Technical Standard

Data Management: Structured Query Language (SQL) Version 2

THE Open GROUP

[This page intentionally left blank]

X/Open CAE Specification

Data Management:

Structured Query Language (SQL), Version 2

X/Open Company Ltd.

X/Open CAE Specification

Data Management: Structured Query Language (SQL), Version 2

ISBN: 1-85912-151-9
X/Open Document Number: C449

Published by X/Open Company Ltd., U.K.

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

X/Open Company Limited
Apex Plaza
Forbury Road
Reading
Berkshire, RG1 1AX
United Kingdom

or by Electronic Mail to:

XoSpecs@xopen.org

Contents

Preface

X/Open

X/Open is an independent, worldwide, open systems organisation supported by most of the world's largest information systems suppliers, user organisations and software companies. Its mission is to bring to users greater value from computing, through the practical implementation of open systems.

X/Open's strategy for achieving this goal is to combine existing and emerging standards into a comprehensive, integrated, high-value and usable open system environment, called the Common Applications Environment (CAE). This environment covers the standards, above the hardware level, that are needed to support open systems. It provides for portability and interoperability of applications, and so protects investment in existing software while enabling additions and enhancements. It also allows users to move between systems with a minimum of retraining.

X/Open defines this CAE in a set of specifications which include an evolving portfolio of application programming interfaces (APIs) which significantly enhance portability of application programs at the source code level, along with definitions of and references to protocols and protocol profiles which significantly enhance the interoperability of applications and systems.

The X/Open CAE is implemented in real products and recognised by a distinctive trade mark — the X/Open brand — that is licensed by X/Open and may be used on products which have demonstrated their conformance.

X/Open Technical Publications

X/Open publishes a wide range of technical literature, the main part of which is focussed on specification development, but which also includes Guides, Snapshots, Technical Studies, Branding/Testing documents, industry surveys, and business titles.

There are two types of X/Open specification:

- *CAE Specifications*

  CAE (Common Applications Environment) specifications are the stable specifications that form the basis for X/Open-branded products. These specifications are intended to be used widely within the industry for product development and procurement purposes.

  Anyone developing products that implement an X/Open CAE specification can enjoy the benefits of a single, widely supported standard. In addition, they can demonstrate compliance with the majority of X/Open CAE specifications once these specifications are referenced in an X/Open component or profile definition and included in the X/Open branding programme.

  CAE specifications are published as soon as they are developed, not published to coincide with the launch of a particular X/Open brand. By making its specifications available in this way, X/Open makes it possible for conformant products to be developed as soon as is practicable, so enhancing the value of the X/Open brand as a procurement aid to users.

- *Preliminary Specifications*

  These specifications, which often address an emerging area of technology and consequently are not yet supported by multiple sources of stable conformant implementations, are released in a controlled manner for the purpose of validation through implementation of products. A Preliminary specification is not a draft specification. In fact, it is as stable as X/Open can make it, and on publication has gone through the same rigorous X/Open development and review procedures as a CAE specification.

  Preliminary specifications are analogous to the *trial-use* standards issued by formal standards organisations, and product development teams are encouraged to develop products on the basis of them. However, because of the nature of the technology that a Preliminary specification is addressing, it may be untried in multiple independent implementations, and may therefore change before being published as a CAE specification. There is always the intent to progress to a corresponding CAE specification, but the ability to do so depends on consensus among X/Open members. In all cases, any resulting CAE specification is made as upwards-compatible as possible. However, complete upwards-compatibility from the Preliminary to the CAE specification cannot be guaranteed.

In addition, X/Open publishes:

- *Guides*

  These provide information that X/Open believes is useful in the evaluation, procurement, development or management of open systems, particularly those that are X/Open-compliant. X/Open Guides are advisory, not normative, and should not be referenced for purposes of specifying or claiming X/Open conformance.

- *Technical Studies*

  X/Open Technical Studies present results of analyses performed by X/Open on subjects of interest in areas relevant to X/Open's Technical Programme. They are intended to communicate the findings to the outside world and, where appropriate, stimulate discussion and actions by other bodies and the industry in general.

- *Snapshots*

  These provide a mechanism for X/Open to disseminate information on its current direction and thinking, in advance of possible development of a Specification, Guide or Technical Study. The intention is to stimulate industry debate and prototyping, and solicit feedback. A Snapshot represents the interim results of an X/Open technical activity. Although at the time of its publication, there may be an intention to progress the activity towards publication of a Specification, Guide or Technical Study, X/Open is a consensus organisation, and makes no commitment regarding future development and further publication. Similarly, a Snapshot does not represent any commitment by X/Open members to develop any specific products.

Versions and Issues of Specifications

As with all *live* documents, CAE Specifications require revision, in this case as the subject technology develops and to align with emerging associated international standards. X/Open makes a distinction between revised specifications which are fully backward compatible and those which are not:

- a new *Version* indicates that this publication includes all the same (unchanged) definitive information from the previous publication of that title, but also includes extensions or additional information. As such, it *replaces* the previous publication.

- a new *Issue* does include changes to the definitive information contained in the previous publication of that title (and may also include extensions or additional information). As such, X/Open maintains *both* the previous and new issue as current publications.

Corrigenda

Most X/Open publications deal with technology at the leading edge of open systems development. Feedback from implementation experience gained from using these publications occasionally uncovers errors or inconsistencies. Significant errors or recommended solutions to reported problems are communicated by means of Corrigenda.

The reader of this document is advised to check periodically if any Corrigenda apply to this publication. This may be done in any one of the following ways:

- anonymous ftp to ftp.xopen.org
- ftpmail (see below)
- reference to the Corrigenda list in the latest X/Open Publications Price List.

To request Corrigenda information using ftpmail, send a message to ftpmail@xopen.org with the following four lines in the body of the message:

```
open
cd pub/Corrigenda
get index
quit
```

This will return the index of publications for which Corrigenda exist. Use the same email address to request a copy of the full corrigendum information following the email instructions.

SQL Access Group Moved into X/Open

Until the fourth quarter of 1994, the SQL Access Group was a separate body operating in the U.S.A., working with X/Open under a joint development and publishing agreement. It developed SQL-related specifications in collaboration with the X/Open Data Management Group.

