLBLOB;
typedef long            SQLBLOB_LOCATOR;
typedef unsigned char   SQLNUMERIC;
typedef unsigned char   SQLDECIMAL;
typedef short           SQLSMALLINT;
typedef long            SQLINTEGER;
typedef long long       SQLBIGINT;
typedef float           SQLREAL;
typedef double          SQLDOUBLE;
typedef unsigned char   SQLDATE;
typedef unsigned char   SQLTIME;
typedef unsigned char   SQLTIMESTAMP;
typedef unsigned char   SQLINTERVAL;
typedef long            SQLUDT_LOCATOR;
typedef unsigned char   SQLREF;
typedef long            SQLARRAY_LOCATOR;
typedef long            SQLMULTISET_LOCATOR;
typedef unsigned char   SQLBINARY;
typedef unsigned char   SQLVARBINARY;
/* Function return type                                        */
typedef SQLSMALLINT     SQLRETURN;
```

```
/* Generic data structures                                         */
typedef SQLINTEGER      SQLHENV;    /* environment handle    */
typedef SQLINTEGER      SQLHDBC;    /* connection handle     */
typedef SQLINTEGER      SQLHSTMT;   /* statement handle      */
typedef SQLINTEGER      SQLHDESC;   /* descriptor handle     */
/* Special length/indicator values                                 */
#define SQL_NULL_DATA                               -1
#define SQL_DATA_AT_EXEC                            -2
/* Return values from functions                                    */
#define SQL_SUCCESS                                  0
#define SQL_SUCCESS_WITH_INFO                        1
#define SQL_NEED_DATA                               99
#define SQL_NO_DATA                                100
#define SQL_ERROR                                   -1
#define SQL_INVALID_HANDLE                          -2
/* Row status values after a call to a fetch function        */
#define SQL_ROW_SUCCESS                              0
#define SQL_ROW_SUCCESS_WITH_INFO                    6
#define SQL_ROW_ERROR                                5
#define SQL_ROW_NO_ROW                               3
/* Test for SQL_SUCCESS or SQL_SUCCESS_WITH_INFO             */
#define SQL_SUCCEEDED(rc) (((rc)&(~1))==0)
/* flags for null-terminated string                                */
#define SQL_NTS                                     -3
#define SQL_NTSL                                   -3L
/* Maximum message length                                          */
#define SQL_MAXIMUM_MESSAGE_LENGTH                 512
/* Handle type identifiers                                         */
#define SQL_HANDLE_ENV                               1
#define SQL_HANDLE_DBC                               2
#define SQL_HANDLE_STMT                              3
#define SQL_HANDLE_DESC                              4
/* Environment attribute                                           */
#define SQL_ATTR_OUTPUT_NTS                      10001
/* Connection attribute */
#define SQL_ATTR_AUTO_IPD                        10001
#define SQL_ATTR_SAVEPOINT_NAME                  10027
/* Statement attributes                                            */
#define SQL_ATTR_CURSOR_SCROLLABLE                  -1
#define SQL_ATTR_CURSOR_SENSITIVITY                 -2
#define SQL_ATTR_CURSOR_HOLDABLE                    -3
#define SQL_ATTR_APP_ROW_DESC                    10010
#define SQL_ATTR_APP_PARAM_DESC                  10011
#define SQL_ATTR_IMP_ROW_DESC                    10012
#define SQL_ATTR_IMP_PARAM_DESC                  10013
#define SQL_ATTR_METADATA_ID                     10014
#define SQL_ATTR_CURRENT_OF_POSITION             10027
#define SQL_ATTR_NEST_DESCRIPTOR                 10029
/* Identifiers of fields in the SQL/CLI item descriptor area */
#define SQL_DESC_ARRAY_SIZE                         20
#define SQL_DESC_ARRAY_STATUS_POINTER               21
#define SQL_DESC_DATETIME_INTERVAL_PRECISION        26
#define SQL_DESC_ROWS_PROCESSED_POINTER             34
#define SQL_DESC_COUNT                            1001
#define SQL_DESC_TYPE                             1002
#define SQL_DESC_LENGTH                           1003
#define SQL_DESC_OCTET_LENGTH_POINTER             1004
```

```
#define SQL_DESC_PRECISION                      1005
#define SQL_DESC_SCALE                          1006
#define SQL_DESC_DATETIME_INTERVAL_CODE         1007
#define SQL_DESC_NULLABLE                       1008
#define SQL_DESC_INDICATOR_POINTER              1009
#define SQL_DESC_DATA_POINTER                   1010
#define SQL_DESC_NAME                           1011
#define SQL_DESC_UNNAMED                        1012
#define SQL_DESC_OCTET_LENGTH                   1013
#define SQL_DESC_COLLATION_CATALOG              1015
#define SQL_DESC_COLLATION_SCHEMA               1016
#define SQL_DESC_COLLATION_NAME                 1017
#define SQL_DESC_CHARACTER_SET_CATALOG          1018
#define SQL_DESC_CHARACTER_SET_SCHEMA           1019
#define SQL_DESC_CHARACTER_SET_NAME             1020
#define SQL_DESC_PARAMETER_MODE                 1021
#define SQL_DESC_PARAMETER_ORDINAL_POSITION     1022
#define SQL_DESC_PARAMETER_SPECIFIC_CATALOG     1023
#define SQL_DESC_PARAMETER_SPECIFIC_SCHEMA      1024
#define SQL_DESC_PARAMETER_SPECIFIC_NAME        1025
#define SQL_DESC_UDT_CATALOG                    1026
#define SQL_DESC_UDT_SCHEMA                     1027
#define SQL_DESC_UDT_NAME                       1028
#define SQL_DESC_KEY_TYPE                       1029
#define SQL_DESC_KEY_MEMBER                     1030
#define SQL_DESC_DYNAMIC_FUNCTION               1031
#define SQL_DESC_DYNAMIC_FUNCTION_CODE          1032
#define SQL_DESC_SCOPE_CATALOG                  1033
#define SQL_DESC_SCOPE_SCHEMA                   1034
#define SQL_DESC_SCOPE_NAME                     1035
#define SQL_DESC_SPECIFIC_TYPE_CATALOG          1036
#define SQL_DESC_SPECIFIC_TYPE_SCHEMA           1037
#define SQL_DESC_SPECIFIC_TYPE_NAME             1038
#define SQL_DESC_CURRENT_TRANSFORM_GROUP        1039
#define SQL_DESC_CARDINALITY                    1040
#define SQL_DESC_DEGREE                         1041
#define SQL_DESC_LEVEL                          1042
#define SQL_DESC_RETURNED_CARDINALITY_POINTER   1043
#define SQL_DESC_TOP_LEVEL_COUNT                1044
#define SQL_DESC_USER_DEFINED_TYPE_CODE         1045
#define SQL_DESC_ALLOC_TYPE                     1099
/* Identifiers of fields in the diagnostics area            */
#define SQL_DIAG_ROW_NUMBER                    -1248
#define SQL_DIAG_COLUMN_NUMBER                 -1247
#define SQL_DIAG_RETURNCODE                        1
#define SQL_DIAG_NUMBER                            2
#define SQL_DIAG_ROW_COUNT                         3
#define SQL_DIAG_SQLSTATE                          4
#define SQL_DIAG_NATIVE_CODE                       5
#define SQL_DIAG_MESSAGE_TEXT                      6
#define SQL_DIAG_DYNAMIC_FUNCTION                  7
#define SQL_DIAG_CLASS_ORIGIN                      8
#define SQL_DIAG_SUBCLASS_ORIGIN                   9
#define SQL_DIAG_CONNECTION_NAME                  10
#define SQL_DIAG_SERVER_NAME                      11
#define SQL_DIAG_DYNAMIC_FUNCTION_CODE            12
#define SQL_DIAG_MORE                             13
```

```
#define SQL_DIAG_CONDITION_NUMBER                     14
#define SQL_DIAG_CONSTRAINT_CATALOG                   15
#define SQL_DIAG_CONSTRAINT_SCHEMA                    16
#define SQL_DIAG_CONSTRAINT_NAME                      17
#define SQL_DIAG_CATALOG_NAME                         18
#define SQL_DIAG_SCHEMA_NAME                          19
#define SQL_DIAG_TABLE_NAME                           20
#define SQL_DIAG_COLUMN_NAME                          21
#define SQL_DIAG_CURSOR_NAME                          22
#define SQL_DIAG_MESSAGE_LENGTH                       23
#define SQL_DIAG_MESSAGE_OCTET_LENGTH                 24
#define SQL_DIAG_CONDITION_IDENTIFIER                 25
#define SQL_DIAG_PARAMETER_NAME                       26
#define SQL_DIAG_ROUTINE_CATALOG                      27
#define SQL_DIAG_ROUTINE_SCHEMA                       28
#define SQL_DIAG_ROUTINE_NAME                         29
#define SQL_DIAG_SPECIFIC_NAME                        30
#define SQL_DIAG_TRIGGER_CATALOG                      31
#define SQL_DIAG_TRIGGER_SCHEMA                       32
#define SQL_DIAG_TRIGGER_NAME                         33
#define SQL_DIAG_TRANSACTIONS_COMMITTED               34
#define SQL_DIAG_TRANSACTIONS_ROLLED_BACK             35
#define SQL_DIAG_TRANSACTION_ACTIVE                   36
#define SQL_DIAG_PARAMETER_MODE                       37
#define SQL_DIAG_PARAMETER_ORDINAL_POSITION           38
/* Dynamic function codes returned in diagnostics area       */
#define SQL_DIAG_ALTER_DOMAIN                          3
#define SQL_DIAG_ALTER_TABLE                           4
#define SQL_DIAG_CALL                                  7
#define SQL_DIAG_CREATE_ASSERTION                      6
#define SQL_DIAG_CREATE_CHARACTER_SET                  8
#define SQL_DIAG_CREATE_COLLATION                     10
#define SQL_DIAG_CREATE_DOMAIN                        23
#define SQL_DIAG_CREATE_SCHEMA                        64
#define SQL_DIAG_CREATE_TABLE                         77
#define SQL_DIAG_CREATE_TRANSLATION                   79
#define SQL_DIAG_CREATE_VIEW                          84
#define SQL_DIAG_DELETE_WHERE                         19
#define SQL_DIAG_DROP_ASSERTION                       24
#define SQL_DIAG_DROP_CHARACTER_SET                   25
#define SQL_DIAG_DROP_COLLATION                       26
#define SQL_DIAG_DROP_DOMAIN                          27
#define SQL_DIAG_DROP_SCHEMA                          31
#define SQL_DIAG_DROP_TABLE                           32
#define SQL_DIAG_DROP_TRANSLATION                     33
#define SQL_DIAG_DROP_VIEW                            36
#define SQL_DIAG_DYNAMIC_DELETE_CURSOR                54
#define SQL_DIAG_DYNAMIC_UPDATE_CURSOR                55
#define SQL_DIAG_GRANT                                48
#define SQL_DIAG_INSERT                               50
#define SQL_DIAG_MERGE                               128
#define SQL_DIAG_REVOKE                               59
#define SQL_DIAG_SELECT                               41
#define SQL_DIAG_SELECT_CURSOR                        85
#define SQL_DIAG_SET_CATALOG                          66
#define SQL_DIAG_SET_CONSTRAINT                       68
#define SQL_DIAG_SET_NAMES                            72
```

```
#define SQL_DIAG_SET_SCHEMA                        74
#define SQL_DIAG_SET_SESSION_AUTHORIZATION         76
#define SQL_DIAG_SET_TIME_ZONE                     71
#define SQL_DIAG_SET_TRANSACTION                   75
#define SQL_DIAG_UNKNOWN_STATEMENT                  0
#define SQL_DIAG_UPDATE_WHERE                      82
/* SQL data type codes                                       */
#define SQL_CHAR                                    1
#define SQL_NUMERIC                                 2
#define SQL_DECIMAL                                 3
#define SQL_INTEGER                                 4
#define SQL_SMALLINT                                5
#define SQL_FLOAT                                   6
#define SQL_REAL                                    7
#define SQL_DOUBLE                                  8
#define SQL_DATETIME                                9
#define SQL_INTERVAL                               10
#define SQL_VARCHAR                                12
#define SQL_BOOLEAN                                16
#define SQL_UDT                                    17
#define SQL_UDT_LOCATOR                            18
#define SQL_ROW                                    19
#define SQL_REF                                    20
#define SQL_BIGINT                                 25
#define SQL_BLOB                                   30
#define SQL_BLOB_LOCATOR                           31
#define SQL_CLOB                                   40
#define SQL_CLOB_LOCATOR                           41
#define SQL_ARRAY                                  50
#define SQL_ARRAY_LOCATOR                          51
#define SQL_MULTISET                               55
#define SQL_MULTISET_LOCATOR                       56
#define SQL_BINARY                                 60
#define SQL_VARBINARY                              61
/*      Concise codes for datetime and interval data types   */
#define SQL_TYPE_DATE                              91
#define SQL_TYPE_TIME                              92
#define SQL_TYPE_TIME_WITH_TIMEZONE                94
#define SQL_TYPE_TIMESTAMP                         93
#define SQL_TYPE_TIMESTAMP_WITH_TIMEZONE           95
#define SQL_INTERVAL_DAY                          103
#define SQL_INTERVAL_DAY_TO_HOUR                  108
#define SQL_INTERVAL_DAY_TO_MINUTE                109
#define SQL_INTERVAL_DAY_TO_SECOND                110
#define SQL_INTERVAL_HOUR                         104
#define SQL_INTERVAL_HOUR_TO_MINUTE               111
#define SQL_INTERVAL_HOUR_TO_SECOND               112
#define SQL_INTERVAL_MINUTE                       105
#define SQL_INTERVAL_MINUTE_TO_SECOND             113
#define SQL_INTERVAL_MONTH                        102
#define SQL_INTERVAL_SECOND                       106
#define SQL_INTERVAL_YEAR                         101
#define SQL_INTERVAL_YEAR_TO_MONTH                107
/* User-defined data type codes                              */
#define SQL_DISTINCT                                1
#define SQL_STRUCTURED                              2
/*      GetTypeInfo() request for all data types             */
```

```
#define SQL_ALL_TYPES                                 0
/* BindCol() and BindParameter() default conversion code     */
#define SQL_DEFAULT                                  99
/*  GetData() and \() code indicating that the
    application parameter descriptor specifies the data type */
#define SQL_APD_TYPE                                -99
#define SQL_ARD_TYPE                                -99
/* Date/time type subcodes                                   */
#define SQL_CODE_DATE                                 1
#define SQL_CODE_TIME                                 2
#define SQL_CODE_TIMESTAMP                            3
#define SQL_CODE_TIME_ZONE                            4
#define SQL_CODE_TIMESTAMP_ZONE                       5
/* Interval qualifier codes                                  */
#define SQL_DAY                                       3
#define SQL_DAY_TO_HOUR                               8
#define SQL_DAY_TO_MINUTE                             9
#define SQL_DAY_TO_SECOND                            10
#define SQL_HOUR                                      4
#define SQL_HOUR_TO_MINUTE                           11
#define SQL_HOUR_TO_SECOND                           12
#define SQL_MINUTE                                    5
#define SQL_MINUTE_TO_SECOND                         13
#define SQL_MONTH                                     2
#define SQL_SECOND                                    6
#define SQL_YEAR                                      1
#define SQL_YEAR_TO_MONTH                             7
/* CLI option values                                         */
#define SQL_FALSE                                     0
#define SQL_TRUE                                      1
#define SQL_NONSCROLLABLE                             0
#define SQL_SCROLLABLE                                1
#define SQL_NONHOLDABLE                               0
#define SQL_HOLDABLE                                  1
#define SQL_INITIALLY_DEFERRED                        5
#define SQL_INITIALLY_IMMEDIATE                       6
#define SQL_NOT_DEFERRABLE                            7
/* Parameter mode values                                     */
#define SQL_PARAM_MODE_IN                             1
#define SQL_PARAM_MODE_OUT                            4
#define SQL_PARAM_MODE_INOUT                          2
/* Codes used for FetchOrientation                           */
#define SQL_FETCH_NEXT                                1
#define SQL_FETCH_FIRST                               2
#define SQL_FETCH_LAST                                3
#define SQL_FETCH_PRIOR                               4
#define SQL_FETCH_ABSOLUTE                            5
#define SQL_FETCH_RELATIVE                            6
/* Values of NULLABLE field in descriptor                    */
#define SQL_NO_NULLS                                  0
#define SQL_NULLABLE                                  1
/* Values returned by GetTypeInfo for the SEARCHABLE column  */
#define SQL_PRED_NONE                                 0
#define SQL_PRED_CHAR                                 1
#define SQL_PRED_BASIC                                2
/* Values of UNNAMED field in descriptor                     */
#define SQL_NAMED                                     0
```

```
#define SQL_UNNAMED                                    1
/* Values of ALLOC_TYPE field in descriptor                     */
#define SQL_DESC_ALLOC_AUTO                            1
#define SQL_DESC_ALLOC_USER                            2
/* EndTran() options */
#define SQL_COMMIT                                     0
#define SQL_ROLLBACK                                   1
#define SQL_SAVEPOINT_NAME_ROLLBACK                    2
#define SQL_SAVEPOINT_NAME_RELEASE                     4
#define SQL_COMMIT_AND_CHAIN                           6
#define SQL_ROLLBACK_AND_CHAIN                         7
/* FreeStmt() options                                           */
#define SQL_CLOSE_CURSOR                               0
#define SQL_FREE_HANDLE                                1
#define SQL_UNBIND_COLUMNS                             2
#define SQL_UNBIND_PARAMETERS                          3
#define SQL_REALLOCATE                                 4
/* Provided for backwards compatibility                         */
#define SQL_CLOSE                                      0
#define SQL_DROP                                       1
#define SQL_UNBIND                                     2
#define SQL_RESET_PARAMS                               3
/* Null handle used when allocating HENV                        */
#define SQL_NULL_HANDLE                               0L
/* Null handles returned by AllocHandle()                       */
#define SQL_NULL_HENV                    SQL_NULL_HANDLE
#define SQL_NULL_HDBC                    SQL_NULL_HANDLE
#define SQL_NULL_HSTMT                   SQL_NULL_HANDLE
#define SQL_NULL_HDESC                   SQL_NULL_HANDLE
/*      GetFunctions values to identify CLI routines            */
#define SQL_API_SQLALLOCCONNECT                        1
#define SQL_API_SQLALLOCENV                            2
#define SQL_API_SQLALLOCHANDLE                      1001
#define SQL_API_SQLALLOCSTMT                           3
#define SQL_API_SQLBINDCOL                             4
#define SQL_API_SQLBINDPARAMETER                      72
#define SQL_API_SQLCANCEL                              5
#define SQL_API_SQLCLOSECURSOR                      1003
#define SQL_API_SQLCOLATTRIBUTE                        6
#define SQL_API_SQLCOLUMNPRIVILEGES                   56
#define SQL_API_SQLCOLUMNS                            40
#define SQL_API_SQLCONNECT                             7
#define SQL_API_SQLCOPYDESC                         1004
#define SQL_API_SQLDATASOURCES                        57
#define SQL_API_SQLDESCRIBECOL                         8
#define SQL_API_SQLDISCONNECT                          9
#define SQL_API_SQLENDTRAN                          1005
#define SQL_API_SQLERROR                              10
#define SQL_API_SQLEXECDIRECT                         11
#define SQL_API_SQLEXECUTE                            12
#define SQL_API_SQLFETCH                              13
#define SQL_API_SQLFETCHSCROLL                      1021
#define SQL_API_SQLFOREIGNKEYS                        60
#define SQL_API_SQLFREECONNECT                        14
#define SQL_API_SQLFREEENV                            15
#define SQL_API_SQLFREEHANDLE                       1006
#define SQL_API_SQLFREESTMT                           16
```