During the fourth quarter of 1994, the SQL Access Group transferred all its assets and current activities to X/Open. These activities are now continuing in a technical working group within X/Open, called the X/Open SQL Access Group.

Much of the development work on this Structured Query Language (SQL), Version 2 CAE Specification was achieved before the transfer of the SQL Access Group to X/Open. Under the joint agreement which applied before this transfer, this joint development work resulted in joint publication of the Embedded SQL, Version 2 Snapshot (S424) in September, 1994. The Acknowledgements section of this publication expresses recognition of this progression, which has now resulted in publication of this CAE Specification.

Network Management Forum (NMF) SPIRIT Project

X/Open and NMF collaborate in a number of ways; the Service Provider's Integrated Requirements for Information Technology (SPIRIT) project is one example. SPIRIT provides a procurement specification for world-wide telecommunications service providers on a range of topics including SQL. The SPIRIT SQL procurement specification is included as Appendix D in this X/Open Specification. This illustrates the desire of X/Open and NMF to work together to achieve technical convergence of their respective specifications. The SPIRIT project, as a whole, is described in the SPIRIT Platform Blueprint (J401), whilst the SPIRIT SQL specification is covered in SPIRIT Language Profiles (J402). However, readers wishing to refer to the full text of the SPIRIT SQL specification are referred back to Appendix D as well as to the tables of limits in Section 7.1 on page 175.

This Document

This document is an X/Open CAE Specification (see above). It specifies the application program interface (API) for X/Open-compliant relational database management systems. It is closely based on the International Standard for the Database Language SQL, ISO 9075: 1992, but includes some extensions to the International Standard based on current implementations. This document encompasses the features required by NIST FIPS 127-2, also known as *Transitional SQL*. This document also contains two SPIRIT profiles of SQL.

Chapter 1 is an introduction to this specification.

Chapter 2 defines concepts and terms for relational database systems, and is an essential basis for the rest of the specification.

Chapter 3 introduces the syntax and semantics of the common elements of SQL and defines the notation used in the formal SQL syntax sections.

Chapter 4 describes how SQL constructs are embedded within a host-language source program, with particular reference to the COBOL and C languages.

Chapter 5 fully defines the syntax and semantics of executable SQL statements.

Chapter 6 describes the schema information available to the database user through system views.

Chapter 7 addresses implementation-specific issues and enumerates limits typically imposed by implementations.

Appendix A is a complete, symbolic definition of X/Open SQL syntax.

Appendix B lists all SQLSTATE return values that apply to X/Open SQL.

Appendix C compares X/Open SQL with the International Standard.

Appendix D and Appendix E define SPIRIT SQL, technically equivalent to that presented by Draft 2 of the Service Providers Integrated Requirements for Information Technology (SPIRIT) procurement specification. Appendix D is technically equivalent to SPIRIT SQL Issue 2; Appendix E contains additional information which, in conjunction with the information in Appendix D, is technically equivalent to SPIRIT SQL Issue 3. These appendices form a complete definition except that tabular information is included in the tables in Chapter 7.

Appendix F identifies possible future developments of the X/Open SQL definition.

Trade Marks

Acknowledgements

X/Open gratefully acknowledges past collaboration with and contributions from members of the SQL Access Group, which transferred its activities and assets to X/Open in the fourth quarter of 1994. Its activities are now continuing in a technical working group within X/Open, called the X/Open SQL Access Group.

Referenced Documents

The following documents are referenced in this specification:

ANS X3.168-1989
: American National Standard Database Language Embedded SQL.

CLI
: X/Open CAE Specification, February 1995, Data Management: SQL Call Level Interface (CLI) (ISBN: 1-85912-081-2, C451).

FIPS 1-2
: Federal Information Procurement Standard (FIPS) 1-2, Code for Information Interchange, Its Representations, Subsets and Extensions, NIST, 14 November 1994.

FIPS 127-2
: Federal Information Procurement Standard (FIPS) 127-2, Database Language SQL, NIST, 25 January 1993.

Internationalisation Guide
: X/Open Guide, July 1993, Internationalisation Guide, Version 2 (ISBN: 1-859120-02-4, G304).

ISO 1989:1985
: ISO 1989:1985, Programming Languages — COBOL (endorsement of ANSI Standard X3.23-1985).

ISO 8859-1:1987
: ISO 8859-1:1987, Information Processing — 8-bit Single-byte Coded Graphic Character Sets — Part 1: Latin Alphabet No. 1.

ISO/IEC 1539
: ISO/IEC 1539:1991, Information Technology — Programming Languages — Fortran (technically identical to ANSI standard X3.9-1978 [FORTRAN 77]).

ISO/IEC 1989:1974
: ISO/IEC 1989:1974, Programming Languages — COBOL (endorsement of ANSI Standard X3.23-1974).

ISO/IEC 4873
: ISO/IEC 4873:1991, Information Technology — ISO 8-bit Code for Information Interchange — Structure and Rules for Implementation.

ISO/IEC 646
: ISO/IEC 646:1991, Information Processing — ISO 7-bit Coded Character Set for Information Interchange.

ISO SQL or International Standard
: ISO/IEC 9075:1992, Information Technology — Database Language SQL (endorsement of ANSI standard X3.135-1992).

MIA
: Multivendor Integration Architecture, Book 5: Division 2, Application Program Interface Specifications, Part 4: Database Language, 31 March 1992.

RDA
: X/Open CAE Specification, August 1993, Data Management: SQL Remote Database Access (ISBN: 1-872630-98-7, C307).

SPIRIT Issue 2.0, Volume 1
: X/Open NMF SPIRIT Documentation, November 1994, SPIRIT Platform Blueprint, Issue 2.0, Volume 1 (ISBN: 1-85912-059-8, J401).

SPIRIT Issue 2.0, Volume 2
: X/Open NMF SPIRIT Documentation, March 1995, SPIRIT Language Profiles, Issue 2.0, Volume 2 (ISBN: 1-85912-062-8, J402).

SPIRIT Issue 3.0
: X/Open NMF SPIRIT Documentation, December 1995, SPIRIT Platform Blueprint (SPIRIT Issue 3.0) (ISBN: 1-85912-110-1, J405).

SQL Technical Corrigendum
: SQL Technical Corrigendum, December 1994, to ISO/IEC 9075:1992, Information Technology — Database Language SQL.

TX
: X/Open Preliminary Specification, October 1992, Distributed Transaction Processing: The TX (Transaction Demarcation) Specification (ISBN: 1-872630-65-0, P209).

XA
: X/Open CAE Specification, December 1991, Distributed Transaction Processing: The XA Specification (ISBN: 1-872630-24-3, C193 or XO/CAE/91/300).

Chapter 1

Introduction

1.1 Relational Database Standards

This document specifies the application program interface (API) for X/Open-compliant relational database management systems.

The widely-accepted standard for such an interface is the International Standard for the Database Language SQL, ISO 9075: 1992, which is identical to the American National Standard, X3.135-1992. References in this X/Open specification to the International Standard also implicitly reference the identical ANSI Standard. (See Appendix C on page 209 for a detailed comparison to the International Standard.)

This X/Open definition is based closely on the International Standard. However, it also reflects the capabilities of current implementations. X/Open has worked closely with vendors of the leading database management systems. This definition includes some features that were widely-available at the time of publication, and also some enhancements of these products known to be imminent.

This document encompasses the features required by the Transitional Level of NIST FIPS 127-2, also known as *Transitional SQL* (see Section 1.4.1 on page 7).[1]

1. The FIPS (Federal Information Processing Standards publication) is a procurement specification of the U.S. Government.

1.2 The X/Open Specification

1.2.1 Audience

This specification is written for writers of application programs and for implementors of database management systems.

For application writers, this specification emphasises portability — the ability to move the application to any X/Open-compliant database system without the need to modify source code. The specification embraces a common subset of facilities that various SQL products support. An important chapter for application writers is Chapter 7. It summarises maximum values for certain limits that application writers can safely assume all X/Open-compliant implementations support. It also discusses features that vary among implementations, giving coding recommendations in each case. Finally, it provides a syntactic technique application writers can use to delimit SQL code that is non-portable, without inhibiting the compilation of the program on any X/Open-compliant implementation.

For implementors, this specification guides the implementation of a database system for which portable applications can be written. In addition, this specification addresses emerging SQL features that are not yet widespread in the industry. It encourages future portability by giving a preferred specification for these features. Implementors should study the definition in this document of optional features (see below) and read Appendix F to discern areas of likely future X/Open specification activity.

1.2.2 Compliance Terminology

The dual nature of the specification (discussed above) requires that the following terms be defined clearly:

Optional features

An optional feature serves as guidance to implementors on the preferred syntax for an SQL feature that is not yet widespread. Implementors are currently free not to implement the feature, but if they implement it, they should do so as specified in this document. X/Open does not currently enforce the implementation of optional features, but intends to make them mandatory in future issues of this document, in the manner in which they are specified herein. At that time, implementations will be required to provide the feature.

X/Open may test implementations to see if they implement optional features as specified in this document.

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

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

OP The optional features in X/Open SQL are:

- catalogs, a component of qualified object names
- the DESCRIBE INPUT statement.

Deprecated features

Deprecated features include SQL syntax that X/Open views as obsolete or non-optimal. Deprecation indicates that X/Open intends to remove such features from future issues of this document.

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

Discussions of deprecated features are shaded with the DE margin notation, as shown below.

Implications on Applications

For syntax that is valid in this issue and that will become invalid in future issues, application writers should not use this deprecated syntax, but should instead use the alternatives specified in this document.

For syntax that is optional in this issue and that will become mandatory in future issues, it is the optionality that is deprecated. Application writers should convert to the new syntax.

Implications on Implementations

X/Open-compliant database systems are required to provide features even if they are deprecated, in the interest of backward-compatibility.

Inventory

The following feature of X/Open SQL is deprecated:

DE
- the schema OLD_INFO_SCHEM described in Chapter 6.

Compliance

A database implementation is X/Open-compliant if it supports all the assertions this document makes that are not labelled optional. In particular:

- It must support the limits[2] defined in Section 7.1 on page 175 and the escape clause defined in Section 7.2 on page 181.
- It must support the embedding of SQL in either C or COBOL or both as defined in Chapter 4.

The implementation may also support the features labelled optional.

An application program is X/Open-compliant if it uses only the SQL syntax contained in this document and not labelled optional. For portability, applications should follow the recommendations in Chapter 7. X/Open currently does not test compliance of application programs.

For specific details of X/Open policy regarding compliance with this specification, see Section 1.3 on page 5. For detailed information on compliance with respect to the International Standard, see Appendix C.

Implementation-defined

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

2. Where Section 7.1 on page 175 contains tables with X/Open and SPIRIT columns, X/Open-compliant implementations are only required to support the limits in the X/Open column.

Undefined

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

1.3 Compliance Policy

1.3.1 Language Embedding

An embedded SQL host program is a compilation unit that consists of statements in a host programming language and embedded SQL constructs. Chapter 4 defines the way in which SQL statements can be embedded in COBOL and C. Many SQL products support additional languages. The programming language statements must conform to the X/Open definitions for the host language with restrictions as indicated in Chapter 4 In particular, this X/Open specification does not yet support new features in the ISO 1989: 1985 COBOL Standard that impact the definition of host variables.

The goal of this X/Open specification is the ability to write portable SQL programs. Conformance to this specification means that the SQL implementation must include bindings to an X/Open-compliant COBOL or C implementation. X/Open intends to publish, for each SQL product it brands, the extent to which purchasers can use the product to write portable applications.

1.3.2 Flagging Non-portable Usage

During compilation of SQL programs, X/Open-compliant implementations report all non-portable SQL usage in the application program. A null report should assert that the program being compiled is fully portable to all X/Open SQL implementations. Therefore, the implementation checks all the following aspects of SQL usage:

- use of SQL language not defined in this specification, even if the implementation allows that usage as an extension to X/Open SQL; this includes use of SQL features selected by compile-time option
- use of a reserved word (from Section 3.1.6 on page 37, including the words not currently used by X/Open) as identifiers.

The portability report is a check of the program's use of SQL syntax, not whether the code is meaningful or likely to produce the desired effect.

Implementations may also check data type compatibility of SQL operations, but doing so requires knowledge of the metadata at compile time.