```
#define SQL_API_SQLGETCONNECTATTR            1007
#define SQL_API_SQLGETCURSORNAME               17
#define SQL_API_SQLGETDATA                     43
#define SQL_API_SQLGETDESCFIELD              1008
#define SQL_API_SQLGETDESCREC                1009
#define SQL_API_SQLGETDIAGFIELD              1010
#define SQL_API_SQLGETDIAGREC                1011
#define SQL_API_SQLGETENVATTR                1012
#define SQL_API_SQLGETFEATUREINFO            1027
#define SQL_API_SQLGETFUNCTIONS                44
#define SQL_API_SQLGETINFO                     45
#define SQL_API_SQLGETLENGTH                 1022
#define SQL_API_SQLGETPARAMDATA              1025
#define SQL_API_SQLGETPOSITION               1023
#define SQL_API_SQLGETSESSIONINFO            1028
#define SQL_API_SQLGETSTMTATTR               1014
#define SQL_API_SQLGETSUBSTRING              1024
#define SQL_API_SQLGETTYPEINFO                 47
#define SQL_API_SQLMORERESULTS                 61
#define SQL_API_SQLNEXTRESULT                  73
#define SQL_API_SQLNUMRESULTCOLS               18
#define SQL_API_SQLPARAMDATA                   48
#define SQL_API_SQLPREPARE                     19
#define SQL_API_SQLPRIMARYKEYS                 65
#define SQL_API_SQLPUTDATA                     49
#define SQL_API_SQLROWCOUNT                    20
#define SQL_API_SQLSETCONNECTATTR            1016
#define SQL_API_SQLSETCURSORNAME               21
#define SQL_API_SQLSETDESCFIELD              1017
#define SQL_API_SQLSETDESCREC                1018
#define SQL_API_SQLSETENVATTR                1019
#define SQL_API_SQLSETSTMTATTR               1020
#define SQL_API_SQLSPECIALCOLUMNS              52
#define SQL_API_SQLSTARTTRAN                   74
#define SQL_API_SQLTABLES                      54
#define SQL_API_SQLTABLEPRIVILEGES             70
/*      Information requested by GetInfo()              */
#define SQL_MAXIMUM_DRIVER_CONNECTIONS          0
#define SQL_MAXIMUM_CONCURRENT_ACTIVITIES       1
#define SQL_DATA_SOURCE_NAME                    2
#define SQL_FETCH_DIRECTION                     8
#define SQL_SERVER_NAME                        13
#define SQL_SEARCH_PATTERN_ESCAPE              14
#define SQL_DBMS_NAME                          17
#define SQL_DBMS_VERSION                       18
#define SQL_CURSOR_COMMIT_BEHAVIOR             23
#define SQL_DATA_SOURCE_READ_ONLY              25
#define SQL_DEFAULT_TRANSACTION_ISOLATION      26
#define SQL_IDENTIFIER_CASE                    28
#define SQL_MAXIMUM_COLUMN_NAME_LENGTH         30
#define SQL_MAXIMUM_CURSOR_NAME_LENGTH         31
#define SQL_MAXIMUM_SCHEMA_NAME_LENGTH         32
#define SQL_MAXIMUM_CATALOG_NAME_LENGTH        34
#define SQL_MAXIMUM_TABLE_NAME_LENGTH          35
#define SQL_SCROLL_CONCURRENCY                 43
#define SQL_TRANSACTION_CAPABLE                46
#define SQL_USER_NAME                          47
```

```
#define SQL_TRANSACTION_ISOLATION_OPTION          72
#define SQL_INTEGRITY                             73
#define SQL_GETDATA_EXTENSIONS                    81
#define SQL_NULL_COLLATION                        85
#define SQL_ALTER_TABLE                           86
#define SQL_ORDER_BY_COLUMNS_IN_SELECT            90
#define SQL_SPECIAL_CHARACTERS                    94
#define SQL_MAXIMUM_COLUMNS_IN_GROUP_BY           97
#define SQL_MAXIMUM_COLUMNS_IN_ORDER_BY           99
#define SQL_MAXIMUM_COLUMNS_IN_SELECT            100
#define SQL_MAXIMUM_COLUMNS_IN_TABLE             101
#define SQL_MAXIMUM_TABLES_IN_SELECT             106
#define SQL_MAXIMUM_USER_NAME_LENGTH             107
#define SQL_OUTER_JOIN_CAPABILITIES              115
#define SQL_CURSOR_SENSITIVITY                 10001
#define SQL_DESCRIBE_PARAMETER                 10002
#define SQL_CATALOG_NAME                       10003
#define SQL_COLLATING_SEQUENCE                 10004
#define SQL_MAXIMUM_IDENTIFIER_LENGTH          10005
#define SQL_MAXIMUM_STMT_OCTETS                20000
#define SQL_MAXIMUM_STMT_OCTETS_DATA           20001
#define SQL_MAXIMUM_STMT_OCTETS_SCHEMA         20002
/*      Information requested by GetSessionInfo()            */
#define SQL_CURRENT_USER                          47
#define SQL_CURRENT_DEFAULT_TRANSFORM_GROUP     20004
#define SQL_CURRENT_PATH                       20005
#define SQL_CURRENT_ROLE                       20006
#define SQL_SESSION_USER                       20007
#define SQL_SYSTEM_USER                        20008
#define SQL_CURRENT_CATALOG                    20009
#define SQL_CURRENT_SCHEMA                     20010
/* Statement attribute values for cursor sensitivity         */
#define SQL_ASENSITIVE                   0x00000000L
#define SQL_INSENSITIVE                  0x00000001L
#define SQL_SENSITIVE                    0x00000002L
/* Define SQL_UNSPECIFIED for backwards compatibiltiy         */
#define SQL_UNSPECIFIED               SQL_ASENSITIVE
/* SQL_ALTER_TABLE bitmasks                                  */
#define SQL_AT_ADD_COLUMN                0x00000001L
#define SQL_AT_DROP_COLUMN               0x00000002L
#define SQL_AT_ALTER_COLUMN              0x00000004L
#define SQL_AT_ADD_CONSTRAINT            0x00000008L
#define SQL_AT_DROP_CONSTRAINT           0x00000010L
/* SQL_CURSOR_COMMIT_BEHAVIOR values                         */
#define SQL_CB_DELETE                              0
#define SQL_CB_CLOSE                               1
#define SQL_CB_PRESERVE                            2
/* SQL_FETCH_DIRECTION bitmasks                              */
#define SQL_FD_FETCH_NEXT                0x00000001L
#define SQL_FD_FETCH_FIRST               0x00000002L
#define SQL_FD_FETCH_LAST                0x00000004L
#define SQL_FD_FETCH_PRIOR               0x00000008L
#define SQL_FD_FETCH_ABSOLUTE            0x00000010L
#define SQL_FD_FETCH_RELATIVE            0x00000020L
/* SQL_GETDATA_EXTENSIONS bitmasks                           */
#define SQL_GD_ANY_COLUMN                0x00000001L
#define SQL_GD_ANY_ORDER                 0x00000002L
```

```
    /* SQL_IDENTIFIER_CASE values                                   */
    #define SQL_IC_UPPER                                  1
    #define SQL_IC_LOWER                                  2
    #define SQL_IC_SENSITIVE                              3
    #define SQL_IC_MIXED                                  4
    /* SQL_NULL_COLLATION values                                    */
    #define SQL_NC_HIGH                                   1
    #define SQL_NC_LOW                                    2
    /* SQL_OUTER_JOIN_CAPABILITIES bitmasks                         */
    #define SQL_OUTER_JOIN_LEFT                  0x00000001L
    #define SQL_OUTER_JOIN_RIGHT                 0x00000002L
    #define SQL_OUTER_JOIN_FULL                  0x00000004L
    #define SQL_OUTER_JOIN_NESTED                0x00000008L
    #define SQL_OUTER_JOIN_NOT_ORDERED           0x00000010L
    #define SQL_OUTER_JOIN_INNER                 0x00000020L
    #define SQL_OUTER_JOIN_ALL_COMPARISON_OPS 0x00000040L
    /* SQL_SCROLL_CONCURRENCY bitmasks                              */
    #define SQL_SCCO_READ_ONLY                   0x00000001L
    #define SQL_SCCO_LOCK                        0x00000002L
    #define SQL_SCCO_OPT_ROWVER                  0x00000004L
    #define SQL_SCCO_OPT_VALUES                  0x00000008L
    /* SQL_TRANSACTION_CAPABLE values                               */
    #define SQL_TC_NONE                                   0
    #define SQL_TC_DML                                    1
    #define SQL_TC_ALL                                    2
    #define SQL_TC_DDL_COMMIT                             3
    #define SQL_TC_DDL_IGNORE                             4
    /* SQL_TRANSACTION_ISOLATION bitmasks                           */
    #define SQL_TRANSACTION_READ_UNCOMMITTED  0x00000001L
    #define SQL_TRANSACTION_READ_COMMITTED    0x00000002L
    #define SQL_TRANSACTION_REPEATABLE_READ   0x00000004L
    #define SQL_TRANSACTION_SERIALIZABLE      0x00000008L
    /* SQL_TRANSACTION_ACCESS_MODE bitmasks                         */
    #define SQL_TRANSACTION_READ_ONLY         0x00000001L
    #define SQL_TRANSACTION_READ_WRITE        0x00000002L
/* Column types and scopes in SpecialColumns */
    #define SQL_BEST_ROWID                               1
    #define SQL_SCOPE_CURROW                             0
    #define SQL_SCOPE_TRANSACTION                        1
    #define SQL_SCOPE_SESSION                            2
    #define SQL_PC_UNKNOWN                               0
    #define SQL_PC_NOT_PSEUDO                            1
    #define SQL_PC_PSEUDO                                2
/* Foreign Key UPDATE and DELETE rules */
    #define SQL_CASCADE                                  0
    #define SQL_RESTRICT                                 1
    #define SQL_SET_NULL                                 2
    #define SQL_NO_ACTION                                3
    #define SQL_SET_DEFAULT                              4
    /* Special parameter values                                     */
    #define SQL_ALL_CATALOGS                           '%'
    #define SQL_ALL_SCHEMAS                            '%'
    #define SQL_ALL_TABLE_TYPES                        '%'
    /* Function prototypes                                          */
    SQLRETURN  SQLAllocConnect(SQLHENV EnvironmentHandle,
               SQLHDBC *ConnectionHandle);
    SQLRETURN  SQLAllocEnv(SQLHENV *EnvironmentHandle);
```

```
SQLRETURN  SQLAllocHandle(SQLSMALLINT HandleType,
           SQLINTEGER InputHandle, SQLINTEGER *OutputHandle);
SQLRETURN  SQLAllocStmt(SQLHDBC ConnectionHandle,
           SQLHSTMT *StatementHandle);
SQLRETURN  SQLBindCol(SQLHSTMT StatementHandle,
           SQLSMALLINT ColumnNumber, SQLSMALLINT BufferType,
           SQLPOINTER Data, SQLINTEGER BufferLength,
           SQLINTEGER *StrLen_or_Ind);
SQLRETURN  SQLBindParameter(SQLHSTMT StatementHandle,
           SQLSMALLINT ParamNumber, SQLSMALLINT InputOutputMode,
           SQLSMALLINT ValueType, SQLSMALLINT ParameterType,
           SQLINTEGER ColumnSize, SQLSMALLINT DecimalDigits,
           SQLPOINTER ParameterValue, SQLINTEGER BufferLength,
           SQLINTEGER *StrLen_or_Ind);
SQLRETURN  SQLCancel(SQLHSTMT StatementHandle);
SQLRETURN  SQLCloseCursor(SQLHSTMT StatementHandle);
SQLRETURN  SQLColAttribute(SQLHSTMT StatementHandle,
           SQLSMALLINT ColumnNumber, SQLSMALLINT FieldIdentifier,
           SQLCHAR *CharacterAttribute, SQLSMALLINT BufferLength,
           SQLSMALLINT *StringLength, SQLINTEGER *NumericAttribute);
SQLRETURN  SQLColumnPrivileges(SQLHSTMT StatementHandle,
           SQLCHAR *CatalogName, SQLSMALLINT NameLength1,
           SQLCHAR *SchemaName, SQLSMALLINT NameLength2,
           SQLCHAR *TableName, SQLSMALLINT NameLength3,
           SQLCHAR *ColumnName, SQLSMALLINT NameLength4 );
SQLRETURN  SQLColumns(SQLHSTMT StatementHandle,
           SQLCHAR *CatalogName, SQLSMALLINT NameLength1,
           SQLCHAR *SchemaName, SQLSMALLINT NameLength2,
           SQLCHAR *TableName, SQLSMALLINT NameLength3,
           SQLCHAR *ColumnName, SQLSMALLINT NameLength4 );
SQLRETURN  SQLConnect(SQLHDBC ConnectionHandle,
           SQLCHAR *ServerName, SQLSMALLINT NameLength1,
           SQLCHAR *UserName, SQLSMALLINT NameLength2,
           SQLCHAR *Authentication, SQLSMALLINT NameLength3);
SQLRETURN  SQLCopyDesc(SQLHDESC SourceDescHandle,
           SQLHDESC TargetDescHandle);
SQLRETURN  SQLDataSources(SQLHENV EnvironmentHandle,
           SQLSMALLINT Direction, SQLCHAR *ServerName,
           SQLSMALLINT BufferLength1, SQLSMALLINT *NameLength1,
           SQLCHAR *Description, SQLSMALLINT BufferLength2,
           SQLSMALLINT *NameLength2);
SQLRETURN  SQLDescribeCol(SQLHSTMT StatementHandle,
           SQLSMALLINT ColumnNumber, SQLCHAR *ColumnName,
           SQLSMALLINT BufferLength, SQLSMALLINT *NameLength,
           SQLSMALLINT *DataType, SQLINTEGER *ColumnSize,
           SQLSMALLINT *DecimalDigits, SQLSMALLINT *Nullable);
SQLRETURN  SQLDisconnect(SQLHDBC ConnectionHandle);
SQLRETURN  SQLEndTran(SQLSMALLINT HandleType, SQLINTEGER Handle,
           SQLSMALLINT CompletionType);
SQLRETURN  SQLError(SQLHENV EnvironmentHandle,
           SQLHDBC ConnectionHandle, SQLHSTMT StatementHandle,
           SQLCHAR *Sqlstate, SQLINTEGER *NativeError,
           SQLCHAR *MessageText, SQLSMALLINT BufferLength,
           SQLSMALLINT *TextLength);
SQLRETURN  SQLExecDirect(SQLHSTMT StatementHandle,
           SQLCHAR *StatementText, SQLINTEGER TextLength);
SQLRETURN  SQLExecute(SQLHSTMT StatementHandle);
```

```
SQLRETURN  SQLFetch(SQLHSTMT StatementHandle);
SQLRETURN  SQLFetchScroll(SQLHSTMT StatementHandle,
           SQLSMALLINT FetchOrientation, SQLINTEGER FetchOffset);
SQLRETURN  SQLForeignKeys(SQLHSTMT StatementHandle,
           SQLCHAR *PKCatalogName, SQLSMALLINT NameLength1,
           SQLCHAR *PKSchemaName, SQLSMALLINT NameLength2,
           SQLCHAR *PKTableName, SQLSMALLINT NameLength3,
           SQLCHAR *FKCatalogName, SQLSMALLINT NameLength4,
           SQLCHAR *FKSchemaName, SQLSMALLINT NameLength5,
           SQLCHAR *FKTableName, SQLSMALLINT NameLength6);
SQLRETURN  SQLFreeConnect(SQLHDBC ConnectionHandle);
SQLRETURN  SQLFreeEnv(SQLHENV EnvironmentHandle);
SQLRETURN  SQLFreeHandle(SQLSMALLINT HandleType,
           SQLINTEGER Handle);
SQLRETURN  SQLFreeStmt(SQLHSTMT StatementHandle, SQLSMALLINT Option);
SQLRETURN  SQLGetConnectAttr(SQLHDBC ConnectionHandle,
           SQLINTEGER Attribute, SQLPOINTER Value,
           SQLINTEGER BufferLength, SQLINTEGER *StringLength);
SQLRETURN  SQLGetCursorName(SQLHSTMT StatementHandle,
           SQLCHAR *CursorName, SQLSMALLINT BufferLength,
           SQLSMALLINT *NameLength);
SQLRETURN  SQLGetData(SQLHSTMT StatementHandle,
           SQLSMALLINT ColumnNumber, SQLSMALLINT TargetType,
           SQLPOINTER TargetValue, SQLINTEGER BufferLength,
           SQLINTEGER *StrLen_or_Ind);
SQLRETURN  SQLGetDescField(SQLHDESC DescriptorHandle,
           SQLSMALLINT RecordNumber, SQLSMALLINT FieldIdentifier,
           SQLPOINTER Value, SQLINTEGER BufferLength,
           SQLINTEGER *StringLength);
SQLRETURN  SQLGetDescRec(SQLHDESC DescriptorHandle,
           SQLSMALLINT RecordNumber, SQLCHAR *Name,
           SQLSMALLINT BufferLength, SQLSMALLINT *NameLength,
           SQLSMALLINT *Type, SQLSMALLINT *SubType,
           SQLINTEGER *Length, SQLSMALLINT *Precision,
           SQLSMALLINT *Scale, SQLSMALLINT *Nullable);
SQLRETURN  SQLGetDiagField(SQLSMALLINT HandleType,
           SQLINTEGER Handle, SQLSMALLINT RecordNumber,
           SQLSMALLINT DiagIdentifier, SQLPOINTER DiagInfo,
           SQLSMALLINT BufferLength, SQLSMALLINT *StringLength);
SQLRETURN  SQLGetDiagRec(SQLSMALLINT HandleType, SQLINTEGER Handle,
           SQLSMALLINT RecordNumber, SQLCHAR *Sqlstate,
           SQLINTEGER *NativeError, SQLCHAR *MessageText,
           SQLSMALLINT BufferLength, SQLSMALLINT *TextLength);
SQLRETURN  SQLGetEnvAttr(SQLHENV EnvironmentHandle,
           SQLINTEGER Attribute, SQLPOINTER Value,
           SQLINTEGER BufferLength, SQLINTEGER *StringLength);
SQLRETURN  SQLGetFeatureInfo(SQLHDBC ConnectionHandle,
           SQLCHAR *FeatureType, SQLSMALLINT FeatureTypeLength,
           SQLCHAR *FeatureId, SQLSMALLINT FeatureIdLength,
           SQLCHAR *SubFeatureId, SQLSMALLINT SubFeatureIdLength,
           SQLSMALLINT *Supported);
SQLRETURN  SQLGetFunctions(SQLHDBC ConnectionHandle,
           SQLSMALLINT FunctionId, SQLSMALLINT *Supported);
SQLRETURN  SQLGetInfo(SQLHDBC ConnectionHandle,
           SQLSMALLINT InfoType, SQLPOINTER InfoValue,
           SQLSMALLINT BufferLength, SQLSMALLINT *StringLength);
SQLRETURN  SQLGetLength(SQLHSTMT StatementHandle,
```

```
            SQLSMALLINT LocatorType, SQLINTEGER Locator,
            SQLINTEGER *StringLength, SQLINTEGER *IndicatorValue);
SQLRETURN   SQLGetParamData(SQLHSTMT StatementHandle,
            SQLSMALLINT ParameterNumber, SQLSMALLINT TargetType,
            SQLPOINTER TargetValue, SQLINTEGER BufferLength,
            SQLINTEGER *StrLen_or_Ind);
SQLRETURN SQLGetPosition(SQLHSTMT StatementHandle,
            SQLSMALLINT LocatorType, SQLINTEGER SourceLocator,
            SQLINTEGER SearchLocator, SQLCHAR *SearchLiteral,
            SQLINTEGER SearchLiteralLength, SQLINTEGER FromPosition,
            SQLINTEGER *LocatedAt, SQLINTEGER *IndicatorValue);
SQLRETURN   SQLGetSessionInfo(SQLHDBC ConnectionHandle,
            SQLSMALLINT InfoType, SQLPOINTER InfoValue,
            SQLSMALLINT BufferLength, SQLSMALLINT *StringLength);
SQLRETURN   SQLGetStmtAttr(SQLHSTMT StatementHandle,
            SQLINTEGER Attribute, SQLPOINTER Value,
            SQLINTEGER BufferLength, SQLINTEGER *StringLength);
SQLRETURN   SQLGetSubString(SQLHSTMT StatementHandle,
            SQLSMALLINT LocatorType, SQLINTEGER SourceLocator,
            SQLINTEGER FromPosition, SQLINTEGER ForLength,
            SQLSMALLINT TargetType, SQLPOINTER TargetValue,
            SQLINTEGER BufferLength, SQLINTEGER *StringLength,
            SQLINTEGER *IndicatorValue);
SQLRETURN   SQLGetTypeInfo(SQLHSTMT StatementHandle,
            SQLSMALLINT DataType);
SQLRETURN   SQLMoreResults(SQLHSTMT StatementHandle);
SQLRETURN   SQLNextResult(SQLHSTMT StatementHandle1,
            SQLHSTMT *StatementHandle2);
SQLRETURN   SQLNumResultCols(SQLHSTMT StatementHandle,
            SQLSMALLINT *ColumnCount);
SQLRETURN   SQLParamData(SQLHSTMT StatementHandle,
            SQLPOINTER *Value);
SQLRETURN   SQLPrepare(SQLHSTMT StatementHandle,
            SQLCHAR *StatementText, SQLINTEGER TextLength);
SQLRETURN   SQLPrimaryKeys(SQLHSTMT StatementHandle,
            SQLCHAR *CatalogName, SQLSMALLINT NameLength1,
            SQLCHAR *SchemaName, SQLSMALLINT NameLength2,
            SQLCHAR *TableName, SQLSMALLINT NameLength3);
SQLRETURN   SQLPutData(SQLHSTMT StatementHandle,
            SQLPOINTER Data, SQLINTEGER StrLen_or_Ind);
SQLRETURN   SQLRowCount(SQLHSTMT StatementHandle,
            SQLINTEGER *RowCount);
SQLRETURN   SQLSetConnectAttr(SQLHDBC ConnectionHandle,
            SQLINTEGER Attribute, SQLPOINTER Value,
            SQLINTEGER StringLength);
SQLRETURN   SQLSetCursorName(SQLHSTMT StatementHandle,
            SQLCHAR *CursorName, SQLSMALLINT NameLength);
SQLRETURN   SQLSetDescField(SQLHDESC DescriptorHandle,
            SQLSMALLINT RecordNumber, SQLSMALLINT FieldIdentifier,
            SQLPOINTER Value, SQLINTEGER BufferLength);
SQLRETURN   SQLSetDescRec(SQLHDESC DescriptorHandle,
            SQLSMALLINT RecordNumber, SQLSMALLINT Type,
            SQLSMALLINT SubType, SQLINTEGER Length,
            SQLSMALLINT Precision, SQLSMALLINT Scale,
            SQLPOINTER Data, SQLINTEGER *StringLength,
            SQLINTEGER *Indicator);
SQLRETURN   SQLSetEnvAttr(SQLHENV EnvironmentHandle,
```