Implementations are not required to report non-conformant usage of the host programming language, nor erroneous use of the host language that is detected by the compiler, nor non-portable SQL usage in a host language statement that is in error. It is not possible for implementations to report at compile time all cases that will constitute access violations at run time.

The dynamic SQL statements described in Section 5.5 on page 124 accommodate SQL statement text that may not be known at compile time. The implementation detects and reports non-portable usage in dynamic SQL, as listed above, at run time.

To assist the writing of portable programs, X/Open recommends that implementations also report all uses of the following features of X/Open SQL:

- features that X/Open defines as optional
- features that X/Open defines as deprecated
- features for which Chapter 7 discusses implementation variability, including any use of the vendor escape clause defined in Section 7.2 on page 181 that specifies an ISO SQL dialect other than YEAR(1992).

1.3.3 Distributed Transaction Delimitation

Section 2.7 on page 28 discusses transactions, which are sequences of database operations with certain collective characteristics such as atomicity. Transaction **delimitation** must exist in order to define the grouping of database operations into transactions.

X/Open compliance policy on transaction delimitation is as follows:

- All implementations support the rule that a transaction begins when an application executes an SQL statement that operates on a database, as defined in Section 5.6 on page 139. The transaction ends when the application executes the SQL statement COMMIT or ROLLBACK.
- It is implementation-defined if a transaction begins when an application executes the *tx_begin*( ) function described in the X/Open **TX** Specification; and ends when it executes *tx_end*( ).
- Other implementation-defined transaction delimitation interfaces may be supported.

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

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

1.3.4 Character Set Support

Section 2.4.3 on page 21 defines character sets. Other parts of this specification discuss various cases where an embedded SQL program can specify character sets, but do not indicate which character sets the implementation supports.

X/Open-compliant implementations shall support at least the following character sets:

ASCII_FULL
ASCII_GRAPHIC
LATIN1
SQL_CHARACTER
SQL_TEXT

These character sets are defined in Section 16.7 of FIPS 127-2. SQL_CHARACTER is the SQL character set, which is also discussed in **SQL Character Set** on page 32.

(SPIRIT Issue 3 also requires this degree of support, beginning in the summer of 1997; see item (44)(a) in Section E.4 on page 239 of this specification.)

1.4 This Issue

This section describes the differences between this issue and the August 1992 issue of this document.

1.4.1 Transitional SQL

The primary reason for the new issue is to align this specification with the definition of the Transitional SQL level of FIPS 127-2. X/Open anticipates that Transitional SQL will be a frequently-cited level of SQL implementation because it incorporates some advanced SQL features but is likely to be widely implemented by products in the marketplace.

This issue includes the features listed below, which are required for classification as Transitional SQL and which were not included in the August 1992 issue. Other features of Transitional SQL were already specified in the August 1992 issue.

New Table Combination Operations

Joined tables are now specified in Section 3.11.2 on page 73. X/Open specifies NATURAL JOIN, JOIN ON, and JOIN USING, qualified by INNER, LEFT OUTER, and RIGHT OUTER.

The *query-expression* syntactic element is moved to the Common Elements chapter because it can now be used in a CREATE VIEW statement as well as in DECLARE CURSOR. This means the keyword UNION can appear within CREATE VIEW.

Referential ON DELETE Actions

This issue defines the ON DELETE clause of the CREATE TABLE statement, specifying how database integrity should be maintained if deletion of rows from a table results in the violation of a FOREIGN KEY reference in the table being defined. X/Open defines the NO ACTION, CASCADE, SET NULL, and SET DEFAULT actions.[3]

CAST Function

The CAST function explicitly converts an expression or null value from one data type to another data type. The CAST function can raise a variety of syntax, range, and formatting exceptions.

String Operations

This issue defines operations on fixed-length and variable-length character strings, including the concatenation operator and the CHAR_LENGTH, SUBSTRING and TRIM scalar functions.

Date/Time

This issue includes the date/time and interval generic data type with the DATE, INTERVAL, TIME and TIMESTAMP named data types, as specified in the International Standard.[4]

3. Neither this document nor the Transitional SQL level of FIPS 127-2 specifies the corresponding ON UPDATE clause.
4. Neither this document nor the Transitional SQL level of FIPS 127-2 specifies timezones in date/time and interval data types.

Expanded ALTER TABLE Statement

A new form of the ALTER TABLE statement is specified that lets applications drop as well as add columns. The ADD COLUMN form adopts all the syntax for defining the new column that can be used in the CREATE TABLE statement.

Transaction Control

The previous issue specified SERIALIZABLE as the isolation level for all transactions and discussed database phenomena that can occur at lower isolation levels. The current issue allows three lower isolation levels. Specifying a lower isolation level waives guarantees against such phenomena and may increase database performance. Implementations are not required to provide four discrete isolation levels as long as they provide all the guarantees the application specified.

The SET TRANSACTION statement is now specified. This statement sets the access mode, isolation level, and size of the diagnostics area for the next transaction.

Session Statements

The SET CATALOG and SET SCHEMA statements are added. They let applications specify the default catalog and schema name in a session that implicitly qualify objects in cases where the application does not use explicit qualification.

PRIMARY KEY Enhancement

It is now possible to specify a unique constraint (PRIMARY KEY or UNIQUE) in CREATE TABLE without also specifying NOT NULL. Removal of the previous restriction permits UNIQUE constraints on columns where null values are permitted; PRIMARY KEY always prohibits null values.

New System Views

This issue specifies the COLUMN_PRIVILEGES, INDEXES, SCHEMATA, TABLE_PRIVILEGES, USAGE_PRIVILEGES and VIEWS system views. The INDEXES system view is an extension to the International Standard because the concept of indexes is an X/Open extension.

New Syntax Options

This issue permits the keyword AS in correlation names, TABLE in the GRANT and REVOKE statements, and FROM in the FETCH statement. This issue also makes the keyword WORK optional in the COMMIT and ROLLBACK statements; it was formerly mandatory.

Enhancements to GET DIAGNOSTICS

The GET DIAGNOSTICS statement can now be used to determine the type of SQL statement that generated the diagnostics.

New fields associated with a specific diagnostic indicate any table, column or cursor to which the diagnostic applies. The message length is now indicated in characters as well as octets. Additional fields indicate any constraint to which the diagnostic applies, but their use is not mandatory for implementations because this specification does not define constraints as persistent objects.