```
              SQLINTEGER Attribute, SQLPOINTER Value,
              SQLINTEGER StringLength);
SQLRETURN  SQLSetStmtAttr(SQLHSTMT StatementHandle,
              SQLINTEGER Attribute, SQLPOINTER Value,
              SQLINTEGER StringLength);
SQLRETURN SQLSpecialColumns(SQLHSTMT StatementHandle,
              SQLSMALLINT IdentifierType, SQLCHAR *CatalogName,
              SQLSMALLINT NameLength1, SQLCHAR *SchemaName,
              SQLSMALLINT NameLength2, SQLCHAR *TableName,
              SQLSMALLINT NameLength3, SQLSMALLINT Scope,
              SQLSMALLINT Nullable);
SQLRETURN SQLStartTran(SQLSMALLINT HandleType,
              SQLINTEGER Handle, SQLINTEGER AccessMode,
              SQLINTEGER IsolationLevel);
SQLRETURN SQLTablePrivileges(SQLHSTMT StatementHandle,
              SQLCHAR *CatalogName, SQLSMALLINT NameLength1,
              SQLCHAR *SchemaName, SQLSMALLINT NameLength2,
              SQLCHAR *TableName, SQLSMALLINT NameLength3);
SQLRETURN  SQLTables(SQLHSTMT StatementHandle,
              SQLCHAR *CatalogName, SQLSMALLINT NameLength1,
              SQLCHAR *SchemaName, SQLSMALLINT NameLength2,
              SQLCHAR *TableName, SQLSMALLINT NameLength3,
              SQLCHAR *TableType, SQLSMALLINT NameLength4 );
```

A.2 COBOL library item SQLCLI

This Subclause is modified by Subclause E.2, “COBOL Library Item SQLCLI”, in ISO/IEC 9075-9.

Here is a typical SQLCLI COBOL Library Item. COBOL applications include this library item into the Working-Storage Section by containing the following statement:

```
COPY SQLCLI.
```

The following file does not include prototypes of the SQL/CLI functions because COBOL applications are not required to specify them.

The following file has been coded with example COBOL syntax. When this file is used with a conforming CLI implementation, each ocurrence of “`PIC S9(4) BINARY`” should be replaced with the appropriate COBOL data type for SMALLINT from Table 42, “SQL/CLI data type correspondences for COBOL”, and each occurrence of “`PIC S9(9) BINARY`” should be replaced with the appropriate COBOL data type for INTEGER from Table 42, “SQL/CLI data type correspondences for COBOL”.

```
* SPECIAL LENGTH/INDICATOR VALUES
01 SQL-NULL-DATA                         PIC S9(9) BINARY VALUE IS    -1.
01 SQL-DATA-AT-EXEC                      PIC S9(9) BINARY VALUE IS    -2.
* RETURN VALUES FROM FUNCTIONS
01 SQL-SUCCESS                           PIC S9(4) BINARY VALUE IS     0.
01 SQL-SUCCESS-WITH-INFO                 PIC S9(4) BINARY VALUE IS     1.
01 SQL-NEED-DATA                         PIC S9(4) BINARY VALUE IS    99.
01 SQL-NO-DATA                           PIC S9(4) BINARY VALUE IS   100.
01 SQL-ERROR                             PIC S9(4) BINARY VALUE IS    -1.
01 SQL-INVALID-HANDLE                    PIC S9(4) BINARY VALUE IS    -2.
* ROW STATUS VALUES AFTER A CALL TO A FETCH FUNCTION
```

```
01 SQL-ROW-SUCCESS                      PIC S9(4) BINARY VALUE IS     0.
01 SQL-ROW-SUCCESS-WITH-INFO            PIC S9(4) BINARY VALUE IS     6.
01 SQL-ROW-ERROR                        PIC S9(4) BINARY VALUE IS     5.
01 SQL-ROW-NO-ROW                       PIC S9(4) BINARY VALUE IS     3.
* FLAGS FOR NULL-TERMINATED STRING
01 SQL-NTS                              PIC S9(4) BINARY VALUE IS    -3.
01 SQL-NTSL                             PIC S9(9) BINARY VALUE IS    -3.
* MAXIMUM MESSAGE LENGTH
01 SQL-MAXIMUM-MESSAGE-LENGTH           PIC S9(4) BINARY VALUE IS   512.
* ENVIRONMENT ATTRIBUTE
01 SQL-ATTR-OUTPUT-NTS                  PIC S9(9) BINARY VALUE IS 10001.
* CONNECTION ATTRIBUTE
01 SQL-ATTR-AUTO-IPD                    PIC S9(9) BINARY VALUE IS 10001.
01 SQL-ATTR-SAVEPOINT-NAME              PIC S9(9) BINARY VALUE IS 10027.
* HANDLE TYPE IDENTIFIERS
01 SQL-HANDLE-ENV                       PIC S9(4) BINARY VALUE IS     1.
01 SQL-HANDLE-DBC                       PIC S9(4) BINARY VALUE IS     2.
01 SQL-HANDLE-STMT                      PIC S9(4) BINARY VALUE IS     3.
01 SQL-HANDLE-DESC                      PIC S9(4) BINARY VALUE IS     4.
* STATEMENT ATTRIBUTES
01 SQL-ATTR-CURSOR-SCROLLABLE           PIC S9(9) BINARY VALUE IS    -1.
01 SQL-ATTR-CURSOR-SENSITIVITY          PIC S9(9) BINARY VALUE IS    -2.
01 SQL-ATTR-CURSOR-HOLDABLE             PIC S9(9) BINARY VALUE IS    -3.
01 SQL-ATTR-APP-ROW-DESC                PIC S9(9) BINARY VALUE IS 10010.
01 SQL-ATTR-APP-PARAM-DESC              PIC S9(9) BINARY VALUE IS 10011.
01 SQL-ATTR-IMP-ROW-DESC                PIC S9(9) BINARY VALUE IS 10012.
01 SQL-ATTR-IMP-PARAM-DESC              PIC S9(9) BINARY VALUE IS 10013.
01 SQL-ATTR-METADATA-ID                 PIC S9(9) BINARY VALUE IS 10014.
01 SQL-ATTR-CURRENT-OF-POSITION         PIC S9(9) BINARY VALUE IS 10027.
01 SQL-ATTR-NEST-DESCRIPTOR             PIC S9(9) BINARY VALUE IS 10029.
* IDENTIFIERS OF FIELDS IN THE SQL/CLI ITEM DESCRIPTOR AREA
01 SQL-DESC-ARRAY-SIZE                  PIC S9(4) BINARY VALUE IS    20.
01 SQL-DESC-ARRAY-STATUS-POINTER        PIC S9(4) BINARY VALUE IS    21.
01 SQL-DESC-DATETIME-INTERVAL-PRECISION PIC S9(4) BINARY VALUE IS    26.
01 SQL-DESC-ROWS-PROCESSED-POINTER      PIC S9(4) BINARY VALUE IS    34.
01 SQL-DESC-COUNT                       PIC S9(4) BINARY VALUE IS  1001.
01 SQL-DESC-TYPE                        PIC S9(4) BINARY VALUE IS  1002.
01 SQL-DESC-LENGTH                      PIC S9(4) BINARY VALUE IS  1003.
01 SQL-DESC-OCTET-LENGTH-POINTER        PIC S9(4) BINARY VALUE IS  1004.
01 SQL-DESC-PRECISION                   PIC S9(4) BINARY VALUE IS  1005.
01 SQL-DESC-SCALE                       PIC S9(4) BINARY VALUE IS  1006.
01 SQL-DESC-DATETIME-INTERVAL-CODE      PIC S9(4) BINARY VALUE IS  1007.
01 SQL-DESC-NULLABLE                    PIC S9(4) BINARY VALUE IS  1008.
01 SQL-DESC-INDICATOR-POINTER           PIC S9(4) BINARY VALUE IS  1009.
01 SQL-DESC-DATA-POINTER                PIC S9(4) BINARY VALUE IS  1010.
01 SQL-DESC-NAME                        PIC S9(4) BINARY VALUE IS  1011.
01 SQL-DESC-UNNAMED                     PIC S9(4) BINARY VALUE IS  1012.
01 SQL-DESC-OCTET-LENGTH                PIC S9(4) BINARY VALUE IS  1013.
01 SQL-DESC-COLLATION-CATALOG           PIC S9(4) BINARY VALUE IS  1015.
01 SQL-DESC-COLLLATION-SCHEMA           PIC S9(4) BINARY VALUE IS  1016.
01 SQL-DESC-COLLATION-NAME              PIC S9(4) BINARY VALUE IS  1017.
01 SQL-DESC-CHARACTER-SET-CATALOG       PIC S9(4) BINARY VALUE IS  1018.
01 SQL-DESC-CHARACTER-SET-SCHEMA        PIC S9(4) BINARY VALUE IS  1019.
01 SQL-DESC-CHARACTER-SET-NAME          PIC S9(4) BINARY VALUE IS  1020.
01 SQL-DESC-PARAMETER-MODE              PIC S9(4) BINARY VALUE IS  1021.
01 SQL-DESC-PARAMETER-ORDINAL-POSITION  PIC S9(4) BINARY VALUE IS  1022.
01 SQL-DESC-PARAMETER-SPECIFIC-CATALOG  PIC S9(4) BINARY VALUE IS  1023.
```

```
01 SQL-DESC-PARAMETER-SPECIFIC-SCHEMA    PIC S9(4) BINARY VALUE IS  1024.
01 SQL-DESC-PARAMETER-SPECIFIC-NAME      PIC S9(4) BINARY VALUE IS  1025.
01 SQL-DESC-UDT-CATALOG                  PIC S9(4) BINARY VALUE IS  1026.
01 SQL-DESC-UDT-SCHEMA                   PIC S9(4) BINARY VALUE IS  1027.
01 SQL-DESC-UDT-NAME                     PIC S9(4) BINARY VALUE IS  1028.
01 SQL-DESC-KEY-TYPE                     PIC S9(4) BINARY VALUE IS  1029.
01 SQL-DESC-KEY-MEMBER                   PIC S9(4) BINARY VALUE IS  1030.
01 SQL-DESC-DYNAMIC-FUNCTION             PIC S9(4) BINARY VALUE IS  1031.
01 SQL-DESC-DYNAMIC-FUNCTION-CODE        PIC S9(4) BINARY VALUE IS  1032.
01 SQL-DESC-SCOPE-CATALOG                PIC S9(4) BINARY VALUE IS  1033.
01 SQL-DESC-SCOPE-SCHEMA                 PIC S9(4) BINARY VALUE IS  1034.
01 SQL-DESC-SCOPE-NAME                   PIC S9(4) BINARY VALUE IS  1035.
01 SQL-DESC-SPECIFIC-SCOPE-CATALOG       PIC S9(4) BINARY VALUE IS  1036.
01 SQL-DESC-SPECIFIC-SCOPE-SCHEMA        PIC S9(4) BINARY VALUE IS  1037.
01 SQL-DESC-SPECIFIC-SCOPE-NAME          PIC S9(4) BINARY VALUE IS  1038.
01 SQL-DESC-CURRENT-TRANSFORM-GROUP      PIC S9(4) BINARY VALUE IS  1039.
01 SQL-DESC-CARDINALITY                  PIC S9(4) BINARY VALUE IS  1040.
01 SQL-DESC-DEGREE                       PIC S9(4) BINARY VALUE IS  1041.
01 SQL-DESC-LEVEL                        PIC S9(4) BINARY VALUE IS  1042.
01 SQL-DESC-RETURNED-CARDINALITY-POINTER PIC S9(4) BINARY VALUE IS  1043.
01 SQL-DESC-TOP-LEVEL-COUNT              PIC S9(4) BINARY VALUE IS  1044.
01 SQL-DESC-USER-DEFINED-TYPE-CODE       PIC S9(4) BINARY VALUE IS  1045.
01 SQL-DESC-ALLOC-TYPE                   PIC S9(4) BINARY VALUE IS  1099.
* IDENTIFIERS OF FIELDS IN THE DIAGNOSTICS AREA
01 SQL-DIAG-ROW-NUMBER                   PIC S9(4) BINARY VALUE IS -1248.
01 SQL-DIAG-COLUMN-NUMBER                PIC S9(4) BINARY VALUE IS -1247.
01 SQL-DIAG-RETURNCODE                   PIC S9(4) BINARY VALUE IS     1.
01 SQL-DIAG-NUMBER                       PIC S9(4) BINARY VALUE IS     2.
01 SQL-DIAG-ROW-COUNT                    PIC S9(4) BINARY VALUE IS     3.
01 SQL-DIAG-SQLSTATE                     PIC S9(4) BINARY VALUE IS     4.
01 SQL-DIAG-NATIVE-CODE                  PIC S9(4) BINARY VALUE IS     5.
01 SQL-DIAG-MESSAGE-TEXT                 PIC S9(4) BINARY VALUE IS     6.
01 SQL-DIAG-DYNAMIC-FUNCTION             PIC S9(4) BINARY VALUE IS     7.
01 SQL-DIAG-CLASS-ORIGIN                 PIC S9(4) BINARY VALUE IS     8.
01 SQL-DIAG-SUBCLASS-ORIGIN              PIC S9(4) BINARY VALUE IS     9.
01 SQL-DIAG-CONNECTION-NAME              PIC S9(4) BINARY VALUE IS    10.
01 SQL-DIAG-SERVER-NAME                  PIC S9(4) BINARY VALUE IS    11.
01 SQL-DIAG-DYNAMIC-FUNCTION-CODE        PIC S9(4) BINARY VALUE IS    12.
01 SQL-DIAG-MORE                         PIC S9(4) BINARY VALUE IS    13.
01 SQL-DIAG-CONDITION-NUMBER             PIC S9(4) BINARY VALUE IS    14.
01 SQL-DIAG-CONSTRAINT-CATALOG           PIC S9(4) BINARY VALUE IS    15.
01 SQL-DIAG-CONSTRAINT-SCHEMA            PIC S9(4) BINARY VALUE IS    16.
01 SQL-DIAG-CONSTRAINT-NAME              PIC S9(4) BINARY VALUE IS    17.
01 SQL-DIAG-CATALOG-NAME                 PIC S9(4) BINARY VALUE IS    18.
01 SQL-DIAG-SCHEMA-NAME                  PIC S9(4) BINARY VALUE IS    19.
01 SQL-DIAG-TABLE-NAME                   PIC S9(4) BINARY VALUE IS    20.
01 SQL-DIAG-COLUMN-NAME                  PIC S9(4) BINARY VALUE IS    21.
01 SQL-DIAG-CURSOR-NAME                  PIC S9(4) BINARY VALUE IS    22.
01 SQL-DIAG-MESSAGE-LENGTH               PIC S9(4) BINARY VALUE IS    23.
01 SQL-DIAG-MESSAGE-OCTET-LENGTH         PIC S9(4) BINARY VALUE IS    24.
01 SQL-DIAG-CONDITION-IDENTIFIER         PIC S9(4) BINARY VALUE IS    25.
01 SQL-DIAG-PARAMETER-NAME               PIC S9(4) BINARY VALUE IS    26.
01 SQL-DIAG-ROUTINE-CATALOG              PIC S9(4) BINARY VALUE IS    27.
01 SQL-DIAG-ROUTINE_SCHEMA               PIC S9(4) BINARY VALUE IS    28.
01 SQL-DIAG-ROUTINE-NAME                 PIC S9(4) BINARY VALUE IS    29.
01 SQL-DIAG-SPECIFIC-NAME                PIC S9(4) BINARY VALUE IS    30.
01 SQL-DIAG-TRIGGER-CATALOG              PIC S9(4) BINARY VALUE IS    31.
```

```
01 SQL-DIAG-TRIGGER-SCHEMA              PIC S9(4) BINARY VALUE IS    32.
01 SQL-DIAG-TRIGGER-NAME                PIC S9(4) BINARY VALUE IS    33.
01 SQL-DIAG-TRANSACTIONS-COMMITTED      PIC S9(4) BINARY VALUE IS    34.
01 SQL-DIAG-TRANSACTIONS-ROLLED-BACK    PIC S9(4) BINARY VALUE IS    35.
01 SQL-DIAG-TRANSACTION-ACTIVE          PIC S9(4) BINARY VALUE IS    36.
01 SQL-DIAG-PARAMETER-MODE              PIC S9(4) BINARY VALUE IS    37.
01 SQL-DIAG-PARAMETER-ORDINAL-POSITION  PIC S9(4) BINARY VALUE IS    38.
* DYNAMIC FUNCTION CODES RETURNED IN DIAGNOSTICS AREA
01 SQL-DIAG-ALTER-DOMAIN                PIC S9(9) BINARY VALUE IS     3.
01 SQL-DIAG-ALTER-TABLE                 PIC S9(9) BINARY VALUE IS     4.
01 SQL-DIAG-CALL                        PIC S9(9) BINARY VALUE IS     7.
01 SQL-DIAG-CREATE-ASSERTION            PIC S9(9) BINARY VALUE IS     6.
01 SQL-DIAG-CREATE-CHARACTER-SET        PIC S9(9) BINARY VALUE IS     8.
01 SQL-DIAG-CREATE-COLLATION            PIC S9(9) BINARY VALUE IS    10.
01 SQL-DIAG-CREATE-DOMAIN               PIC S9(9) BINARY VALUE IS    23.
01 SQL-DIAG-CREATE-SCHEMA               PIC S9(9) BINARY VALUE IS    64.
01 SQL-DIAG-CREATE-TABLE                PIC S9(9) BINARY VALUE IS    77.
01 SQL-DIAG-CREATE-TRANSLATION          PIC S9(9) BINARY VALUE IS    79.
01 SQL-DIAG-CREATE-VIEW                 PIC S9(9) BINARY VALUE IS    84.
01 SQL-DIAG-DELETE-WHERE                PIC S9(9) BINARY VALUE IS    19.
01 SQL-DIAG-DROP-ASSERTION              PIC S9(9) BINARY VALUE IS    24.
01 SQL-DIAG-DROP-CHARACTER-SET          PIC S9(9) BINARY VALUE IS    25.
01 SQL-DIAG-DROP-COLLATION              PIC S9(9) BINARY VALUE IS    26.
01 SQL-DIAG-DROP-DOMAIN                 PIC S9(9) BINARY VALUE IS    27.
01 SQL-DIAG-DROP-SCHEMA                 PIC S9(9) BINARY VALUE IS    31.
01 SQL-DIAG-DROP-TABLE                  PIC S9(9) BINARY VALUE IS    32.
01 SQL-DIAG-DROP-TRANSLATION            PIC S9(9) BINARY VALUE IS    33.
01 SQL-DIAG-DROP-VIEW                   PIC S9(9) BINARY VALUE IS    36.
01 SQL-DIAG-DYNAMIC-DELETE-CURSOR       PIC S9(9) BINARY VALUE IS    54.
01 SQL-DIAG-DYNAMIC-UPDATE-CURSOR       PIC S9(9) BINARY VALUE IS    55.
01 SQL-DIAG-GRANT                       PIC S9(9) BINARY VALUE IS    48.
01 SQL-DIAG-INSERT                      PIC S9(9) BINARY VALUE IS    50.
01 SQL-DIAG-MERGE                       PIC S9(9) BINARY VALUE IS   128.
01 SQL-DIAG-REVOKE                      PIC S9(9) BINARY VALUE IS    59.
01 SQL-DIAG-SELECT                      PIC S9(9) BINARY VALUE IS    41.
01 SQL-DIAG-SELECT-CURSOR               PIC S9(9) BINARY VALUE IS    85.
01 SQL-DIAG-SET-CATALOG                 PIC S9(9) BINARY VALUE IS    66.
01 SQL-DIAG-SET-CONSTRAINT              PIC S9(9) BINARY VALUE IS    68.
01 SQL-DIAG-SET-NAMES                   PIC S9(9) BINARY VALUE IS    72.
01 SQL-DIAG-SET-SCHEMA                  PIC S9(9) BINARY VALUE IS    74.
01 SQL-DIAG-SET-SESSION-AUTHORIZATION   PIC S9(9) BINARY VALUE IS    76.
01 SQL-DIAG-SET-TIME-ZONE               PIC S9(9) BINARY VALUE IS    71.
01 SQL-DIAG-SET-TRANSACTION             PIC S9(9) BINARY VALUE IS    75.
01 SQL-DIAG-UNKNOWN-STATEMENT           PIC S9(9) BINARY VALUE IS     0.
01 SQL-DIAG-UPDATE-WHERE                PIC S9(9) BINARY VALUE IS    82.
* SQL DATA TYPE CODES
01 SQL-CHAR                             PIC S9(4) BINARY VALUE IS     1.
01 SQL-NUMERIC                          PIC S9(4) BINARY VALUE IS     2.
01 SQL-DECIMAL                          PIC S9(4) BINARY VALUE IS     3.
01 SQL-INTEGER                          PIC S9(4) BINARY VALUE IS     4.
01 SQL-SMALLINT                         PIC S9(4) BINARY VALUE IS     5.
01 SQL-FLOAT                            PIC S9(4) BINARY VALUE IS     6.
01 SQL-REAL                             PIC S9(4) BINARY VALUE IS     7.
01 SQL-DOUBLE                           PIC S9(4) BINARY VALUE IS     8.
01 SQL-DATETIME                         PIC S9(4) BINARY VALUE IS     9.
01 SQL-INTERVAL                         PIC S9(4) BINARY VALUE IS    10.
01 SQL-VARCHAR                          PIC S9(4) BINARY VALUE IS    12.
```