Enhancements to INSERT

New forms of the INSERT statement let the application program specify the values to be inserted using a *row-value-constructor* or specify a row consisting totally of the columns' default values.

1.4.2 Internationalisation

This issue includes the following internationalisation features from the International Standard:

- A new compliance policy, specifying which character sets an implementation must support, now appears as Section 1.3.4 on page 6.
- A general discussion of character sets and collations in SQL appears in Section 2.4.3 on page 21. Section 3.1.2 on page 32 has been rewritten to apply to non-European character sets and to define syntactic elements used in the specification of character sets. Section 3.1.5 on page 36 specifies syntax by which syntactic elements of SQL can be modified to specify their character set. Section 3.2.1 on page 40 contains new information on how a character set is specified for a variable of a character-string data type.
- This issue adds support for the NATIONAL CHARACTER data type (and its synonyms NATIONAL CHAR and NCHAR, and their VARYING counterparts).
- The CONVERT and TRANSLATE functions are added to Section 3.9.3 on page 59.
- New syntax is allowed in the SQL declare section (see Section 4.2 on page 84) to specify the character set of any host variable declared therein; and, by using the SQL NAMES ARE syntax, to specify a default character set for the compilation unit.
- Chapter 5 defines new executable SQL statements:
  - — CREATE and DROP statements are defined for CHARACTER SET, COLLATION and TRANSLATION.
  - — The existing GRANT and REVOKE statements are extended to apply to character sets, collations and translations.
- The CHARACTER_SETS, COLLATIONS and TRANSLATIONS system views are added to Chapter 6.

1.4.3 Alignment with SPIRIT SQL

SPIRIT SQL is defined, separately from the definition of X/Open SQL, in Appendix D and Appendix E. X/Open has modified its definition of embedded SQL to more closely align its feature set with that chosen by SPIRIT. This resulted in the following changes:

LOWER, POSITION and UPPER String Functions

The LOWER, POSITION and UPPER string functions are added. LOWER and UPPER convert relevant characters within a character string to the corresponding upper-case or lower-case letter. POSITION reports the location of a specified string within another string.

1.4.4 Other New Material in this Issue

The August 1992 issue eliminated the majority of the exclusions that previously existed with respect to the Entry Level of the International Standard. The current issue eliminates the two remaining exclusions and converges completely with Entry Level SQL as specified in the International Standard. (See Appendix C for a detailed comparison to the International Standard.)

The two changes are as follows:

- The previous issue did not address how a schema is created, as it does not affect production application programs. The current issue specifies CREATE SCHEMA and DROP SCHEMA statements that conform to the definition in the International Standard.
- The Integrity Enhancement Feature (IEF), an optional SQL feature in the August 1992 issue, is now mandatory. (The IEF lets the database designer indicate which columns of a table form a primary key, which columns of a table form a foreign key referencing the same or another table, and what values are permitted in an individual column. It also allows specification of a default value for a column.)

In addition, two sections of Chapter 7, which gave the impression that the CONNECT and EXECUTE IMMEDIATE statements were optional for X/Open-compliance, have been deleted.

Three-part Naming

OP This issue includes catalogs as an optional feature. It specifies a unique naming scheme consisting of catalog, schema, and object name, while providing subsetted and implementation-defined alternatives. A schema name no longer has to be the same as the user name. This provides the possibility of more than one schema per user.

DESCRIBE INPUT

OP New syntax is specified for the DESCRIBE statement. DESCRIBE OUTPUT, discussed in the previous issue, retrieves information on columns of a result set that would result from execution of a specified statement. A new DESCRIBE INPUT form retrieves information on dynamic parameters of a specified statement.

ISO Technical Corrigendum

Minor clarifications have been added to this specification based on a study of the referenced Technical Corrigendum (ISO SQL TC1) to the International Standard.

1.4.5 Substantive Changes in this Issue

This issue is designed to be harmonious with previous specifications. However, the following changes from previous issues have been made:

Additional Keywords

This issue reserves the following keywords, which the previous issue permitted applications to use as user-defined names:

- The keywords ACTION, BOTH, CROSS, CURRENT_USER, INNER, JOIN, LEADING, LEFT, NATURAL, NO, OUTER, PAD, RELATIVE, RIGHT, SCHEMA, SESSION, SESSION_USER, SPACE, SYSTEM_USER, TRAILING and TRIM appear in the International Standard but were omitted from the previous X/Open issue.

- The previous issue advised applications not to use the reserved words AUTHORIZATION, COLLATE, COLLATION, CURRENT_USER, INPUT, NATIONAL, NCHAR, SESSION_USER, SYSTEM_USER and TRANSLATE. These reserved words are now used in X/Open SQL.

The keywords DICTIONARY, DISPLACEMENT, IGNORE, OFF, and SYSTEM, reserved in the last issue, are no longer reserved and are available for use as user-defined names.

A complete list of reserved SQL keywords appears in Section 3.1.6 on page 37.

Removal of Deprecated Features

The August 1992 issue listed five deprecated features. These features were present in existing implementations and used by existing application programs despite their deviation from standards. Deprecation obliged X/Open-compliant implementations to maintain support for the features but warned application writers to cease to use them. All five deprecated features are deviations from the International Standard and are omitted from the current X/Open issue. This means that X/Open-compliant implementations are no longer required to support the features and that applications that continue to use the features will not necessarily still be portable. The changes are as follows:

- The SQL Communication Area (SQLCA) is no longer specified in this issue.
- A form of the ALTER TABLE statement that lets a single statement specify changes to more than one column is no longer specified in this issue. Application writers should code separate ALTER TABLE statements for each column.
- This issue no longer lets CREATE TABLE statements have multiple UNIQUE or PRIMARY KEY clauses that apply to the same set of columns.
- This issue requires every DROP TABLE, DROP VIEW, and REVOKE statement to use either the CASCADE or the RESTRICT keyword to specify the effect if the statement would require the dropping of other tables or constraints. In the previous issue, CASCADE was assumed, but the specification noted some implementation variability.
- The keyword PRIVILEGES (in GRANT ALL PRIVILEGES and REVOKE ALL PRIVILEGES) is mandatory in this issue. The previous issue let applications omit the keyword.