```
01 SQL-BOOLEAN                          PIC S9(4) BINARY VALUE IS    16.
01 SQL-UDT                              PIC S9(4) BINARY VALUE IS    17.
01 SQL-UDT-LOCATOR                      PIC S9(4) BINARY VALUE IS    18.
01 SQL-ROW                              PIC S9(4) BINARY VALUE IS    19.
01 SQL-REF                              PIC S9(4) BINARY VALUE IS    20.
01 SQL-BIGINT                           PIC S9(4) BINARY VALUE IS    25.
01 SQL-BLOB                             PIC S9(4) BINARY VALUE IS    30.
01 SQL-BLOB-LOCATOR                     PIC S9(4) BINARY VALUE IS    31.
01 SQL-CLOB                             PIC S9(4) BINARY VALUE IS    40.
01 SQL-CLOB-LOCATOR                     PIC S9(4) BINARY VALUE IS    41.
01 SQL-ARRAY                            PIC S9(4) BINARY VALUE IS    50.
01 SQL-ARRAY-LOCATOR                    PIC S9(4) BINARY VALUE IS    51.
01 SQL-MULTISET                         PIC S9(4) BINARY VALUE IS    55.
01 SQL-MULTISET-LOCATOR                 PIC S9(4) BINARY VALUE IS    56.
01 SQL-BINARY                           PIC S9(4) BINARY VALUE IS    60.
01 SQL-VARBINARY                        PIC S9(4) BINARY VALUE IS    61.
*  CONCISE CODES FOR DATETIME AND INTERVAL DATA TYPES
01 SQL-TYPE-DATE                        PIC S9(4) BINARY VALUE IS    91.
01 SQL-TYPE-TIME                        PIC S9(4) BINARY VALUE IS    92.
01 SQL-TYPE-TIME-WITH-TIMEZONE          PIC S9(4) BINARY VALUE IS    94.
01 SQL-TYPE-TIMESTAMP                   PIC S9(4) BINARY VALUE IS    93.
01 SQL-TYPE-TIMESTAMP-WITH-TIMEZONE     PIC S9(4) BINARY VALUE IS    95.
01 SQL-INTERVAL-DAY                     PIC S9(4) BINARY VALUE IS   103.
01 SQL-INTERVAL-DAY-TO-HOUR             PIC S9(4) BINARY VALUE IS   108.
01 SQL-INTERVAL-DAY-TO-MINUTE           PIC S9(4) BINARY VALUE IS   109.
01 SQL-INTERVAL-DAY-TO-SECOND           PIC S9(4) BINARY VALUE IS   112.
01 SQL-INTERVAL-HOUR                    PIC S9(4) BINARY VALUE IS   104.
01 SQL-INTERVAL-HOUR-TO-MINUTE          PIC S9(4) BINARY VALUE IS   111.
01 SQL-INTERVAL-HOUR-TO-SECOND          PIC S9(4) BINARY VALUE IS   112.
01 SQL-INTERVAL-MINUTE                  PIC S9(4) BINARY VALUE IS   105.
01 SQL-INTERVAL-MINUTE-TO-SECOND        PIC S9(4) BINARY VALUE IS   113.
01 SQL-INTERVAL-MONTH                   PIC S9(4) BINARY VALUE IS   102.
01 SQL-INTERVAL-SECOND                  PIC S9(4) BINARY VALUE IS   106.
01 SQL-INTERVAL-YEAR                    PIC S9(4) BINARY VALUE IS   101.
01 SQL-INTERVAL-YEAR-TO-MONTH           PIC S9(4) BINARY VALUE IS   107.
* USER-DEFINED DATA TYPE CODES
01 SQL-DISTINCT                         PIC S9(4) BINARY VALUE IS     1.
01 SQL-STRUCTURED                       PIC S9(4) BINARY VALUE IS     2.
* SQLRGETTYPEINFO REQUEST FOR ALL DATA TYPES
01 SQL-ALL-TYPES                        PIC S9(4) BINARY VALUE IS     0.
*  SQLRBINDCOL AND SQLRBINDPARAMETER DEFAULT CONVERSION CODE
01 SQL-DEFAULT                          PIC S9(4) BINARY VALUE IS    99.
* SQLRGETDATA AND GETPARAMDATA CODES INDICATING THAT THE APPLICATION
* DESCRIPTOR SPECIFIES THE DATA TYPE
01 SQL-APD-TYPE                         PIC S9(4) BINARY VALUE IS   -99.
01 SQL-ARD-TYPE                         PIC S9(4) BINARY VALUE IS   -99.
* DATE/TIME TYPE SUBCODES
01 SQL-CODE-DATE                        PIC S9(4) BINARY VALUE IS     1.
01 SQL-CODE-TIME                        PIC S9(4) BINARY VALUE IS     2.
01 SQL-CODE-TIMESTAMP                   PIC S9(4) BINARY VALUE IS     3.
01 SQL-CODE-TIME-ZONE                   PIC S9(4) BINARY VALUE IS     4.
01 SQL-CODE-TIMESTAMP-ZONE              PIC S9(4) BINARY VALUE IS     5.
* INTERVAL QUALIFIER CODES
01 SQL-DAY                              PIC S9(4) BINARY VALUE IS     3.
01 SQL-DAY-TO-HOUR                      PIC S9(4) BINARY VALUE IS     8.
01 SQL-DAY-TO-MINUTE                    PIC S9(4) BINARY VALUE IS     9.
01 SQL-DAY-TO-SECOND                    PIC S9(4) BINARY VALUE IS    10.
```

```
01 SQL-HOUR                          PIC S9(4) BINARY VALUE IS     4.
01 SQL-HOUR-TO-MINUTE                PIC S9(4) BINARY VALUE IS    11.
01 SQL-HOUR-TO-SECOND                PIC S9(4) BINARY VALUE IS    12.
01 SQL-MINUTE                        PIC S9(4) BINARY VALUE IS     5.
01 SQL-MINUTE-TO-SECOND              PIC S9(4) BINARY VALUE IS    13.
01 SQL-MONTH                         PIC S9(4) BINARY VALUE IS     2.
01 SQL-SECOND                        PIC S9(4) BINARY VALUE IS     6.
01 SQL-YEAR                          PIC S9(4) BINARY VALUE IS     1.
01 SQL-YEAR-TO-MONTH                 PIC S9(4) BINARY VALUE IS     7.
* CLI OPTION VALUES
01 SQL-FALSE                         PIC S9(4) BINARY VALUE IS     0.
01 SQL-FALSEL                        PIC S9(9) BINARY VALUE IS     0.
01 SQL-TRUE                          PIC S9(4) BINARY VALUE IS     1.
01 SQL-TRUEL                         PIC S9(9) BINARY VALUE IS     1.
01 SQL-NONSCROLLABLE                 PIC S9(9) BINARY VALUE IS     0.
01 SQL-SCROLLABLE                    PIC S9(9) BINARY VALUE IS     1.
01 SQL-NONHOLDABLE                   PIC S9(9) BINARY VALUE IS     0.
01 SQL-HOLDABLE                      PIC S9(9) BINARY VALUE IS     1.
01 SQL-INITIALLY-DEFERRED            PIC S9(9) BINARY VALUE IS     5.
01 SQL-INITIALLY-IMMEDIATE           PIC S9(9) BINARY VALUE IS     6.
01 SQL-NOT-DEFERRABLE                PIC S9(9) BINARY VALUE IS     7.
* PARAMETER MODE VALUES
01 SQL-PARAM-MODE-IN                 PIC S9(4) BINARY VALUE IS     1.
01 SQL-PARAM-MODE-OUT                PIC S9(4) BINARY VALUE IS     4.
01 SQL-PARAM-MODE-INOUT              PIC S9(4) BINARY VALUE IS     2.
* CODES USED FOR FETCHORIENTATION
01 SQL-FETCH-NEXT                    PIC S9(4) BINARY VALUE IS     1.
01 SQL-FETCH-FIRST                   PIC S9(4) BINARY VALUE IS     2.
01 SQL-FETCH-LAST                    PIC S9(4) BINARY VALUE IS     3.
01 SQL-FETCH-PRIOR                   PIC S9(4) BINARY VALUE IS     4.
01 SQL-FETCH-ABSOLUTE                PIC S9(4) BINARY VALUE IS     5.
01 SQL-FETCH-RELATIVE                PIC S9(4) BINARY VALUE IS     6.
* VALUES OF NULLABLE FIELD IN DESCRIPTOR
01 SQL-NO-NULLS                      PIC S9(4) BINARY VALUE IS     0.
01 SQL-NULLABLE                      PIC S9(4) BINARY VALUE IS     1.
* VALUES RETURNED BY SQLRGETTYPEINFO FOR THE SEARCHABLE COLUMN
01 SQL-PRED-NONE                     PIC S9(4) BINARY VALUE IS     0.
01 SQL-PRED-CHAR                     PIC S9(4) BINARY VALUE IS     1.
01 SQL-PRED-BASIC                    PIC S9(4) BINARY VALUE IS     2.
* VALUES OF UNNAMED FIELD IN DESCRIPTOR
01 SQL-NAMED                         PIC S9(4) BINARY VALUE IS     0.
01 SQL-UNNAMED                       PIC S9(4) BINARY VALUE IS     1.
* VALUES OF ALLOC-TYPE FIELD IN DESCRIPTOR
01 SQL-DESC-ALLOC-AUTO               PIC S9(4) BINARY VALUE IS     1.
01 SQL-DESC-ALLOC-USER               PIC S9(4) BINARY VALUE IS     2.
* SQLRENDTRAN OPTIONS
01 SQL-COMMIT                        PIC S9(4) BINARY VALUE IS     0.
01 SQL-ROLLBACK                      PIC S9(4) BINARY VALUE IS     1.
01 SQL-SAVEPOINT-NAME-ROLLBACK       PIC S9(4) BINARY VALUE IS     2.
01 SQL-SAVEPOINT-NAME-RELEASE        PIC S9(4) BINARY VALUE IS     4.
01 SQL-COMMIT-AND-CHAIN              PIC S9(4) BINARY VALUE IS     6.
01 SQL-ROLLBACK-AND-CHAIN            PIC S9(4) BINARY VALUE IS     7.
* SQLRFREESTMT OPTIONS
01 SQL-CLOSE-CURSOR                  PIC S9(4) BINARY VALUE IS     0.
01 SQL-FREE-HANDLE                   PIC S9(4) BINARY VALUE IS     1.
01 SQL-UNBIND-COLUMNS                PIC S9(4) BINARY VALUE IS     2.
01 SQL-UNBIND-PARAMETERS             PIC S9(4) BINARY VALUE IS     3.
```

```
01 SQL-REALLOCATE                 PIC S9(4) BINARY VALUE IS    4.
* PROVIDED FOR BACKWARDS COMPABILITY
01 SQL-CLOSE                      PIC S9(4) BINARY VALUE IS    0.
01 SQL-DROP                       PIC S9(4) BINARY VALUE IS    1.
01 SQL-UNBIND                     PIC S9(4) BINARY VALUE IS    2.
01 SQL-RESET-PARAMS               PIC S9(4) BINARY VALUE IS    3.
* NULL HANDLE USED WHEN ALLOCATING HENV
01 SQL-NULL-HANDLE                PIC S9(9) BINARY VALUE IS    0.
* NULL HANDLES RETURNED BY SQLRALLOCHANDLE
01 SQL-NULL-HENV                  PIC S9(9) BINARY VALUE IS    0.
01 SQL-NULL-HDBC                  PIC S9(9) BINARY VALUE IS    0.
01 SQL-NULL-HSTMT                 PIC S9(9) BINARY VALUE IS    0.
01 SQL-NULL-HDESC                 PIC S9(9) BINARY VALUE IS    0.
* SQLRGETFUNCTIONS VALUES TO IDENTIFY CLI ROUTINES
01 SQL-API-SQLALLOCCONNECT        PIC S9(4) BINARY VALUE IS    1.
01 SQL-API-SQLALLOCENV            PIC S9(4) BINARY VALUE IS    2.
01 SQL-API-SQLALLOCHANDLE         PIC S9(4) BINARY VALUE IS 1001.
01 SQL-API-SQLALLOCSTMT           PIC S9(4) BINARY VALUE IS    3.
01 SQL-API-SQLBINDCOL             PIC S9(4) BINARY VALUE IS    4.
01 SQL-API-SQLBINDPARAMETER       PIC S9(4) BINARY VALUE IS   72.
01 SQL-API-SQLCANCEL              PIC S9(4) BINARY VALUE IS    5.
01 SQL-API-SQLCLOSECURSOR         PIC S9(4) BINARY VALUE IS 1003.
01 SQL-API-SQLCOLATTRIBUTE        PIC S9(4) BINARY VALUE IS    6.
01 SQL-API-SQLCOLUMNPRIVILEGES    PIC S9(4) BINARY VALUE IS   56.
01 SQL-API-SQLCOLUMNS             PIC S9(4) BINARY VALUE IS   40.
01 SQL-API-SQLCONNECT             PIC S9(4) BINARY VALUE IS    7.
01 SQL-API-SQLCOPYDESC            PIC S9(4) BINARY VALUE IS 1004.
01 SQL-API-SQLDATASOURCES         PIC S9(4) BINARY VALUE IS   57.
01 SQL-API-SQLDESCRIBECOL         PIC S9(4) BINARY VALUE IS    8.
01 SQL-API-SQLDISCONNECT          PIC S9(4) BINARY VALUE IS    9.
01 SQL-API-SQLENDTRAN             PIC S9(4) BINARY VALUE IS 1005.
01 SQL-API-SQLERROR               PIC S9(4) BINARY VALUE IS   10.
01 SQL-API-SQLEXECDIRECT          PIC S9(4) BINARY VALUE IS   11.
01 SQL-API-SQLEXECUTE             PIC S9(4) BINARY VALUE IS   12.
01 SQL-API-SQLFETCH               PIC S9(4) BINARY VALUE IS   13.
01 SQL-API-SQLFETCHSCROLL         PIC S9(4) BINARY VALUE IS 1021.
01 SQL-API-SQLFOREIGNKEYS         PIC S9(4) BINARY VALUE IS   60.
01 SQL-API-SQLFREECONNECT         PIC S9(4) BINARY VALUE IS   14.
01 SQL-API-SQLFREEENV             PIC S9(4) BINARY VALUE IS   15.
01 SQL-API-SQLFREEHANDLE          PIC S9(4) BINARY VALUE IS 1006.
01 SQL-API-SQLFREESTMT            PIC S9(4) BINARY VALUE IS   16.
01 SQL-API-SQLGETCONNECTATTR      PIC S9(4) BINARY VALUE IS 1007.
01 SQL-API-SQLGETCURSORNAME       PIC S9(4) BINARY VALUE IS   17.
01 SQL-API-SQLGETDATA             PIC S9(4) BINARY VALUE IS   43.
01 SQL-API-SQLGETDESCFIELD        PIC S9(4) BINARY VALUE IS 1008.
01 SQL-API-SQLGETDESCREC          PIC S9(4) BINARY VALUE IS 1009.
01 SQL-API-SQLGETDIAGFIELD        PIC S9(4) BINARY VALUE IS 1010.
01 SQL-API-SQLGETDIAGREC          PIC S9(4) BINARY VALUE IS 1011.
01 SQL-API-SQLGETENVATTR          PIC S9(4) BINARY VALUE IS 1012.
01 SQL-API-SQLGETFEATUREINFO      PIC S9(4) BINARY VALUE IS 1027.
01 SQL-API-SQLGETFUNCTIONS        PIC S9(4) BINARY VALUE IS   44.
01 SQL-API-SQLGETINFO             PIC S9(4) BINARY VALUE IS   45.
01 SQL-API-SQLGETLENGTH           PIC S9(4) BINARY VALUE IS 1022.
01 SQL-API-SQLGETPARAMDATA        PIC S9(4) BINARY VALUE IS 1025.
01 SQL-API-SQLGETPOSITION         PIC S9(4) BINARY VALUE IS 1023.
01 SQL-API-SQLGETSESSIONINFO      PIC S9(4) BINARY VALUE IS 1028.
01 SQL-API-SQLGETSTMTATTR         PIC S9(4) BINARY VALUE IS 1014.
```

```
01 SQL-API-SQLGETSUBSTRING              PIC S9(4) BINARY VALUE IS  1024.
01 SQL-API-SQLGETTYPEINFO               PIC S9(4) BINARY VALUE IS    47.
01 SQL-API-SQLMORERESULTS               PIC S9(4) BINARY VALUE IS    61.
01 SQL-API-SQLNEXTRESULT                PIC S9(4) BINARY VALUE IS    73.
01 SQL-API-SQLNUMRESULTCOLS             PIC S9(4) BINARY VALUE IS    18.
01 SQL-API-SQLPARAMDATA                 PIC S9(4) BINARY VALUE IS    48.
01 SQL-API-SQLPREPARE                   PIC S9(4) BINARY VALUE IS    19.
01 SQL-API-SQLPRIMARYKEYS               PIC S9(4) BINARY VALUE IS    65.
01 SQL-API-SQLPUTDATA                   PIC S9(4) BINARY VALUE IS    49.
01 SQL-API-SQLROWCOUNT                  PIC S9(4) BINARY VALUE IS    20.
01 SQL-API-SQLSETCONNECTATTR            PIC S9(4) BINARY VALUE IS  1016.
01 SQL-API-SQLSETCURSORNAME             PIC S9(4) BINARY VALUE IS    21.
01 SQL-API-SQLSETDESCFIELD              PIC S9(4) BINARY VALUE IS  1017.
01 SQL-API-SQLSETDESCREC                PIC S9(4) BINARY VALUE IS  1018.
01 SQL-API-SQLSETENVATTR                PIC S9(4) BINARY VALUE IS  1019.
01 SQL-API-SQLSETSTMTATTR               PIC S9(4) BINARY VALUE IS  1020.
01 SQL-API-SQLSPECIALCOLUMNS            PIC S9(4) BINARY VALUE IS    52.
01 SQL-API-SQLSTARTTRAN                 PIC S9(4) BINARY VALUE IS    74.
01 SQL-API-SQLTABLES                    PIC S9(4) BINARY VALUE IS    54.
01 SQL-API-SQLTABLEPRIVILEGES           PIC S9(4) BINARY VALUE IS    70.
* INFORMATION REQUESTED BY SQLRGETINFO
01 SQL-MAXIMUM-DRIVER-CONNECTIONS       PIC S9(4) BINARY VALUE IS     0.
01 SQL-MAXIMUM-CONCURRENT-ACTIVITIES    PIC S9(4) BINARY VALUE IS     1.
01 SQL-DATA-SOURCE-NAME                 PIC S9(4) BINARY VALUE IS     2.
01 SQL-FETCH-DIRECTION                  PIC S9(4) BINARY VALUE IS     8.
01 SQL-SERVER-NAME                      PIC S9(4) BINARY VALUE IS    13.
01 SQL-SEARCH-PATTERN-ESCAPE            PIC S9(4) BINARY VALUE IS    14.
01 SQL-DBMS-NAME                        PIC S9(4) BINARY VALUE IS    17.
01 SQL-DBMS-VERSION                     PIC S9(4) BINARY VALUE IS    18.
01 SQL-CURSOR-COMMIT-BEHAVIOR           PIC S9(4) BINARY VALUE IS    23.
01 SQL-DATA-SOURCE-READ-ONLY            PIC S9(4) BINARY VALUE IS    25.
01 SQL-DEFAULT-TRANSACTION-ISOLATION    PIC S9(4) BINARY VALUE IS    26.
01 SQL-IDENTIFIER-CASE                  PIC S9(4) BINARY VALUE IS    28.
01 SQL-MAXIMUM-COLUMN-NAME-LENGTH       PIC S9(4) BINARY VALUE IS    30.
01 SQL-MAXIMUM-CURSOR-NAME-LENGTH       PIC S9(4) BINARY VALUE IS    31.
01 SQL-MAXIMUM-SCHEMA-NAME-LENGTH       PIC S9(4) BINARY VALUE IS    32.
01 SQL-MAXIMUM-CATALOG-NAME-LENGTH      PIC S9(4) BINARY VALUE IS    34.
01 SQL-MAXIMUM-TABLE-NAME-LENGTH        PIC S9(4) BINARY VALUE IS    35.
01 SQL-SCROLL-CONCURRENCY               PIC S9(4) BINARY VALUE IS    43.
01 SQL-TRANSACTION-CAPABLE              PIC S9(4) BINARY VALUE IS    46.
01 SQL-USER-NAME                        PIC S9(4) BINARY VALUE IS    47.
01 SQL-TRANSACTION-ISOLATION-OPTION     PIC S9(4) BINARY VALUE IS    72.
01 SQL-INTEGRITY                        PIC S9(4) BINARY VALUE IS    73.
01 SQL-GETDATA-EXTENSIONS               PIC S9(4) BINARY VALUE IS    81.
01 SQL-NULL-COLLATION                   PIC S9(4) BINARY VALUE IS    85.
01 SQL-ALTER-TABLE                      PIC S9(4) BINARY VALUE IS    86.
01 SQL-ORDER-BY-COLUMNS-IN-SELECT       PIC S9(4) BINARY VALUE IS    90.
01 SQL-SPECIAL-CHARACTERS               PIC S9(4) BINARY VALUE IS    94.
01 SQL-MAXIMUM-COLUMNS-IN-GROUP-BY      PIC S9(4) BINARY VALUE IS    97.
01 SQL-MAXIMUM-COLUMNS-IN-ORDER-BY      PIC S9(4) BINARY VALUE IS    99.
01 SQL-MAXIMUM-COLUMNS-IN-SELECT        PIC S9(4) BINARY VALUE IS   100.
01 SQL-MAXIMUM-COLUMNS-IN-TABLE         PIC S9(4) BINARY VALUE IS   101.
01 SQL-MAXIMUM-STMT-OCTETS              PIC S9(4) BINARY VALUE IS 20000.
01 SQL-MAXIMUM-STMT-OCTETS-DATA         PIC S9(4) BINARY VALUE IS 20001.
01 SQL-MAXIMUM-STMT-OCTETS-SCHEMA       PIC S9(4) BINARY VALUE IS 20002.
01 SQL-MAXIMUM-TABLES-IN-SELECT         PIC S9(4) BINARY VALUE IS   106.
01 SQL-MAXIMUM-USER-NAME-LENGTH         PIC S9(4) BINARY VALUE IS   107.
```