Changes to Diagnostics

The subtle differences between SQLSTATEs '**37**000' and '**42**000' have been deleted. Consistent with a recent correction to the International Standard, all syntax error reports now use SQLSTATE '**42**000'.

There is a change to the criteria by which the implementation determines which record to place first in the diagnostics area; see Section 4.5.2 on page 90.

Other Changes

This issue includes error corrections previously reported in the X/Open SQL CAE Corrigendum.

1.4.6 Documentation and Policy Changes in this Issue

Flagger

This issue includes the compliance requirement that implementations report at compile time on any non-conformant or non-portable SQL usage (see Section 1.3.2 on page 5).

XA Specification

This issue includes, in Section 1.3.3 on page 6, a description of how to use SQL implementations that comply with this specification and with the X/Open **XA** Specification to achieve atomic completion of SQL and non-SQL work.

Rewritten Text

Finally, a small amount of text has been rewritten in the continuing interest of clarity. Readers should not interpret a change in phrasing as a change to the substance of this specification.

1.5 SQL Registry

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

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

Obtaining Copies of the Registry

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

- On the World-Wide Web using the following Universal Resource Locators (URL):

  — Plain text in **http://www.xopen.org/infosrv/SQL_Registry/registry.txt**

  — Acrobat PDF in **http://www.xopen.org/infosrv/SQL_Registry/registry.pdf**

  — HTML in **http://www.xopen.org/infosrv/SQL_Registry/registry.htm**

  — PostScript in **http://www.xopen.org/infosrv/SQL_Registry/registry.ps**.

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

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

  The text below describes the anonymous ftp service.

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

  ```
  ftp ftp.xopen.org
  ```

  — At the login prompt, enter `anonymous` or `ftp` as your user name.

  — You will then be prompted for a password. Respond by typing your full e-mail address including Internet domain. You will be granted access to any of the files that have been made available for anonymous FTP, but not to other files on the system.

  — Select the SQL Registry by typing:

  ```
  cd pub/SQL_Registry
  ```

  — Retrieve registry information by typing one of the following:

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

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

Submitting Requests to the Registry

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

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

The text of the message must specify:

- the organization on whose behalf the request is being made
- the name of the individual making the request
- the exact title of each table in the Registry into which values should be allocated
- for each category of values, an approximation of the number of values needed.

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

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

Chapter 2

Concepts

2.1 Introduction

This chapter defines terms and concepts of relational database systems. The remainder of this X/Open specification relies on these definitions.

2.1.1 General Terms

A **program** means the host application program that uses SQL to access data.

Compilation means the process of converting a program to its executable form. It is implementation-defined whether a pre-compiler is used and how the task of processing SQL syntax is divided between the compiler and any pre-compiler. When this specification asserts that a fact must be known at compilation time, the fact must be known before any pre-compilation begins.

A **set** is an unordered collection of distinct objects.

A **multi-set** is an unordered collection of objects that are not necessarily distinct.

A **sequence** is an ordered collection of objects that are not necessarily distinct.

The **cardinality** of a collection is the number of objects in that collection. Unless specified otherwise, any collection may be empty.

2.2 Data Types and Values

A **value** is a data item in the database. A single value is the smallest logical subdivision in the database. The physical representation of a value in a database is undefined.

A **null value** is distinct from all non-null values and represents a case where an actual value is not known or not applicable.

Each non-null value has a **data type**. A data type is a set of possible values. Section 3.2 on page 39 defines four **generic data types**, character string, numeric, date / time and interval. Within each generic data type, there are several **named data types** with specific attributes.

The null value is a valid value for all data types. It is distinct from all non-null values of that data type.

Each non-null value has a **logical representation** which is its display form in the syntax of SQL. Its logical representation is a **literal** (see Section 3.4 on page 48). The null value has no literal representation. In some SQL statements, the keyword NULL refers to a null value.

Each non-null value is in the character-string, date / time, interval or numeric class. Any two values can be compared unless they are in different classes.

The SQL language provides **expressions** that let database queries be based on computations (see Section 3.9 on page 57) and **predicates** that extract rows from tables based on whether they satisfy true / false assertions (see Section 3.10 on page 67). The host language may provide mechanisms to perform additional computations, whose results can be used to generate a query.

2.3 Tables

A **table** is a collection of values that may vary over time. The table's values are arranged into **rows** and **columns**. The columns are ordered and at least one must be present. The rows are not ordered and a table can have zero rows. Thus, a table is a multi-set of rows; a row is a non-empty sequence of values.

A value is the smallest unit of data in a table that can be selected or updated. A row is the smallest unit of data that can be inserted into a table and deleted from a table.

A table has a specification, which includes a specification of each of its columns.

Every row of a table has the same cardinality and contains a value of every column of that table. The *i*th value in every row of a table is a value of the *i*th column of that table.

The **degree** of a table is its number of columns, which is the cardinality of each row.

The **cardinality** of the table is its number of rows, which is the cardinality of each column.

2.3.1 Attributes of Columns

A column is a multi-set of values. The creator of a table specifies each column's attributes and ordinal position within the table. A column's attributes include its name, its data type, and whether it is constrained to contain only non-null values. Columns can also be added to a table after it is created.

Each column has a **default value**. This is the value to be given to the column when a program inserts a new row into the table without specifying a value for that column. A default value can be specified when the column is added to a table. If no default value is specified, the column's default value is the null value.

The creator of a table can specify additional constraints on a table or on a column.

2.3.2 Types of Table

Each user-defined table is one of the following:

Base table	A base table is a named table defined by a CREATE TABLE statement. The specification of a base table includes its name.
Derived table	A derived table is a table derived directly or indirectly from one or more other tables by the evaluation of a *query-expression*. Section 3.11.3 on page 75 describes the options the SQL program has for conceptually combining tables and for extracting desired information from the table(s).

Examples of derived tables include:

Viewed table	A viewed table (or a **view**) is a named derived table defined by a CREATE VIEW statement. The specification of a viewed table includes its name. (Derived tables other than views are not named.)
Grouped table	When a query specification contains a GROUP BY or HAVING clause, evaluation involves arranging rows into groups so that, in each group, all values of specified columns (called the grouping columns) are equal. A grouped table is a derived table that contains such groups. In some contexts, set functions operate on the individual groups. For example, a computation can use the sum, average, or cardinality of a group.
Grouped view	A grouped view is a viewed table derived from a grouped table.