```
01 SQL-OUTER-JOIN-CAPABILITIES          PIC S9(4) BINARY VALUE IS   115.
01 SQL-CURSOR-SENSITIVITY               PIC S9(4) BINARY VALUE IS 10001.
01 SQL-DESCRIBE-PARAMETER               PIC S9(4) BINARY VALUE IS 10002.
01 SQL-CATALOG-NAME                     PIC S9(4) BINARY VALUE IS 10003.
01 SQL-COLLATING-SEQUENCE               PIC S9(4) BINARY VALUE IS 10004.
01 SQL-MAXIMUM-IDENTIFIER-LENGTH        PIC S9(4) BINARY VALUE IS 10005.
* INFORMATION REQUESTED BY SQLRGETSESSIONINFO
01 SQL-CURRENT-USER                     PIC S9(4) BINARY VALUE IS    47.
01 SQL-CURRENT-DEFAULT-TRANSFORM-GROUP  PIC S9(4) BINARY VALUE IS 20004.
01 SQL-CURRENT-PATH                     PIC S9(4) BINARY VALUE IS 20005.
01 SQL-CURRENT-ROLE                     PIC S9(4) BINARY VALUE IS 20006.
01 SQL-SESSION-USER                     PIC S9(4) BINARY VALUE IS 20007.
01 SQL-SYSTEM-USER                      PIC S9(4) BINARY VALUE IS 20008.
01 SQL-CURRENT-CATALOG                  PIC S9(4) BINARY VALUE IS 20009.
01 SQL-CURRENT-SCHEMA                   PIC S9(4) BINARY VALUE IS 20010.
* STATEMENT ATTRIBUTE VALUES FOR CURSOR SENSITIVITY
01 SQL-ASENSITIVE                       PIC S9(9) BINARY VALUE IS     0.
01 SQL-INSENSITIVE                      PIC S9(9) BINARY VALUE IS     1.
01 SQL-SENSITIVE                        PIC S9(9) BINARY VALUE IS     2.
* DEFINE SQL-UNSPECIFIED FOR BACKWARDS COMPATIBILITY
01 SQL-UNSPECIFIED                      PIC S9(9) BINARY VALUE IS     0.
* SQL-ALTER-TABLE VALUES
01 SQL-AT-ADD-COLUMN                    PIC S9(9) BINARY VALUE IS     1.
01 SQL-AT-DROP-COLUMN                   PIC S9(9) BINARY VALUE IS     2.
01 SQL-AT-ALTER-COLUMN                  PIC S9(9) BINARY VALUE IS     4.
01 SQL-AT-ADD-CONSTRAINT                PIC S9(9) BINARY VALUE IS     8.
01 SQL-AT-DROP-CONSTRAINT               PIC S9(9) BINARY VALUE IS    16.
* SQL-CURSOR-COMMIT-BEHAVIOR VALUES
01 SQL-CB-DELETE                        PIC S9(4) BINARY VALUE IS     0.
01 SQL-CB-CLOSE                         PIC S9(4) BINARY VALUE IS     1.
01 SQL-CB-PRESERVE                      PIC S9(4) BINARY VALUE IS     2.
* SQL-FETCH-DIRECTION VALUES
01 SQL-FD-FETCH-NEXT                    PIC S9(9) BINARY VALUE IS     1.
01 SQL-FD-FETCH-FIRST                   PIC S9(9) BINARY VALUE IS     2.
01 SQL-FD-FETCH-LAST                    PIC S9(9) BINARY VALUE IS     4.
01 SQL-FD-FETCH-PRIOR                   PIC S9(9) BINARY VALUE IS     8.
01 SQL-FD-FETCH-ABSOLUTE                PIC S9(9) BINARY VALUE IS    16.
01 SQL-FD-FETCH-RELATIVE                PIC S9(9) BINARY VALUE IS    32.
* SQL-GETDATA-EXTENSIONS VALUES
01 SQL-GD-ANY-COLUMNS                   PIC S9(9) BINARY VALUE IS     1.
01 SQL-GD-ANY-ORDER                     PIC S9(9) BINARY VALUE IS     2.
* SQL-IDENTIFIER-CASE VALUES
01 SQL-IC-UPPER                         PIC S9(4) BINARY VALUE IS     1.
01 SQL-IC-LOWER                         PIC S9(4) BINARY VALUE IS     2.
01 SQL-IC-SENSITIVE                     PIC S9(4) BINARY VALUE IS     3.
01 SQL-IC-MIXED                         PIC S9(4) BINARY VALUE IS     4.
* SQL-NULL-COLLATION VALUES
01 SQL-NC-HIGH                          PIC S9(4) BINARY VALUE IS     1.
01 SQL-NC-LOW                           PIC S9(4) BINARY VALUE IS     2.
* SQL-OUTER-JOIN-CAPABILITIES VALUES
01 SQL-OUTER-JOIN-LEFT                  PIC S9(9) BINARY VALUE IS     1.
01 SQL-OUTER-JOIN-RIGHT                 PIC S9(9) BINARY VALUE IS     2.
01 SQL-OUTER-JOIN-FULL                  PIC S9(9) BINARY VALUE IS     4.
01 SQL-OUTER-JOIN-NESTED                PIC S9(9) BINARY VALUE IS     8.
01 SQL-OUTER-JOIN-NOT-ORDERED           PIC S9(9) BINARY VALUE IS    16.
01 SQL-OUTER-JOIN-INNER                 PIC S9(9) BINARY VALUE IS    32.
01 SQL-OUTER-JOIN-ALL-COMPARISON-OPS    PIC S9(9) BINARY VALUE IS    64.
```

```
* SQL-SCROLL-CONCURRENCY VALUES
01 SQL-SCCO-READ-ONLY                PIC S9(9) BINARY VALUE IS     1.
01 SQL-SCCO-LOCK                     PIC S9(9) BINARY VALUE IS     2.
01 SQL-SCCO-OPT-ROWVER               PIC S9(9) BINARY VALUE IS     4.
01 SQL-SCCO-OPT-VALUES               PIC S9(9) BINARY VALUE IS     8.
* SQL-TRANSACTION-CAPABLE VALUES
01 SQL-TC-NONE                       PIC S9(4) BINARY VALUE IS     0.
01 SQL-TC-DML                        PIC S9(4) BINARY VALUE IS     1.
01 SQL-TC-ALL                        PIC S9(4) BINARY VALUE IS     2.
01 SQL-TC-ALL-COMMIT                 PIC S9(4) BINARY VALUE IS     3.
01 SQL-TC-DDL-IGNORE                 PIC S9(4) BINARY VALUE IS     4.
* SQL-TRANSACTION-ISOLATION VALUES
01 SQL-TRANSACTION-READ-UNCOMMITTED  PIC S9(9) BINARY VALUE IS     1.
01 SQL-TRANSACTION-READ-COMMITTED    PIC S9(9) BINARY VALUE IS     2.
01 SQL-TRANSACTION-REPEATABLE-READ   PIC S9(9) BINARY VALUE IS     4.
01 SQL-TRANSACTION-SERIALIZABLE      PIC S9(9) BINARY VALUE IS     8.
* SQL-TRANSACTION-ACCESS-MODE VALUES
01 SQL-TRANSACTION-READ-ONLY         PIC S9(9) BINARY VALUE IS     1.
01 SQL-TRANSACTION-READ-WRITE        PIC S9(9) BINARY VALUE IS     2.
* COLUMN TYPES AND SCOPES IN SPECIALCOLUMNS
01 SQL-BEST-ROWID                    PIC S9(4) BINARY VALUE IS     1.
01 SQL-SCOPE-CURRROW                 PIC S9(4) BINARY VALUE IS     0.
01 SQL-SCOPE-TRANSACTION             PIC S9(4) BINARY VALUE IS     1.
01 SQL-SCOPE-SESSION                 PIC S9(4) BINARY VALUE IS     2.
01 SQL-PC-UNKNOWN                    PIC S9(4) BINARY VALUE IS     0.
01 SQL-PC-NOT-PSEUDO                 PIC S9(4) BINARY VALUE IS     1.
01 SQL-PC-PSEUDO                     PIC S9(4) BINARY VALUE IS     2.
* FOREIGN KEY UPDATE AND DELETE RULES
01 SQL-CASCADE                       PIC S9(4) BINARY VALUE IS     0.
01 SQL-RESTRICT                      PIC S9(4) BINARY VALUE IS     1.
01 SQL-SET-NULL                      PIC S9(4) BINARY VALUE IS     2.
01 SQL-NO-ACTION                     PIC S9(4) BINARY VALUE IS     3.
01 SQL-SET-DEFAULT                   PIC S9(4) BINARY VALUE IS     4.
* SPECIAL PARAMETER VALUES
01 SQL-ALL-CATALOGS                            PIC X  VALUE IS   '%'.
01 SQL-ALL-SCHEMAS                             PIC X  VALUE IS   '%'.
01 SQL-ALL-TABLE-TYPES                         PIC X  VALUE IS   '%'.
```

(Blank page)

Annex B
(informative)

Sample C programs

This Annex includes three examples of using SQL/CLI.

The first example illustrates creating a table, adding some data to it, and selecting the inserted data. The second example shows interactive *ad hoc* query processing. The third example demonstrates how to provide long dynamic arguments at Execute time.

Actual SQL/CLI applications include more complete error checking following calls to SQL/CLI routines. That material is omitted from this Annex for the sake of clarity.

B.1 Create table, insert, select

This example function creates a table, inserts data into the table, and selects the inserted data.

This example illustrates the execution of SQL statement text both using the Prepare and Execute method and using the ExecDirect method. The example also illustrates both the case where the SQL/CLI application uses the automatically-generated descriptors and the case where the SQL/CLI application allocates a descriptor of its own and associates this descriptor with the SQL statement.

Code comments include the equivalent statements in embedded SQL to show how embedded SQL operations correspond to SQL/CLI function calls.

```
#include <stddef.h>
#include <string.h>
#include <sqlcli.h>
#define NAMELEN 50
int print_err(SQLSMALLINT handletype, SQLINTEGER handle);
int example1(SQLCHAR *server, SQLCHAR *uid, SQLCHAR *authen)
{
SQLHENV     henv;
SQLHDBC     hdbc;
SQLHDESC    hdesc;
SQLHDESC    hdesc1;
SQLHDESC    hdesc2;
SQLHSTMT    hstmt;
SQLINTEGER  id;
SQLINTEGER  idind;
SQLCHAR     name[NAMELEN+1];
SQLINTEGER  namelen;
SQLINTEGER  nameind;
/* EXEC SQL CONNECT TO :server USER :uid; */
/* allocate an environment handle */
```

```
SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);
/* allocate a connection handle */
SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
/* connect to database */
if (SQLConnect(hdbc, server, SQL_NTS, uid, SQL_NTS,
               authen, SQL_NTS)
      != SQL_SUCCESS)
   return(print_err(SQL_HANDLE_DBC, hdbc));
/* allocate a statement handle */
SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);
/* EXEC SQL CREATE TABLE NAMEID (ID integer, NAME varchar(50));  */
{
   SQLCHAR create[] ="CREATE TABLE NAMEID (ID integer,"
                                       " NAME varchar(50))";
   /* execute the CREATE TABLE statement */
   if (SQLExecDirect(hstmt, create, SQL_NTS) != SQL_SUCCESS)
      return(print_err(SQL_HANDLE_STMT, hstmt));
}
/* EXEC SQL COMMIT WORK; */
/* commit CREATE TABLE */
SQLEndTran(SQL_HANDLE_ENV, henv, SQL_COMMIT);
/* EXEC SQL INSERT INTO NAMEID VALUES ( :id, :name ); */
{
   SQLCHAR insert[]= "INSERT INTO NAMEID VALUES (?, ?)";
   /* show the use of SQLPrepare/SQLExecute method */
   /* prepare the INSERT */
   if (SQLPrepare(hstmt, insert, SQL_NTS) != SQL_SUCCESS)
      return(print_err(SQL_HANDLE_STMT, hstmt));
   /* application parameter descriptor */
   SQLGetStmtAttr(hstmt, SQL_ATTR_APP_PARAM_DESC, &hdesc1, 0L,
         (SQLINTEGER *)NULL);
   SQLSetDescRec(hdesc1, 1, SQL_INTEGER, 0, 0L, 0, 0,
         (SQLPOINTER)&id, (SQLINTEGER *)NULL, (SQLINTEGER *)NULL);
   SQLSetDescRec(hdesc1, 2, SQL_CHAR, 0, NAMELEN, 0, 0,
         (SQLPOINTER)name, (SQLINTEGER *)NULL,
         (SQLINTEGER *)NULL);
   /* implementation parameter descriptor */
   SQLGetStmtAttr(hstmt, SQL_ATTR_IMP_PARAM_DESC, &hdesc2, 0L,
         (SQLINTEGER *)NULL);
   SQLSetDescRec(hdesc2, 1, SQL_INTEGER, 0, 0L, 0, 0,
         (SQLPOINTER)NULL, (SQLINTEGER *)NULL,
         (SQLINTEGER *)NULL);
   SQLSetDescRec(hdesc2, 2, SQL_VARCHAR, 0, NAMELEN, 0, 0,
         (SQLPOINTER)NULL, (SQLINTEGER *)NULL,
         (SQLINTEGER *)NULL);
   /* assign parameter values and execute the INSERT */
   id=500;
   (void)strcpy((char *)name, "Babbage");
   if (SQLExecute(hstmt) != SQL_SUCCESS)
      return(print_err(SQL_HANDLE_STMT, hstmt));
}
/* EXEC SQL COMMIT WORK; */
SQLEndTran(SQL_HANDLE_ENV, henv, SQL_COMMIT); /* commit inserts */
/* EXEC SQL DECLARE c1 CURSOR FOR SELECT ID, NAME FROM NAMEID; */
/* EXEC SQL OPEN c1; */
/* The application doesn't specify "declare c1 cursor for" */
{
```

```
   SQLCHAR select[]= "select ID, NAME from NAMEID";
   if (SQLExecDirect(hstmt, select, SQL_NTS) != SQL_SUCCESS)
      return(print_err(SQL_HANDLE_STMT, hstmt));
}
/* EXEC SQL FETCH c1 INTO :id, :name; */
/* this time, explicitly allocate an application row descriptor */
SQLAllocHandle(SQL_HANDLE_DESC, hdbc, &hdesc);
SQLSetDescRec(hdesc, 1, SQL_INTEGER, 0, 0L, 0, 0,
   (SQLPOINTER)&id, (SQLINTEGER *)NULL, (SQLINTEGER *)&idind);
SQLSetDescRec(hdesc, 2, SQL_CHAR, 0, NAMELEN,
   0, 0, (SQLPOINTER)&name, (SQLINTEGER *)&namelen,
   (SQLINTEGER *)&nameind);
/* associate descriptor with statement handle */
SQLSetStmtAttr(hstmt, SQL_ATTR_APP_ROW_DESC, &hdesc, 0);
/* execute the fetch */
SQLFetch(hstmt);
/* EXEC SQL CLOSE c1; */
SQLCloseCursor(hstmt);
/* EXEC SQL COMMIT WORK; */
/* commit the transaction  */
SQLEndTran(SQL_HANDLE_ENV, henv, SQL_COMMIT);
/* free the statement handle */
SQLFreeHandle(SQL_HANDLE_STMT, hstmt);
/* EXEC SQL DISCONNECT; */
/* disconnect from the database */
SQLDisconnect(hdbc);
/* free descriptor handle */
SQLFreeHandle(SQL_HANDLE_DESC, hdesc);
/* free descriptor handle */
SQLFreeHandle(SQL_HANDLE_DESC, hdesc1);
/* free descriptor handle */
SQLFreeHandle(SQL_HANDLE_DESC, hdesc2);
/* free connection handle */
SQLFreeHandle(SQL_HANDLE_DBC,  hdbc);
/* free environment handle */
SQLFreeHandle(SQL_HANDLE_ENV,  henv);
return(0);
}
```

B.2 Interactive Query

This sample function uses the concise CLI functions to interactively execute an SQL-statement supplied as an argument. In the case where the user types a SELECT statement, the function fetches and displays all rows of the result set.

This example illustrates the use of GetDiagField to identify the type of SQL statement executed and, for SQL statements where the row count is defined on all implementations, the use of GetDiagField to obtain the row count.