In this document, table generally refers to viewed tables as well as base tables.

Updatability of Tables

A table is either **updatable** or **read-only**. Updatability means whether it is valid to modify the table. The INSERT, UPDATE and DELETE statements are permitted for updatable tables and are not permitted for read-only tables.

All base tables are updatable. No grouped tables nor grouped views are updatable. A viewed table is updatable if it is derived from an updatable *query-expression*. The updatability of a *query-expression* is defined in Section 3.11.3 on page 75. It generally depends on the updatability of the constituent *query-specification*(s), which is defined in Section 3.11.1 on page 71.

The positioned DELETE and positioned UPDATE statements modify a table using a cursor. Any cursor used in these contexts must be updatable. Updatability of a cursor is discussed in Section 4.4 on page 87. It depends on the updatability of the underlying *query-expression* and on the syntax used in the DECLARE CURSOR statement.

2.3.3 Indexes

An **index** can be thought of as a list of pointers to the rows of a table, ordered based on the values of one or more specified columns of the table. Existence of an index may enhance performance by obviating certain sort operations or by reducing the scanning of the table that is necessary to build a result set.

2.3.4 System Views

In addition to user-defined viewed tables, there are predefined viewed tables called **system views**. They are user-accessible, read-only viewed tables that contain information about the database. System views are described in more detail in Chapter 6.

2.3.5 Integrity Constraints

Databases may have **integrity constraints**. They define a database's valid states by constraining the values in its base tables.

- **NOT NULL constraint**

  A table may prevent a column from containing null values.

- **Unique constraint**

  A table may require that each row's values for a column, or for a group of columns taken together, be unique. One set of unique columns may form a primary key for the table. (The indexes mentioned in Section 2.3.3 can also be used to prevent two rows in a table from containing the same values in one or more columns.)

- **Referential constraint**

  A table may require that each row's values for a column, or for a group of columns taken together (provided none are null), be present in a corresponding set of columns in some table in the database. These columns are called a **foreign key**.

- **Check constraint**

  A table may limit the values that may be stored in a row.

Integrity constraints are checked at the end of the SQL statement.

A CREATE TABLE statement that creates a table with integrity constraints specifies the result of subsequent operations that would violate a constraint. For example, the subsequent statement may fail, it may have no effect, or it may make additional changes to the database to preserve the required integrity.

2.4 Database System

2.4.1 Clients and Servers

The operation of a database system effectively involves two processors, a **client** and a **server**. The terms client and server in this document always mean SQL client and SQL server. Connection statements such as the CONNECT statement (see **Connections**) manage the associations between a client and one or more servers.

When a program is active, it is bound in an implementation-defined manner to a single client that processes the first implicit or explicit CONNECT statement. The client communicates with one or more servers, either directly or through other agents such as RDA. The client processes the GET DIAGNOSTICS statement and has the primary role in processing connect statements. The client may provide other functions independent of any server.

A server processes database requests from the client. Following the execution of such statements, diagnostic information is passed (in an undefined way) into the diagnostics area of the client.

Number and Location of Servers

An X/Open-compliant client lets the application connect to multiple servers. The application uses a single SQL programming paradigm to gain access to both local and remote servers.

Connections

A **connection** is an association between a client and a server. The following statements manage connections:

- The CONNECT statement associates a client with a server.
- The DISCONNECT statement ends this association.
- The SET CONNECTION statement chooses a connection from among existing connections.

Some implementations restrict the use of connection statements when a transaction is active; see Section 2.7 on page 28.

Connection Context

For the duration of a connection[5] between a client and server, the implementation preserves at least the following context:

- The following information that affects the execution of SQL statements:

5. The International Standard associates some of the context with a SQL **session**, which is a concept defined separately from the concept of a **connection** to a server. In this version of this specification, the terms are interchangeable. This document uses the term connection exclusively. Depending on future development of SQL specifications, this equivalence may cease to be useful, and some connection context may have a duration that is longer or shorter than the duration of a connection.

— The name of the default character set (see Section 2.4.3 on page 21) for user-defined names and literals in preparable SQL statements where a character set is not specified explicitly

— The default catalog name and the default schema name (see Section 3.1.3 on page 35)

- The transaction attributes described in Section 2.7.1 on page 28.

2.4.2 Database Organisation

Metadata and Data

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

- Metadata comprises the definitions of all active base tables, viewed tables, indexes, privileges and user names. (An item of metadata is active if it has been defined and has not subsequently been dropped. User names are defined in Section 2.6.1 on page 26.)
- Data comprises every value in every active table.

Schemata

A **schema** (the plural is **schemata**) is a collection of named objects.[6] There are the following classes of schema object:

- Character sets (see Section 2.4.3 on page 21)
- Collations (see Section 2.4.4 on page 23)
- Indexes (see Section 2.3.3 on page 18)
- Tables (see Section 2.3 on page 17)
- Translations (see **TRANSLATE** on page 62)

A schema may contain base tables, and contains any indexes that are defined on the base tables.

A schema has a separate namespace for each class of object; that is, the name of every such object in the schema must be unique. Every schema has an owner, who has exclusive control over the objects in the schema. (See also Section 2.6.2 on page 26.)

Three-part Naming

OP An application can uniquely identify a table or index by qualifying the table or index identifier (preceding it with its catalog and schema name). The use of catalog, schema, and object name is called three-part naming. For the syntax of qualification, see Section 3.1.3 on page 35.

OP Support for catalogs is optional. If the implementation supports catalogs, every schema resides in a catalog.

6. Correlations and cursors also have names but are not persistent objects in the schema. A correlation relates to a specific query (see Section 3.11.1 on page 71) and its name need only be unique within that query. A cursor relates to a specific table and its name need only be unique within that table.

2.4.3 Character Sets

Every character string has a character set. A character set is a named object.

The most significant aspect of the character set to the user of SQL is that it defines a **repertoire** of characters that can be used to form a string. Different character sets allow use of SQL to represent data in different human languages.