```
/*
 * Sample program - uses concise CLI functions to execute
 * interactively an ad hoc statement.
 */
#include <stddef.h>
#include <string.h>
#include <stdlib.h>
#include <sqlcli.h>
#define MAXCOLS   100
#define max(a,b) ((a)>(b)?(a):(b))
int print_err(SQLSMALLINT handletype, SQLINTEGER handle);
int build_indicator_message(SQLCHAR *errmsg, SQLPOINTER *data,
   SQLINTEGER collen, SQLINTEGER *outlen, SQLSMALLINT colnum);
   SQLINTEGER display_length(SQLSMALLINT coltype, SQLINTEGER collen,
   SQLCHAR *colname);
example2(SQLCHAR *server, SQLCHAR *uid, SQLCHAR *authen, SQLCHAR *sqlstr)
{
 int         i;
 SQLHENV     henv;
 SQLHDBC     hdbc;
 SQLHSTMT    hstmt;
 SQLCHAR     errmsg[256];
 SQLCHAR     colname[32];
 SQLSMALLINT coltype;
 SQLSMALLINT colnamelen;
 SQLSMALLINT nullable;
 SQLINTEGER  collen[MAXCOLS];
 SQLSMALLINT scale;
 SQLINTEGER  outlen[MAXCOLS];
 SQLCHAR    *data[MAXCOLS];
 SQLSMALLINT nresultcols;
 SQLINTEGER  rowcount;
 SQLINTEGER  stmttype;
 SQLRETURN   rc;
 /* allocate an environment handle */
 SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);
 /* allocate a connection handle */
 SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
 /* connect to database */
 if (SQLConnect(hdbc, server, SQL_NTS, uid, SQL_NTS, authen, SQL_NTS)
       != SQL_SUCCESS )
    return(print_err(SQL_HANDLE_DBC, hdbc));
 /* allocate a statement handle */
 SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);
 /* execute the SQL statement */
```

```
if (SQLExecDirect(hstmt, sqlstr, SQL_NTS) != SQL_SUCCESS)
   return(print_err(SQL_HANDLE_STMT, hstmt));
/* see what kind of statement it was */
SQLGetDiagField(SQL_HANDLE_STMT, hstmt, 0,
      SQL_DIAG_DYNAMIC_FUNCTION_CODE,
      (SQLPOINTER)&stmttype, 0, (SQLSMALLINT *)NULL);
switch (stmttype) {
   /* SELECT statement */
   case SQL_DIAG_SELECT_CURSOR:
      /* determine number of result columns */
      SQLNumResultCols(hstmt, &nresultcols);
      /* display column names */
      for (i=0; i < nresultcols; i++) {
         SQLDescribeCol(hstmt, i+1, colname, sizeof(colname),
            &colnamelen, &coltype, &collen[i], &scale, &nullable);
         /* assume there is a display_length function which
            computes correct length given the data type  */
         collen[i] = display_length(coltype, collen[i], colname);
         (void)printf("%*.*s", (int)collen[i], (int)collen[i],
                      (char *)colname);
         /* allocate memory to bind column */
         data[i] = (SQLCHAR *) malloc(collen[i]);
         /* bind columns to program vars, converting all types
            to CHAR */
         SQLBindCol(hstmt, i+1, SQL_CHAR, data[i], collen[i],
            &outlen[i]);
      }
      /* display result rows */
      while ((rc=SQLFetch(hstmt))!=SQL_ERROR) {
         errmsg[0] = '\0';
         if (rc == SQL_SUCCESS_WITH_INFO) {
            for (i=0; i < nresultcols; i++) {
               if (outlen[i] == SQL_NULL_DATA
                           || outlen[i] >= collen[i])
                  build_indicator_message(errmsg,
                     (SQLPOINTER *)&data[i], collen[i],
                     &outlen[i], i);
               (void)printf("%*.*s ", (int)outlen[i], (int)outlen[i],
                     (char *)data[i]);
            } /* for all columns in this row  */
            /* print any truncation messages */
            (void)printf("\n%s", (char *)errmsg);
         }
      } /* while rows to fetch */
      SQLCloseCursor(hstmt);
      break;
   /* searched DELETE, INSERT, MERGE, or searched UPDATE statement */
   case SQL_DIAG_DELETE_WHERE:
   case SQL_DIAG_INSERT:
   case SQL_DIAG_MERGE:
   case SQL_DIAG_UPDATE_WHERE:
      /* check rowcount */
      SQLGetDiagField(SQL_HANDLE_STMT, hstmt, 0,
         SQL_DIAG_ROW_COUNT, (SQLPOINTER)&rowcount, 0,
         (SQLSMALLINT *)NULL);
      if (SQLEndTran(SQL_HANDLE_ENV, henv, SQL_COMMIT)
            == SQL_SUCCESS) {
```

```
            (void) printf("Operation successful\n");
         }
         else {
            (void) printf("Operation failed\n");
         }
         (void)printf("%ld rows affected\n", rowcount);
         break;
      /* other statements */
      case SQL_DIAG_ALTER_TABLE:
      case SQL_DIAG_CREATE_TABLE:
      case SQL_DIAG_CREATE_VIEW:
      case SQL_DIAG_DROP_TABLE:
      case SQL_DIAG_DROP_VIEW:
      case SQL_DIAG_DYNAMIC_DELETE_CURSOR:
      case SQL_DIAG_DYNAMIC_UPDATE_CURSOR:
      case SQL_DIAG_GRANT:
      case SQL_DIAG_REVOKE:
         if (SQLEndTran(SQL_HANDLE_ENV, henv, SQL_COMMIT)
               == SQL_SUCCESS) {
            (void) printf("Operation successful\n");
         }
         else {
            (void) printf("Operation failed\n");
         }
         break;
      /* implementation-defined statement */
      default:
         (void)printf("Statement type=%ld\n", stmttype);
         break;
   }
   /* free data buffers */
   for (i=0; i < nresultcols; i++)  {
      (void)free(data[i]);
   }
   /* free statement handle */
   SQLFreeHandle(SQL_HANDLE_STMT, hstmt);
   /* disconnect from database */
   SQLDisconnect(hdbc);
   /* free connection handle */
   SQLFreeHandle(SQL_HANDLE_DBC, hdbc);
   /* free environment handle */
   SQLFreeHandle(SQL_HANDLE_ENV, henv);
   return(0);
  }
  /************************************************************
  The following functions are given for completeness, but are
  not relevant for understanding the database processing
  nature of CLI
  ***********************************************************/
  #define MAX_NUM_PRECISION 15
  /*#define max length of char string representation of no. as:
     = max(precision) + leading sign + E + exp sign + max exp length
     =  15            + 1            + 1 + 1       + 2
     =  15 + 5
  */
  #define MAX_NUM_STRING_SIZE (MAX_NUM_PRECISION + 5)
  SQLINTEGER  display_length(SQLSMALLINT coltype, SQLINTEGER collen,
```

```
   SQLCHAR *colname)
{
switch (coltype) {
  case SQL_CHAR:
  case SQL_VARCHAR:
  case SQL_CLOB:
  case SQL_BINARY:
  case SQL_VARBINARY:
  case SQL_BLOB:
  case SQL_REF:
      return(max(collen,strlen((char *)colname)));
  case SQL_NUMERIC:
  case SQL_DECIMAL:
  case SQL_FLOAT:
  case SQL_REAL:
  case SQL_DOUBLE:
      return(max(MAX_NUM_STRING_SIZE,strlen((char *)colname)));
  case SQL_TYPE_DATE:
  case SQL_TYPE_TIME:
  case SQL_TYPE_TIME_WITH_TIMEZONE:
  case SQL_TYPE_TIMESTAMP:
  case SQL_TYPE_TIMESTAMP_WITH_TIMEZONE:
  case SQL_INTERVAL_YEAR:
  case SQL_INTERVAL_MONTH:
  case SQL_INTERVAL_DAY:
  case SQL_INTERVAL_HOUR:
  case SQL_INTERVAL_MINUTE:
  case SQL_INTERVAL_SECOND:
  case SQL_INTERVAL_YEAR_TO_MONTH:
  case SQL_INTERVAL_DAY_TO_HOUR:
  case SQL_INTERVAL_DAY_TO_MINUTE:
  case SQL_INTERVAL_DAY_TO_SECOND:
  case SQL_INTERVAL_HOUR_TO_MINUTE:
  case SQL_INTERVAL_HOUR_TO_SECOND:
  case SQL_INTERVAL_MINUTE_TO_SECOND:
      return(max(collen,strlen((char *)colname)));
  case SQL_CLOB_LOCATOR:
  case SQL_BLOB_LOCATOR:
  case SQL_INTEGER:
  case SQL_BIGINT:
  case SQL_UDT_LOCATOR:
  case SQL_ARRAY_LOCATOR:
  case SQL_MULTISET_LOCATOR:
      return(max(10,strlen((char *)colname)));
  case SQL_SMALLINT:
      return(max(5,strlen((char *)colname)));
  default:
      (void)printf("Unknown datatype, %d\n", coltype);
      return(0);
  }
}
int build_indicator_message(SQLCHAR *errmsg, SQLPOINTER *data,
   SQLINTEGER collen, SQLINTEGER *outlen, SQLSMALLINT colnum)
{
  if (*outlen == SQL_NULL_DATA) {
        (void)strcpy((char *)data, "NULL");
        *outlen=4;
```

```
      }
      else {
         sprintf((char *)errmsg+strlen((char *)errmsg),
               "%d chars truncated, col %d\n", *outlen-collen+1,
               colnum);
         *outlen=255;
      }
    }
```

B.3 Providing long dynamic arguments at Execute time

In the following example, an SQL/CLI application prepares an SQL statement to insert data into the EMPLOYEE table. The statement contains parameters for the NAME, ID, and PHOTO columns. For each parameter, the SQL/CLI application calls BindParameter to specify the C and SQL data types of the parameter. It also specifies that the data for the first and third parameters will be passed at execute time, and passes the values 1 (one) and 3 for later retrieval by ParamData. These values will identify which parameter is being processed.

The SQL/CLI application calls GetNextID to get the next available employee ID number. It then calls Execute to execute the statement. The Execute function returns SQL_NEED_DATA when it needs data for the first and third parameters. The SQL/CLI application calls ParamData to retrieve the value it stored with BindParameter; it uses this value to determine which parameter to send data for. For each parameter, the application calls InitUserData to initialise the data routine. It repeatedly calls GetUserData and PutData to get and send the parameter data. Finally, it calls ParamData to indicate it has sent all the data for the parameter and to retrieve the value for the next parameter. After data has been sent for both parameters, ParamData returns SQL_SUCCESS.

For the first parameter, InitUserData does not do anything and GetUserData calls a routine to prompt the user for the employee name. For the third parameter, InitUserData calls a routine to prompt the user for the name of a file containing a bitmap photo of the employee and opens the file. GetUserData retrieves the next MAX_DATA_LENGTH octets of photo data from the file. After it has retrieved all the photo data, it closes the photo file.

Note that some SQL/CLI application routines are omitted for clarity.

```
    /*
     * Sample program - uses ParamData and PutData to pass long data at
     * execute time.
     */
    #include <stddef.h>
    #include <stdio.h>
    #include <string.h>
    #include <sqlcli.h>
    #define NAME_LENGTH            30
    #define MAX_FILE_NAME_LENGTH  256
    #define MAX_PHOTO_LENGTH    32000
    #define MAX_DATA_LENGTH      1024
    int print_err(SQLSMALLINT handletype, SQLINTEGER handle);
    SQLINTEGER GetNextID();
    void InitUserData(SQLSMALLINT sParam, SQLPOINTER InitValue);
    SQLSMALLINT GetUserData(SQLPOINTER InitValue, SQLSMALLINT sParam,
                            SQLCHAR *Data, SQLINTEGER *StrLen_or_Ind);
    example3(SQLCHAR *server, SQLCHAR *uid, SQLCHAR* pwd)
```

```
{
SQLHENV     henv;
SQLHDBC     hdbc;
SQLHSTMT    hstmt;
SQLRETURN   rc;
SQLINTEGER  NameParamLength, IDLength = 0, PhotoParamLength, StrLen_or_Ind;
SQLINTEGER  ID;
SQLSMALLINT Param1=1, Param3=3;
SQLPOINTER  pToken, InitValue;
SQLCHAR     Data[MAX_DATA_LENGTH];
/* allocate an environment handle */
SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);
/* allocate a connection handle */
SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
/* connect to database */
rc = SQLConnect(hdbc, server, SQL_NTS, uid, SQL_NTS, pwd, SQL_NTS);
if (rc != SQL_SUCCESS && rc != SQL_SUCCESS_WITH_INFO)
    return(print_err(SQL_HANDLE_DBC, hdbc));
/* allocate a statement handle */
SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);
/* prepare the INSERT statement */
rc = SQLPrepare(hstmt,
      "INSERT INTO EMPLOYEE (NAME, ID, PHOTO) VALUES (?, ?, ?)",
      SQL_NTS);
if (rc == SQL_SUCCESS) {
    /* Bind the parameters. For parameters 1 and 3, pass the     */
    /* parameter number in ParameterValue instead of a buffer address. */
    SQLBindParameter(hstmt, 1, SQL_PARAM_MODE_IN, SQL_CHAR, SQL_CHAR,
                 NAME_LENGTH, 0, &Param1, NAME_LENGTH, &NameParamLength);
    SQLBindParameter(hstmt, 2, SQL_PARAM_MODE_IN, SQL_INTEGER, SQL_INTEGER,
                 sizeof(ID), 0, &ID, 0 &IDLength);
    SQLBindParameter(hstmt, 3, SQL_PARAM_MODE_IN, SQL_CHAR, SQL_CHAR,
                 MAX_PHOTO_LENGTH, 0, &Param3, MAX_PHOTO_LENGTH, &PhotoParamLength);
    /* Set values so data for parameters 1 and 3 will be passed  */
    /* at execution.                                             */
    NameParamLength = PhotoParamLength = SQL_DATA_AT_EXEC;
    ID = GetNextID();  /* Get next available employee ID number. */
    rc = SQLExecute(hstmt);
    /* For data-at-execution parameters, call SQLParamData to get the */
    /* parameter number set by SQLBindParameter. Call InitUserData.   */
    /* Call GetUserData and SQLPutData repeatedly to get and put all  */
    /* data for the parameter. Call SQLParamData to finish processing */
    /* this parameter and start processing the next parameter.        */
    while (rc == SQL_NEED_DATA) {
            rc = SQLParamData(hstmt, &pToken);
            if (rc == SQL_NEED_DATA) {
                InitUserData(pToken, InitValue);
                while(GetUserData(InitValue,pToken,Data,&StrLen_or_Ind))
                    SQLPutData(hstmt, Data, StrLen_or_Ind);
            }
    }
/* commit the transaction. */
SQLEndTran(SQL_HANDLE_ENV, henv, SQL_COMMIT);
/* free the Statement handle */
SQLFreeHandle(SQL_HANDLE_STMT, hstmt);
/* disconnect from the database */
SQLDisconnect(hdbc);
```

```
/* free the Connection handle */
SQLFreeHandle(SQL_HANDLE_DBC, hdbc);
/* free the Environment handle */
SQLFreeEnv(SQL_HANDLE_ENV, henv);
return(0);
}
/*******************************************************************
The following functions are given for completeness, but are not
relevant for understanding the database processing nature of CLI
*******************************************************************/
void InitUserData(SQLSMALLINT sParam, SQLPOINTER InitValue)
{
SQLCHAR  szPhotoFile[MAX_FILE_NAME_LENGTH];
switch sParam {
    case 3:
        /* Prompt user for bitmap file containing employee photo.    */
        /* OpenPhotoFile opens the file and returns the file handle. */
        PromptPhotoFileName(szPhotoFile);
        OpenPhotoFile(szPhotoFile, (FILE *)InitValue);
        break;
}
}
SQLSMALLINT GetUserData(SQLPOINTER InitValue,
                        SQLSMALLINT sParam,
                        SQLCHAR *Data,
                        SQLINTEGER *StrLen_or_Ind)
{
switch sParam {
    case 1:
        /* Prompt user for employee name. */
        PromptEmployeeName(Data);
        *StrLen_or_Ind = SQL_NTS;
        return (1);
    case 3:
       /* GetNextPhotoData returns the next piece of photo data and  */
       /* the number of octets of data returned (up to MAX_DATA_LENGTH). */
       Done = GetNextPhotoData((FILE *)InitValue, Data,
                               MAX_DATA_LENGTH, StrLen_or_Ind);
       if (Done) {
            ClosePhotoFile((FILE *)InitValue);
                return(1);
       }
       return(0);
}
return(0);
}
```

Annex C
(informative)

Implementation-defined elements

This Annex modifies Annex B, “Implementation-defined elements”, in ISO/IEC 9075-2.

This Annex references those features that are identified in the body of this part of ISO/IEC 9075 as implementation-defined.

1) Subclause 4.3, “Diagnostics areas in SQL/CLI”: If the routine's return code indicates **No data found**, then no status record is generated corresponding to SQLSTATE value '02000' but there may be status records generated corresponding to SQLSTATE value '02*nnn*', where '*nnn*' is an implementation-defined subclass value.

2) Subclause 4.4.1, “Handles”: The validity of a handle in a compilation unit other than the one in which the identified resource was allocated is implementation-defined.

3) Subclause 4.4.2, “Null terminated strings”: The null character that terminates C character strings is implementation-defined.

4) Subclause 5.1, “<CLI routine>”:

   a) It is implementation-defined which of the invocation of *CF* or the invocation of *CP* is supported.

   b) The <implementation-defined CLI generic name> for an implementation-defined CLI function shall be different from the <CLI generic name> of any other CLI function. The <implementation-defined CLI generic name> for an implementation-defined CLI procedure shall be different from the <CLI generic name> of any other CLI procedure.

5) Subclause 5.2, “<CLI routine> invocation”:

   a) If the value of any input argument provided by the host program falls outside the set of allowed values of the data type of the parameter, or if the value of any output argument resulting from the execution of the <CLI routine> falls outside the set of values supported by the host program for that parameter, then the effect is implementation-defined.

   b) If *RN* did not execute successfully, then one or more exception conditions may be raised as determined by implementation-defined rules.

6) Subclause 5.4, “Implicit cursor”:

   a) The visibility of significant changes through a sensitive holdable cursor during a subsequent SQL-transaction is implementation-defined.

   b) Whether an implementation is able to disallow significant changes that would not be visible through a currently open cursor is implementation-defined.

7) Subclause 5.5, “Implicit DESCRIBE USING clause”:

a) The null character that terminates C character strings is implementation-defined.

b) If the value of COUNT for *IPD* is greater than *D*, then it is implementation-defined whether an exception condition is raised: *dynamic SQL error — using clause does not match dynamic parameter specifications*.

8) Subclause 5.6, "Implicit EXECUTE USING and OPEN USING clauses":

a) Let *NIDAL* be the number of item descriptor areas in *IPD* for which LEVEL is 0 (zero). If *NIDAL* is greater than *D*, then it is implementation-defined whether an exception condition is raised: *dynamic SQL error — using clause does not match dynamic parameter specifications*.

b) The null character that terminates C character strings is implementation-defined.

c) There may be an implementation-defined conversion from type *SDT* to type *TDT*.

d) There may be an implementation-defined conversion from type *SDT* to type *UDT*.

9) Subclause 5.7, "Implicit CALL USING clause":

a) If the result is a zero-length character string, then it is implementation-defined whether or not an exception condition is raised: *data exception — zero-length character string*.

b) The maximum length of a variable-length character string is implementation-defined.

c) There may be an implementation-defined conversion from type *SDT* to type *TDT*.

10) Subclause 5.8, "Implicit FETCH USING clause":

a) If separate fetches for the same bound target are inconsistent in whether a locator is used, then it is implementation-defined whether an exception condition is raised: *dynamic SQL error — restricted data type attribute violation*.

b) If the result is a zero-length character string, then it is implementation-defined whether or not an exception condition is raised: *data exception — zero-length character string*.

c) The maximum length of a variable-length character string is implementation-defined.

d) There may be an implementation-defined conversion from type *SDT* to type *TDT*.

11) Subclause 5.9, "Character string retrieval": The null character that terminates C character strings is implementation-defined.

12) Subclause 5.13, "Description of CLI item descriptor areas":

a) The null character that terminates C character strings is implementation-defined.

b) Let *IDA* be an item descriptor area in an implementation parameter descriptor. One condition that allows *IDA* to be *valid* is if TYPE indicates an implementation-defined data type.

c) One condition that allows a CLI item descriptor area in a CLI descriptor area that is not an implementation row descriptor to be *consistent* is if TYPE indicates an implementation-defined data type.

d) Let *IDA* be an item descriptor area in an application parameter descriptor. One condition that allows *IDA* to be *valid* is if TYPE indicates an implementation-defined data type.

e) One condition that allows a CLI item descriptor area in an application row descriptor to be *valid* is if TYPE indicates an implementation-defined data type.

13) Subclause 6.3, “AllocHandle”:

    a) If *HT* indicates ENVIRONMENT HANDLE and the resources to manage an SQL-environment cannot be allocated for implementation-defined reasons, then an implementation-defined exception condition is raised.

    b) If *HT* indicates CONNECTION HANDLE, STATEMENT HANDLE, or DESCRIPTOR HANDLE and the resources to manage an SQL-connection, SQL-statement, or CLI descriptor area, respectively, cannot be allocated for implementation-defined reasons, then OutputHandle is set to zero and an implementation-defined exception condition is raised.

14) Subclause 6.5, “BindCol”: Restrictions on the differences allowed between *ARD* and *IRD* are implementation-defined, except as specified in the General Rules of Subclause 5.8, “Implicit FETCH USING clause”, and the General Rules of Subclause 6.30, “GetData”.

15) Subclause 6.6, “BindParameter”: Restrictions on the differences allowed between *APD* and *IPD* are implementation-defined, except as specified in the General Rules of Subclause 5.6, “Implicit EXECUTE USING and OPEN USING clauses”, Subclause 5.7, “Implicit CALL USING clause”, and the General Rules of Subclause 6.49, “ParamData”.

16) Subclause 6.9, “ColAttribute”: The maximum length of a variable-length character string is implementation-defined.

17) Subclause 6.10, “ColumnPrivileges”:

    a) The maximum length of a variable-length character string is implementation-defined.

    b) If the value of Supported that is returned by the execution of GetFeatureInfo with FeatureType = 'FEATURE' and FeatureId = 'C041' (corresponding to the feature “Information Schema metadata constrained by privileges”) is not 1 (one), then *COLUMN_PRIVILEGES_QUERY* contains a row for each row describing a column in *SS*'s Information Schema COLUMN_PRIVILEGES view that meets implementation-defined authorization criteria.

    c) The null character that terminates C character strings is implementation-defined.

18) Subclause 6.11, “Columns”:

    a) The maximum length of a variable-length character string is implementation-defined.

    b) If the value of Supported that is returned by the execution of GetFeatureInfo with FeatureType = 'FEATURE' and FeatureId = 'C041' (corresponding to the feature “Information Schema metadata constrained by privileges”) is not 1 (one), then *COLUMNS_QUERY* contains a row for each row describing a column in *SS*'s Information Schema COLUMNS view that meets implementation-defined authorization criteria.

    c) For each row of *COLUMNS_QUERY*, the value of TYPE_NAME in *COLUMNS_QUERY* is an implementation-defined value that is the character string by which the data type is known at the data source.

    d) If the value of DATA_TYPE in the COLUMNS view is 'SMALLINT', 'INTEGER', 'BIGINT', 'FLOAT', 'REAL', or 'DOUBLE PRECISION', then the value of COLUMN_SIZE in *COLUMNS_QUERY* is implementation-defined.

    e) The value of BUFFER_LENGTH in COLUMNS_QUERY is implementation-defined.

    f) The value of REMARKS in *COLUMNS_QUERY* is an implementation-defined description of the column.

g) The null character that terminates C character strings is implementation-defined.

19) Subclause 6.12, “Connect”:

a) The maximum length of a variable-length character string is implementation-defined.

b) The length of the Authentication parameter is implementation-defined.

c) The null character that terminates C character strings is implementation-defined.

d) If the value of ServerName is not 'DEFAULT' and the length of that value is zero, then an implementation-defined <authorization identifier> is provided.

e) If the value of ServerName is not 'DEFAULT' and the length of that value is not zero, then that value may have implementation-defined restrictions on its value.

f) If length of the value of Authentication is zero, then an implementation-defined <user identifier> is provided.

g) The method by which a default SQL-server is determined is implementation-defined.

h) The method by which the value of ServerName is used to determine the appropriate SQL-server is determined is implementation-defined.

i) If *UN* does not conform to any implementation-defined restrictions on its value, then an exception condition is raised: *invalid authorization specification*.

20) Subclause 6.14, “DataSources”:

a) The maximum length of a variable-length character string is implementation-defined.

b) The mechanism used to establish the set of names of SQL-servers to which the SQL/CLI application might be eligible to connect is implementation-defined.

c) The mechanism used to establish the strings describing the set SQL-servers to which the SQL/CLI application might be eligible to connect is implementation-defined.

21) Subclause 6.15, “DescribeCol”: The maximum length of a variable-length character string is implementation-defined.

22) Subclause 6.17, “EndTran”:

a) If any other error preventing commitment of the SQL-transaction has occurred, then any changes to SQL-data or schemas that were made by the current SQL-transaction are canceled and an exception condition is raised: *transaction rollback* with an implementation-defined subclass value.

b) The status of any open cursors in *L3* that were opened by the current SQL-transaction before the establishment of *SP* is implementation-defined.

23) Subclause 6.18, “Error”: The maximum length of a variable-length character string is implementation-defined.

24) Subclause 6.19, “ExecDirect”:

a) The maximum length of a variable-length character string is implementation-defined.

b) The null character that terminates C character strings is implementation-defined.

c) If *P* is a <preparable dynamic delete statement: positioned> and the execution of *P* deleted the current row of *CR*, then the effect on the fetched row, if any, associated with the allocated SQL-statement under which that current row was established, is implementation-defined.

d) If *P* is a <preparable dynamic update statement: positioned> and the execution of *P* updated the current row of *CR*, then the effect on the fetched row, if any, associated with the allocated SQL-statement under which that current row was established, is implementation-defined.

25) Subclause 6.20, “Execute”:

a) If *P* is a <preparable dynamic delete statement: positioned> and the execution of *P* deleted the current row of *CR*, then the effect on the fetched row, if any, associated with the allocated SQL-statement under which that current row was established, is implementation-defined.

b) If *P* is a <preparable dynamic update statement: positioned> and the execution of *P* updated the current row of *CR*, then the effect on the fetched row, if any, associated with the allocated SQL-statement under which that current row was established, is implementation-defined.

26) Subclause 6.23, “ForeignKeys”:

a) The maximum length of a variable-length character string is implementation-defined.

b) If the value of Supported that is returned by the execution of GetFeatureInfo with FeatureType = 'FEATURE' and FeatureId = 'C041' (corresponding to the feature “Information Schema metadata constrained by privileges”) is not 1 (one), then *FOREIGN_KEYS_QUERY* contains a row for each row describing a column in *SS*'s Information Schema TABLE_CONSTRAINT view that meets implementation-defined authorization criteria.

c) The null character that terminates C character strings is implementation-defined.

d) If there are no implementation-defined mechanisms for setting the value of DEFERABILITY in *FOREIGN_KEYS_QUERY* to the value of the code for INITIALLY DEFERRED or to the value of the code for INITIALLY IMMEDIATE in Table 27, “Miscellaneous codes used in CLI”, then the value of DEFERABILITY in *FOREIGN_KEYS_QUERY* is the code for NOT DEFERRABLE in Table 27, “Miscellaneous codes used in CLI”; otherwise, the value of DEFERABILITY in *FOREIGN_KEYS_QUERY* can be the code for INITIALLY DEFERRED, the value of the code for INITIALLY IMMEDIATE, or the code for NOT DEFERRABLE in Table 27, “Miscellaneous codes used in CLI”.

e) If CHAR_LENGTH(*PKN*) ≠ 0 (zero) and CHAR_LENGTH(*FKN*) ≠ 0 (zero), then the result of the routine is implementation-defined.

27) Subclause 6.28, “GetConnectAttr”: The value of Attribute might specify an implementation-defined connection attribute.

28) Subclause 6.29, “GetCursorName”: The maximum length of a variable-length character string is implementation-defined.

29) Subclause 6.30, “GetData”:

a) If *CN* is not greater than *HBCN* and the DATA_POINTER field of *IDA* is zero, then it is implementation-defined whether an exception condition is raised: *dynamic SQL error — invalid descriptor index*. That is, it is implementation-defined whether columns with a lower column number than that of the highest bound column can be accessed by GetData.

b) If *FCN* is greater than zero and *CN* is not greater than *FCN*, then it is implementation-defined whether an exception condition is raised: *dynamic SQL error — invalid descriptor index*. That is, it is implementation-defined whether GetData can only access columns in ascending column number order.

c) If *FCN* is less than zero and *CN* is less than *FCN*, then it is implementation-defined whether an exception condition is raised: *dynamic SQL error — invalid descriptor index*.

d) The maximum length of a variable-length character string is implementation-defined.

e) If separate retrievals for the same <target specification> are inconsistent in whether a locator is used, then it is implementation-defined whether an exception condition is raised: *dynamic SQL error — restricted data type attribute violation*.

f) If a zero-length character string is fetched, then it is implementation-defined whether or not an exception condition is raised: *data exception — zero-length character string*.

g) There may be an implementation-defined conversion from type *SDT* to type *TDT*.

30) Subclause 6.31, “GetDescField”:

a) If *TYPE* is 'HEADER', then header information from the descriptor area *D* is retrieved; if *FI* indicates an implementation-defined descriptor header field, then the value retrieved is the value of the implementation-defined descriptor header field identified by *FI*.

b) If *TYPE* is 'ITEM', then item information from the descriptor area *D* is retrieved; if *FI* indicates an implementation-defined descriptor item field, then the value retrieved is the value of the implementation-defined descriptor item field of *IDA* identified by *FI*.

31) Subclause 6.32, “GetDescRec”: The maximum length of a variable-length character string is implementation-defined.

32) Subclause 6.33, “GetDiagField”:

a) If *TYPE* is 'HEADER' and *DI* indicates an implementation-defined diagnostics header field, then the value retrieved is the value of the implementation-defined diagnostics header field.

b) If *TYPE* is 'STATUS' and *DI* indicates an implementation-defined diagnostics header field, then the value retrieved is the value of the implementation-defined diagnostics header field.

c) If *TYPE* is 'STATUS' and *DI* indicates NATIVE_CODE, then the value retrieved is the implementation-defined native error code corresponding to the status condition.

d) If *TYPE* is 'STATUS', *DI* indicates MESSAGE_TEXT, and the value of SQLSTATE does not correspond to *external routine invocation exception*, *external routine exception*, or *warning*, then the value retrieved is an implementation-defined character string.

e) If *TYPE* is 'STATUS' and *DI* indicates CLASS_ORIGIN, then the value retrieved shall be an implementation-defined character string other than 'ISO 9075' for any implementation-defined class value.

f) If *TYPE* is 'STATUS' and *DI* indicates SUBCLASS_ORIGIN, then the value retrieved shall be an implementation-defined character string other than 'ISO 9075' for any implementation-defined subclass value.

g) If *TYPE* is 'STATUS', and *DI* indicates SERVER_NAME or CONNECTION_NAME, and *R* is Connect, then the values retrieved are the name of the SQL-server explicitly or implicitly referenced by *R* and the implementation-defined connection name associated with that SQL-server reference, respectively.

h) If *TYPE* is 'STATUS', and *DI* indicates SERVER_NAME or CONNECTION_NAME, and *R* is Disconnect, then the values retrieved are the name of the SQL-server and the associated implementation-defined connection name, respectively, associated with the allocated SQL-connection referenced by *R*.

i) If *TYPE* is 'STATUS', and *DI* indicates SERVER_NAME or CONNECTION_NAME, and the status condition was caused by the SQL/CLI application of the General Rules of Subclause 5.3, "Implicit set connection", then the values retrieved are the name of the SQL-server and the implementation-defined connection name, respectively, associated with the dormant SQL-connection specified in the application of that Subclause.

j) If *TYPE* is 'STATUS', and *DI* indicates SERVER_NAME or CONNECTION_NAME, and the status condition was raised in an SQL-session, then the values retrieved are name of the SQL-server and the implementation-defined connection name, respectively, associated with the SQL-session in which the status condition was raised.

k) The null character that terminates C character strings is implementation-defined.

33) Subclause 6.34, "GetDiagRec":

a) The maximum length of a variable-length character string is implementation-defined.

b) If NativeError is not a null pointer, then NativeError is set to the implementation-defined native error code corresponding to the status condition.

c) If MessageText is not a null pointer and either null termination is <u>*True*</u> for the current SQL-environment or *BL* is not zero, then an implementation-defined character string is retrieved.

d) The null character that terminates C character strings is implementation-defined.

34) Subclause 6.35, "GetEnvAttr": If the value of Attribute specifies an implementation-defined environment attribute, then Value is set to the value of the implementation-defined environment attribute.

35) Subclause 6.36, "GetFeatureInfo":

a) The maximum length of a variable-length character string is implementation-defined.

b) The null character that terminates C character strings is implementation-defined.

36) Subclause 6.40, "GetParamData":

a) The maximum length of a variable-length character string is implementation-defined.

b) If the DATA_POINTER field of *IDA* is zero, then it is implementation-defined whether an exception condition is raised: *dynamic SQL error — invalid descriptor index*.

c) If *FPN* is greater than zero and *PN* is not greater than *FPN*, then it is implementation-defined whether an exception condition is raised: *dynamic SQL error — invalid descriptor index*.

d) If *PN* is less than *AFPN*, then it is implementation-defined whether an exception condition is raised: *dynamic SQL error — invalid descriptor index*.

e) If a specified <cast specification> does not conform to the Syntax Rules of Subclause 6.12, "<cast specification>", in ISO/IEC 9075-2, and there is an implementation-defined conversion from type *SDT* to type *TDT*, then that implementation-defined conversion is effectively performed, converting *SV* to type *TDT*, and the result is the value *TV* of the *PN*-th <target specification>.

f) If *TV* is a zero-length character string, then it is implementation-defined whether or not an exception condition is raised: *data exception — zero-length character string*.

g) There may be an implementation-defined conversion from type *SDT* to type *TDT*.

37) Subclause 6.43, “GetStmtAttr”: If the value of Attribute specifies an implementation-defined statement attribute, then Value is set to the value of the implementation-defined statement attribute.

38) Subclause 6.44, “GetSubString”: If the result is a zero-length character string, then it is implementation-defined whether or not an exception condition is raised: *data exception — zero-length character string*.

39) Subclause 6.45, “GetTypeInfo”:

a) For all supported data types for which more than one name is supported, it is implementation-defined whether *TYPE_INFO* contains a single row or a row for each supported name.

b) If multiple names are supported for this data type and TYPE_INFO contains only a single row for this data type, then it is implementation-defined which of the names is in TYPE_NAME.

c) The value of COLUMN_SIZE is an implementation-defined value for an implementation-defined data type that has a length or a precision.

d) The value of CREATE_PARAMS is a comma-separated list of specifiable attributes for the data type; the appearance of attributes in implementation-defined data types is implementation-defined.

e) The value of CASE_SENSITIVE is 1 (one) if the data type is a character string type and the default collation for its implementation-defined implicit character set would result in a case sensitive comparison when two values with this data type are compared.

f) The value of LOCAL_TYPE_NAME is an implementation-defined localized representation of the name of the data type.

40) Subclause 6.46, “MoreResults”: If there is no cursor associated with *S* and there exists an implementation-defined capability to support that situation, then implementation-defined rules are evaluated and no further General Rules of this Subclause are evaluated.

41) Subclause 6.49, “ParamData”:

a) If *DPN* is equal to *HPN*, and there is not a select source associated with *S*, and *SS* is either a <preparable dynamic delete statement: positioned> or a <preparable dynamic update statement: positioned>, and the execution of *SS* deleted or updated, respectively, the current row of *CR*, then the effect on the fetched row, if any, associated with the allocated SQL-statement under which that current row was established, is implementation-defined.

b) If the result is a zero-length character string, then it is implementation-defined whether or not an exception condition is raised: *data exception — zero-length character string*.

c) If a specified <cast specification> does not conform to the Syntax Rules of Subclause 6.12, “<cast specification>”, in ISO/IEC 9075-2, and there is an implementation-defined conversion from type *SDT* to type *TDT*, then that implementation-defined conversion is effectively performed, converting *SV* to type *TDT*, and the result is the value *TV* of the *i*-th bound target.

d) The visibility of significant changes through a sensitive holdable cursor during a subsequent SQL-transaction is implementation-defined.

e) Whether an implementation is able to disallow significant changes that would not be visible through a currently open cursor is implementation-defined.

f) The maximum length of a fixed-length character string is implementation-defined.

g) The maximum length of a large object character string is implementation-defined.

h) The maximum length of a fixed-length binary string is implementation-defined.

i) The maximum length of a variable-length binary string is implementation-defined.

j) The maximum length of a binary large object is implementation-defined.

k) There may be an implementation-defined conversion from type *SDT* to type *TDT*.

l) There may be an implementation-defined conversion from type *SDT* to type *UDT*.

42) Subclause 6.50, “Prepare”:

a) The maximum length of a variable-length character string is implementation-defined.

b) The null character that terminates C character strings is implementation-defined.

43) Subclause 6.51, “PrimaryKeys”:

a) The maximum length of a variable-length character string is implementation-defined.

b) If the value of Supported that is returned by the execution of GetFeatureInfo with FeatureType = 'FEATURE' and FeatureId = 'C041' (corresponding to the feature “Information Schema metadata constrained by privileges”) is not 1 (one), then *PRIMARY_KEYS_QUERY* contains a row for each row describing a column in *SS*'s Information Schema TABLE_CONSTRAINT view that meets implementation-defined authorization criteria.

c) The null character that terminates C character strings is implementation-defined.

44) Subclause 6.52, “PutData”: The null character that terminates C character strings is implementation-defined.

45) Subclause 6.54, “SetConnectAttr”:

a) The null character that terminates C character strings is implementation-defined.

b) If the value of Attribute specifies an implementation-defined connection attribute, then the connection attribute is set to the value of Value.

46) Subclause 6.55, “SetCursorName”:

a) The maximum length of a variable-length character string is implementation-defined.

b) The null character that terminates C character strings is implementation-defined.

47) Subclause 6.56, “SetDescField”:

a) If *FI* indicates TYPE and *V* indicates NUMERIC or DECIMAL, then the SCALE field of *IDA* is set to 0 (zero) and the PRECISION field of *IDA* is set to the implementation-defined default value for the precision of NUMERIC or DECIMAL data types, respectively.

b) If *FI* indicates TYPE and *V* indicates FLOAT, then the PRECISION field of *IDA* is set to the implementation-defined default value for the precision of the FLOAT data type.

c) Restrictions on the differences allowed between implementation and application parameter descriptors are implementation-defined, except as specified in the General Rules of Subclause 5.6, “Implicit EXECUTE USING and OPEN USING clauses”, the General Rules of Subclause 5.7, “Implicit CALL

USING clause", and in the General Rules of Subclause 6.49, "ParamData". Restrictions on the differences between the implementation and application row descriptors are implementation-defined, except as specified in the General Rules of Subclause 5.8, "Implicit FETCH USING clause", and the General Rules of Subclause 6.30, "GetData".

d) The null character that terminates C character strings is implementation-defined.

e) If *FI* indicates TYPE and *V* indicates SMALLINT, INTEGER, or BIGINT, then the SCALE field of *IDA* is set to 0 (zero) and the PRECISION field of *IDA* is set to the implementation-defined value for the precision of the SMALLINT, INTEGER, or BIGINT data types, respectively.

f) If *FI* indicates TYPE and *V* indicates REAL or DOUBLE PRECISION, then the PRECISION field of *IDA* is set to the implementation-defined value for the precision of the REAL or DOUBLE PRECISION data types, respectively.

g) If *FI* indicates TYPE and *V* indicates an implementation-defined data type, then an implementation-defined set of fields of *IDA* are set to implementation-defined default values.

48) Subclause 6.57, "SetDescRec": Restrictions on the differences allowed between implementation and application parameter descriptors are implementation-defined, except as specified in the General Rules of Subclause 5.6, "Implicit EXECUTE USING and OPEN USING clauses", the General Rules of Subclause 5.7, "Implicit CALL USING clause", and in the General Rules of Subclause 6.49, "ParamData". Restrictions on the differences between the implementation and application row descriptors are implementation-defined, except as specified in the General Rules of Subclause 5.8, "Implicit FETCH USING clause", and the General Rules of Subclause 6.30, "GetData".

49) Subclause 6.58, "SetEnvAttr":

a) If the value of Attribute specifies an implementation-defined environment attribute, then the environment attribute is set to the value of Value.

b) The null character that terminates C character strings is implementation-defined.

50) Subclause 6.59, "SetStmtAttr":

a) If the value of Attribute specifies an implementation-defined statement attribute, then the statement attribute is set to the value of Value.

b) The null character that terminates C character strings is implementation-defined.

51) Subclause 6.60, "SpecialColumns":

a) The maximum length of a variable-length character string is implementation-defined.

b) If the value of Supported that is returned by the execution of GetFeatureInfo with FeatureType = 'FEATURE' and FeatureId = 'C041' (corresponding to the feature "Information Schema metadata constrained by privileges") is not 1 (one), then *SPECIAL_COLUMNS_QUERY* contains a row for each row describing a column in *SS*'s Information Schema SPECIAL_COLUMNS view that meets implementation-defined authorization criteria.

c) The null character that terminates C character strings is implementation-defined.

d) The value of IdentifierType may be an implementation-defined extension to Table 39, "Column types and scopes used with SpecialColumns".

e) The value of SCOPE may be an implementation-defined value.

f) The value of TYPE_NAME in *SPECIAL_COLUMNS_QUERY* is an implementation-defined value that is the character string by which the data type is known at the data source.

g) If the value of DATA_TYPE in the COLUMNS view is 'SMALLINT', 'INTEGER', 'BIGINT', 'FLOAT', 'REAL', or 'DOUBLE PRECISION', then the value of COLUMN_SIZE in *SPECIAL_COLUMNS_QUERY* is implementation-defined.

h) The value of BUFFER_LENGTH in *SPECIAL_COLUMNS_QUERY* is implementation-defined.

52) Subclause 6.61, “StartTran”: The isolation level that is set for a transaction is an implementation-defined isolation level that will not exhibit any of the phenomena that the explicit or implicit <level of isolation> would not exhibit, as specified in Table 8, “SQL-transaction isolation levels and the three phenomena”, in ISO/IEC 9075-2.

53) Subclause 6.62, “TablePrivileges”:

a) The maximum length of a variable-length character string is implementation-defined.

b) If the value of Supported that is returned by the execution of GetFeatureInfo with FeatureType = 'FEATURE' and FeatureId = 'C041' (corresponding to the feature “Information Schema metadata constrained by privileges”) is not 1 (one), then *TABLE_PRIVILEGES_QUERY* contains a row for each row describing a column in *SS*'s Information Schema TABLE_PRIVILEGES view that meets implementation-defined authorization criteria.

c) The null character that terminates C character strings is implementation-defined.

54) Subclause 6.63, “Tables”:

a) The maximum length of a variable-length character string is implementation-defined.

b) If the value of Supported that is returned by the execution of GetFeatureInfo with FeatureType = 'FEATURE' and FeatureId = 'C041' (corresponding to the feature “Information Schema metadata constrained by privileges”) is not 1 (one), then *TABLES_QUERY* contains a row for each row describing a column in *SS*'s Information Schema TABLES view that meets implementation-defined authorization criteria.

c) The null character that terminates C character strings is implementation-defined.

d) If the value of TABLE_TYPE in the TABLES view is niether 'VIEW', 'BASE TABLE', nor 'GLOBAL TEMPORARY', then the value of TABLE_TYPE in *TABLES_QUERY* is an implementation-defined value.

e) The value of REMARKS in *TABLES_QUERY* is an implementation-defined description of the table.

f) Implementation-defined table types may be defined.

55) Subclause 7.1, “SQL_IMPLEMENTATION_INFO base table”: Implementation-defined items that are represented in this table shall have an IMPLEMENTATION_INFO_ID value that is in the range 11000 through 14999, inclusive.

56) Subclause 7.2, “SQL_SIZING base table”: Implementation-defined items that are represented in this table shall have a SIZING_ID value that is in the range 15000 through 19999, inclusive.

57) Table 1, “Header fields in SQL/CLI diagnostics areas”:

a) The maximum lengths of CLI diagnostics area fields whose data type is CHARACTER VARYING are implementation-defined.

b) SQL/CLI supports implementation-defined header fields in CLI diagnostics areas.

58) Table 2, “Status record fields in SQL/CLI diagnostics areas”:

a) The maximum lengths of CLI diagnostics area fields whose data type is CHARACTER VARYING are implementation-defined.

59) Table 4, “Abbreviated SQL/CLI generic names”: SQL/CLI supports implementation-defined CLI routines.

60) Table 6, “Fields in SQL/CLI row and parameter descriptor areas”:

a) The maximum lengths of CLI item descriptor fields whose data type is CHARACTER VARYING are implementation-defined.

b) SQL/CLI supports implementation-defined header fields and implementation-defined item fields in row and parameter descriptor areas.

61) Table 7, “Codes used for implementation data types in SQL/CLI”: SQL/CLI supports implementation-defined implementation data types as specified in this table.

62) Table 8, “Codes used for application data types in SQL/CLI”: SQL/CLI supports implementation-defined application data types as specified in this table.

63) Table 13, “Codes used for SQL/CLI diagnostic fields”: SQL/CLI supports implementation-defined diagnostics header fields and implementation-defined diagnostics status fields.

64) Table 14, “Codes used for SQL/CLI handle types”: SQL/CLI supports implementation-defined handle types.

65) Table 15, “Codes used for transaction termination”: SQL/CLI supports implementation-defined transaction termination types.

66) Table 16, “Codes used for environment attributes”: SQL/CLI supports implementation-defined environment attributes.

67) Table 17, “Codes used for connection attributes”: SQL/CLI supports implementation-defined connection attributes.

68) Table 18, “Codes used for statement attributes”: SQL/CLI supports implementation-defined statement attributes.

69) Table 22, “Ability to set SQL/CLI descriptor fields”:

a) “ID” means that it is implementation-defined whether or not the descriptor field is settable.

b) SQL/CLI supports implementation-defined descriptor header fields and implementation-defined descriptor item fields.

70) Table 23, “Ability to retrieve SQL/CLI descriptor fields”:

a) “ID” means that it is implementation-defined whether or not the descriptor field is retrievable.

b) SQL/CLI supports implementation-defined descriptor header fields and implementation-defined descriptor item fields.

71) Table 24, “SQL/CLI descriptor field default values”:

a) “ID” means that the descriptor field's default value is implementation-defined.

b) SQL/CLI supports implementation-defined descriptor header fields and implementation-defined descriptor item fields.

72) Table 28, “Codes used to identify SQL/CLI routines”: SQL/CLI supports implementation-defined CLI routines.

73) Table 29, “Codes and data types for implementation information”:

a) SQL/CLI supports implementation-defined information types with implementation-defined codes and implementation-defined data types as specified in this table.

b) The maximum length of a variable-length character string is implementation-defined.

74) Table 30, “Codes and data types for session implementation information”: The maximum lengths of the session implementation information items are the implementation-defined maximum lengths of the corresponding <general value specification>s.

75) Table 33, “Codes used for concise data types”: SQL/CLI supports implementation-defined data types as specified in this table.

76) Table 42, “SQL/CLI data type correspondences for COBOL”:

a) The number of '9's in a PICTURE clause describing CHARACTER LARGE OBJECT LOCATOR, BINARY LARGE OBJECT LOCATOR, SMALLINT, INTEGER, BIGINT, USER-DEFINED TYPE LOCATOR, ARRAY LOCATOR, and MULTISET LOCATOR data types in the COBOL host language is implementation-defined.

77) Table 46, “SQL/CLI data type correspondences for PL/I”:

a) The number of '9's in a FIXED BINARY clause describing CHARACTER LARGE OBJECT LOCATOR, BINARY LARGE OBJECT LOCATOR, SMALLINT, INTEGER, BIGINT, USER-DEFINED TYPE LOCATOR, ARRAY LOCATOR, and MULTISET LOCATOR data types in the PL/I host language is implementation-defined.

(Blank page)

Annex D
(informative)

Implementation-dependent elements

This Annex modifies Annex C, "Implementation-dependent elements", in ISO/IEC 9075-2.

This Annex references those places where this part of ISO/IEC 9075 states explicitly that the actions of a conforming implementation are implementation-dependent.

1) Subclause 4.1, "Introduction to SQL/CLI": When a <dynamic select statement> or <dynamic single row select statement> is executed, a cursor is implicitly declared and opened; if a cursor name is not supplied by the SQL/CLI application, an implementation-dependent cursor name is generated.

2) Subclause 4.2, "Return codes": After the execution of a CLI routine, the values of all output arguments not explicitly defined by this part of ISO/IEC 9075 are implementation-dependent.

3) Subclause 4.3, "Diagnostics areas in SQL/CLI": If multiple status records are generated, then the order in which status records are placed in a diagnostics area is implementation-dependent, with two exceptions.

4) Subclause 4.5, "Client-server operation": If the execution of a CLI routine causes the implicit or explicit execution of an <SQL procedure statement> by an SQL-server, diagnostic information is passed in an implementation-dependent manner to the SQL-client and then into the appropriate diagnostics area. The effect on diagnostic information of incompatibilities between the character repertoires supported by the SQL-client and the SQL-server is implementation-dependent.

5) Subclause 5.5, "Implicit DESCRIBE USING clause":

   a) If *D* is not zero, then those fields and fields that are not applicable for a particular value of TYPE are set to implementation-dependent values.

   b) If *D* is not zero and the column name is implementation-dependent, then NAME is set to the implementation-dependent name of the column and UNNAMED is set to 1 (one).

   c) If POPULATE IPD for *C* is *True* and *D* is not zero, then those fields and fields that are not applicable for a particular value of TYPE are set to implementation-dependent values and NAME is set to an implementation-dependent value.

   d) If the name of the field is implementation-dependent, then NAME is set to the implementation-dependent name of the field and UNNAMED is set to 1 (one).

6) Subclause 5.7, "Implicit CALL USING clause":

   a) If *TDT* is a locator type and *SV* is not the null value, then a locator *L* that uniquely identifies *SV* is generated and the value *TV* of the *i*-th bound target is set to an implementation-dependent four-octet value that represents *L*.

   b) If TYPE indicates ROW, *TV* is the null value, and *IP* is neither a null pointer for *IDA* nor for any of the subordinate descriptor areas of *IDA* that are not subordinate to an item descriptor area whose type indicates ARRAY, ARRAY LOCATOR, MULTISET, or MULTISET LOCATOR, then the value of the host variable addressed by *IP* for *IDA*, and that in all subordinate descriptor areas of *IDA* that are not subordinate to an item descriptor area whose TYPE indicates ARRAY, ARRAY LOCATOR, MULTISET, or MULTISET LOCATOR, is set to the appropriate 'Code' for SQL NULL DATA in Table 27, “Miscellaneous codes used in CLI”, and the values of variables addressed by *DP* and *LP* are implementation-dependent.

c) If TYPE does not indicate ROW,, *TV* is the null value and *IP* is not a null pointer, then the value of the host variable addressed by *IP* is set to the appropriate 'Code' for SQL NULL DATA in Table 27, “Miscellaneous codes used in CLI”, and the values of the host variables addressed by *DP* and *LP* are implementation-dependent.

7) Subclause 5.8, “Implicit FETCH USING clause”:

a) If *TDT* is a locator type and *SV* is not the null value, then a locator *L* that uniquely identifies *SV* is generated and the value *TV* of the *i*-th bound target is set to an implementation-dependent four-octet value that represents *L*.

b) If TYPE indicates ROW, *TV* is the null value, and *IPE* is not a null pointer for *IDA* nor for any of the subordinate descriptor areas of *IDA* that are not subordinate to an item descriptor area whose type indicates ARRAY, ARRAY LOCATOR, MULTISET, or MULTISET LOCATOR, then the value of the host variable addressed by *IPE* for *IDA*, and that in all subordinate descriptor areas of *IDA* that are not subordinate to an item descriptor area whose TYPE indicates ARRAY, ARRAY LOCATOR, MULTISET, or MULTISET LOCATOR, is set to the appropriate 'Code' for SQL NULL DATA in Table 27, “Miscellaneous codes used in CLI”, and the values of variables addressed by *DPE* and *LPE* are implementation-dependent.

c) If TYPE does not indicate ROW, *TV* is the null value, and *IPE* is not a null pointer, then the value of the host variable addressed by *IPE* is set to the appropriate 'Code' for SQL NULL DATA in Table 27, “Miscellaneous codes used in CLI”, and the values of the host variables addressed by *DPE* and *LPE* are implementation-dependent.

8) Subclause 5.9, “Character string retrieval”:

a) If null termination is *False* for the current SQL-environment and *L* is not greater than *TL*, then the first *L* octets of *T* are set to *V* and the values of the remaining octets of *T* are implementation-dependent.

b) If null termination is *True* for the current SQL-environment and *L* is not greater than (*TL*–*NB*), then the first (*L*+*NB*) octets of *T* are set to *V* concatenated with a single implementation-defined null character that terminates a C character string and the values of the remaining characters of *T* are implementation-dependent.

9) Subclause 5.10, “Binary string retrieval”: If *L* is not greater than *TL*, then the first *L* octets of *T* are set to *V* and the values of the remaining octets of *T* are implementation-dependent.

10) Subclause 6.5, “BindCol”: If an exception condition is raised, then the TYPE, OCTET_LENGTH, LENGTH, DATA_POINTER, INDICATOR_POINTER, and OCTET_LENGTH_POINTER fields of *IDA* are set to implementation-dependent values.

11) Subclause 6.6, “BindParameter”: If an exception condition is raised, then The TYPE, LENGTH, PRECISION, and SCALE fields of *IDA1* are set to implementation-dependent values and the TYPE, DATA_POINTER, INDICATOR_POINTER, and OCTET_LENGTH_POINTER fields of *IDA2* are set to implementation-dependent values.

12) Subclause 6.7, “Cancel”: The method of passing control between concurrently operating programs is implementation-dependent.

13) Subclause 6.11, “Columns”: The value of COLUMN_SIZE in *COLUMNS_QUERY* is implementation-dependent if the value of DATA_TYPE in the columns view is not among the specified values.

14) Subclause 6.12, “Connect”: *AU* and *UN* are used by the SQL-server, along with other implementation-dependent values, to determined whether to accept or reject the establishment of an SQL-session.

15) Subclause 6.15, “DescribeCol”:

    a) When information is retrieved from *IRD*, if the data type of *C* is neither exact numeric, datetime, nor interval, then DecimalDigits is set to an implementation-dependent value.

    b) If *C* has an implementation-dependent name, then the value retrieved is the implementation-dependent name for *C*.

    c) When information is retrieved from *IRD*, if the data type of *C* is neither exact numeric, approximate numeric, datetime, interval, nor reference type, then ColumnSize is set to an implementation-dependent value.

    d) When information is retrieved from *IRD*, if the data type of *C* is neither character string, exact numeric, approximate numeric, datetime, interval, or a reference type, then ColumnSize is set to an implementation-dependent value.

16) Subclause 6.18, “Error”: If the number of status records generated by the execution of *R* is zero or the number of status records generated by the execution of *R* already processed by Error equals the number of status records generated by the execution of *R*, then a completion condition is raised: *no data*, Sqlstate is set to '00000', the values of NativeError, MessageText, and TextLength are set to implementation-dependent values.

17) Subclause 6.19, “ExecDirect”: If *P* is a <dynamic select statement> or a <dynamic single row select statement> and there is no cursor name associated with *S*, then a unique implementation-dependent name that has the prefix 'SQLCUR' or the prefix 'SQL_CUR' becomes the cursor name associated with *S*.

18) Subclause 6.21, “Fetch”:

    a) If *ROWS_PROCESSED* is greater than 0 (zero), then when the General Rules of Subclause 5.8, “Implicit FETCH USING clause”, are applied with *SS*, *RS*, *ROWS_PROCESSED*, and *S* as *SOURCE*, *ROWS*, *ROWS PROCESSED*, and *ALLOCATED STATEMENT*, respectively, if *ROWS_PROCESSED* is 0 (zero), then the values of all bound targets are implementation-dependent, and *CR* remains positioned on *NR*.

    b) If *ROWS_PROCESSED* is greater than 0 (zero), then the values of all bound targets are implementation-dependent and *CR* remains positioned on *NR*.

19) Subclause 6.22, “FetchScroll”: If a completion condition: *no data* has not been raised, and an exception condition is not raised during derivation of any <derived column> associated with *NR*, but an exception condition occurs during the derivation of any target value, then the values of all the bound targets are implementation-dependent.

20) Subclause 6.29, “GetCursorName”: If there is no cursor name associated with *S*, then a unique implementation-dependent name that has the prefix 'SQLCUR' or the prefix 'SQL_CUR' becomes the cursor name associated with *S*.

21) Subclause 6.30, “GetData”: If the fetched row associated with *S* is empty, then a completion condition is raised: *no data* and TargetValue, StringLength, and StrLen_or_Ind are set to implementation-dependent values.

22) Subclause 6.33, “GetDiagField”:

a) If *TYPE* is 'HEADER', *DI* indicates ROW_COUNT, and *S* is a <delete statement: searched> cpmtaomomg a <search condition>, or an <update statement: searched> containing a <search condition>, then the value retrieved following the execution by *R* of an SQL-statement that does not directly result in the execution of a <delete statement: searched>, <insert statement>, <merge statement>, or <update statement: searched> is implementation-dependent.

b) If null termination is *False* for the current SQL-environment and *L* is not greater than *BL*, then the first *L* octets of DiagInfo are set to *V* and the values of the remaining octets of DiagInfo are implementation-dependent.

c) If null termination is *True* for the current SQL-environment and *L* is not greater than (*BL*–*k*), then the first (*L*+*k*) octets of DiagInfo are set to *V* concatenated with a single implementation-defined null character that terminates a C character string and the values of the remaining characters of DiagInfo are implementation-dependent.

23) Subclause 6.34, “GetDiagRec”:

a) If null termination is *False* for the current SQL-environment and *L* is not greater than *BL*, then the first *L* octets of MessageText are set to *V* and the values of the remaining octets of MessageText are implementation-dependent.

b) If null termination is *True* for the current SQL-environment and *L* is not greater than (*BL*–*k*), then the first (*L*+*k*) octets of MessageText are set to *V* concatenated with a single implementation-defined null character that terminates a C character string and the values of the remaining characters of MessageText are implementation-dependent.

24) Subclause 6.39, “GetLength”: If *SV* contains the null value, and either IndicatorValue is not referenced by a pointer or the value of that pointer is not a null pointer, then the value of StringLength is implementation-dependent.

25) Subclause 6.41, “GetPosition”: If *SRCL* represents the null value and either IndicatorValue is not referenced by a pointer or the value of that pointer is not a null pointer, then the value of all output arguments other than IndicatorValue is implementation-dependent.

26) Subclause 6.44, “GetSubString”: If *SRCL* represents the null value and either IndicatorValue is not referenced by a pointer or the value of that pointer is not a null pointer, then the value of all output arguments other than IndicatorValue is implementation-dependent.

27) Subclause 6.49, “ParamData”: It is implementation-dependent whether the establishment of *TV* occurs at ParamData time or during the preceding invocation of PutData.

28) Subclause 6.50, “Prepare”:

a) If *P* is a <dynamic select statement> or a <dynamic single row select statement> and there is no cursor name associated with *S*, then a unique implementation-dependent name that has the prefix 'SQLCUR' or the prefix 'SQL_CUR' becomes the cursor name associated with *S*.

b) The validity of a prepared statement in an SQL-transaction different from the one in which the statement was prepared is implementation-dependent.

29) Subclause 6.56, “SetDescField”:

a) If *FI* indicates TYPE, then all fields of *IDA* other than those prescribed are set to implementation-dependent values.

b) If *FI* indicates DATETIME_INTERVAL_CODE and the TYPE field of *IDA* indicates a <datetime type>, then all the fields of *IDA* other than DATETIME_INTERVAL_CODE and TYPE are set to implementation-dependent values.

c) If an exception condition is raised, then the field of *IDA* indicated by *FI* is set to an implementation-dependent value.

30) Subclause 6.57, “SetDescRec”: If an exception condition is raised, then all fields of *IDA* for which specific values were provided in the invocation of SetDescRec are set to implementation-dependent values.

31) Subclause 6.60, “SpecialColumns”:

a) *SPECIAL_COLUMNS_QUERY* contains a row for each column that is part of a set of columns that can be used to best uniquely identify a row within the tables listed in *SS*'s Information Schema TABLES view. Some tables may not have such a set of columns. Some tables may have more than one such set, in which case it is implementation-dependent as to which set of columns is chosen. It is implementation-dependent as to whether a column identified for a given table is a pseudo-column.

b) If the value of DATA_TYPE in the COLUMNS view is neither 'CHARACTER', 'CHARACTER VARYING', 'CHARACTER LARGE OBJECT', 'BINARY', 'BINARY VARYING', 'BINARY LARGE OBJECT', 'DECIMAL', 'NUMERIC', 'SMALLINT', 'INTEGER', 'REAL', 'DOUBLE PRECISION', 'FLOAT', 'DATE', 'TIME', 'TIMESTAMP', 'TIME WITH TIME ZONE', 'TIMESTAMP WITH TIME ZONE', 'INTERVAL', or 'REF', then the value of COLUMN_SIZE in *SPECIAL_COLUMNS_QUERY* is implementation-dependent.

(Blank page)

Annex E
(informative)

Incompatibilities with ISO/IEC 9075:2003

This Annex modifies Annex E, “Incompatibilities with ISO/IEC 9075-2:2003”, in ISO/IEC 9075-2.

This edition of this part of ISO/IEC 9075 introduces some incompatibilities with the earlier version of Database Language SQL's Call-Level Interface as specified in ISO/IEC 9075-3:2003.

Except as specified in this Annex, features and capabilities of Database Language SQL's Call-Level Interface are compatible with ISO/IEC 9075-3:1999.

(Blank page)

Annex F
(informative)

SQL feature taxonomy

Table 48, “Feature taxonomy and definition for mandatory features”, contains a taxonomy of the mandatory features of the SQL language that are specified in this part of ISO/IEC 9075. In this table, the first column contains a counter that may be used to quickly locate rows of the table; these values otherwise have no use and are not stable — that is, they are subject to change in future editions of or even Technical Corrigenda to ISO/IEC 9075 without notice.

The column “Feature ID” column of this table specifies the formal identification of each feature and each sub-feature contained in the table.

The “Feature Name” column of this table contains a brief description of the feature or subfeature associated with the Feature ID value.

The “Feature Description” column of this table provides the only definition of the mandatory features of this part of ISO/IEC 9075. This definition consists of indications of specific language elements supported in each feature, subject to the constraints of all Syntax Rules, Access Rules, and Conformance Rules.

Table 48 — Feature taxonomy and definition for mandatory features

	Feature ID	**Feature Name**
1	**C011**	**All facilities defined by this part of ISO/IEC 9075**

(Blank page)

Index

Index entries appearing in **boldface** indicate the page where the word, phrase, or BNF nonterminal was defined; index entries appearing in *italics* indicate a page where the BNF nonterminal was used in a Format; and index entries appearing in roman type indicate a page where the word, phrase, or BNF nonterminal was used in a heading, Function, Syntax Rule, Access Rule, General Rule, Leveling Rule, Table, or other descriptive text.

— A —

— B —

— C —

— D —

— P —

— R —

— S —

— Y —

— Z —