Along with the repertoire, a character set also defines a **form-of-use** (see the **Glossary**) and an **encoding** (which octet values represent each character). These aspects become relevant to the user of SQL when moving information between the database and host-language variables. An implementation may define conversions that let application programs convert a character string from one form-of-use to another using the CONVERT string operation (see application programs translate a character string from one character set to another using the TRANSLATE string operation (see **TRANSLATE** on page 62).

A character set also defines a default collation (see Section 2.4.4 on page 23).

Notable Character Sets

The **SQL character set** is a minimal character set consisting of English-language letters, digits, and those symbols required by the syntax of SQL. It is defined in Section 3.1.2 on page 32.

There is a collection of **named character sets** available to the SQL user. Certain named character sets are specified in Section 1.3.4 on page 6. In most cases where strings appear in an SQL program, the program can specify the character set of those strings. (This specification is in the SQL character set.)

There may be other implementation-defined named character sets. Examples of these are LATIN1 and UNICODE. The contents of each such character set, and the name used to reference it, are implementation-defined.

The **national character set** is a single specific character set. It is one of the named character sets that the implementation supports. In some cases, special SQL syntax specifies the national character set more simply than having to name the character set.

It is implementation-defined how a character set is designated as the national character set. The designation may be the result of administrative action at the SQL installation. The designation of a national character set is not a part of an SQL program.

The character set named **SQL_TEXT** is an implementation-defined character set whose repertoire contains all the characters the implementation can represent. SQL_TEXT is the union of all the named character sets the implementation supports. Specifying that a string uses SQL_TEXT indicates that its selection of characters is not limited to the repertoire of a specific, named character set.

Character-set Contexts

Character strings occur in three different contexts in SQL, which are subject to different rules:

- **SQL language components**

  The predefined components of the SQL language (the tokens defined in Section 3.1.2 on page 32 and the keywords listed in Section 3.1.6 on page 37) are in the SQL character set. Aspects of the embedded SQL source program that represent information in the following two items are not restricted to the SQL character set.

- **User-defined names and literals**

  The user-defined names discussed in **User-defined Names** on page 33 (for example, variable names) and character-string literals discussed in Section 3.4 on page 48 have a character set defined as follows:

  — Syntax defined in Section 3.1.5 on page 36 lets the character set of a single such element be specified.

  — An embedded SQL source program can define a default character set (using the NAMES ARE syntax in the SQL DECLARE SECTION; see Section 4.2 on page 84). This character set applies to elements that do not specify their own character set.

  When an SQL statement is prepared, elements in that statement that do not specify their own character set are in the default character set for the connection, which can be changed using the SET NAMES statement.

  — Elements for which a character set is not specified in either of the above methods are in an implementation-defined **default character set**.

  The default character set for user-defined names and literals may be different from the default character set for database data and host variables, presented below. The default character set for user-defined names and literals may be the national character set.

- **Database data and host variables**

  For table columns whose data type is a character-string data type, the metadata includes an indication of the character set.

  — When a table is created with character-string columns, or when character-string columns are added to an existing table, a character set can be specified for the column.

  — A schema can define a default character set. This character set applies to all character-string columns that do not specify their own character set.

  — Columns for which a character set is not specified in either of the above two methods are in an implementation-defined **default character set**.

  The default character set for database data may be different from the default character set for user-defined names and literals. The default character set for database data may be the national character set.

Metadata (for example, names and definitions of objects in the database) is stored in the SQL_TEXT character set, in order to accommodate all characters from all character sets the implementation supports. An indication of the character set that was used when the object was defined is also stored.

The description of each host variable includes an indication of the character set. The rules by which this indication is stored are implementation-defined.

2.4.4 Collations

A collation is a method of determining the sort order of two character strings in a particular character set. A collation is a named object. Every character set (see Section 2.4.3 on page 21) specifies a default collation. Several collations may be defined for the same character set.

The relevant collation indicates the effect of comparing two character strings. For each case in SQL where character strings are compared, a collation is specified or implied.

Collation is defined to apply to character strings, not to individual characters, because there are many aspects of natural languages that make it impossible to compare two strings by simply comparing pairs of corresponding characters in turn. Often the sort order of single characters depends on context, such as on the adjacent characters.

The collation that applies to a string expression is determined as follows:

- Any string expression can use SQL syntax to specify a collation (see Section 3.9.7 on page 66).
- If it does not, but the string is an object in the database, then its collation is implied (from the default collation of its character set) or specified when the object is defined.
- Literals and host variables are assumed to use the collation of an object against which they are compared.
- An attempt to compare two strings for which a collation has not been defined or inferred using one of the above techniques is a syntax error.

In every computation involving strings, the implementation must keep track of the governing collation at every step in the computation, in order to retain the collation that will govern any comparisons performed on the result of the computation.[7]

Pad Attribute

Every collation has a **pad attribute**, which influences the method of comparing character strings. The pad attribute of built-in collations is implementation-defined; the pad attribute of user-defined collations is specified in the CREATE COLLATION executable statement.

The pad attribute of any collation is one of the following:

Pad space	Before such a comparison, the shorter string is conceptually padded with spaces to the length in characters of the longer string.
No pad	No such conceptual padding occurs.

The effect of each value of the pad attribute is discussed in **Character** on page 53.

7. The above rules summarise for the benefit of the application writer the detailed **coercibility** rules for strings of collations contained in the International Standard (Subclause 4.2.3, **Rules determining collating sequence usage**). These rules indicate the collation of a string expression resulting from the use of operators and its effect on subsequent comparisons. X/Open-compliant implementations must obey these rules in order to produce results that are identical to those on implementations that comply with the International Standard.

2.5 Using SQL

2.5.1 Cursors

A program gains access to a derived table, one row at a time, using **cursors**. The DECLARE CURSOR statement defines the cursor by naming it, specifying a derived table, and optionally specifying an ordering of the derived rows. Section 4.4 on page 87 describes the attributes and usage of cursors.

2.5.2 Executable SQL Statements

An executable SQL statement is a statement from one of the following categories:

- A data definition statement modifies the metadata of the database. These statements create, modify and drop (delete) tables and indexes (see Section 5.3 on page 102). Certain data definition statements grant and revoke privileges on tables (see Section 2.6 on page 26).
- A data manipulation statement operates on the data of the database or controls the state of a cursor. Users with suitable privileges can perform select, fetch, insert, update and delete operations on the data in a table. See Section 5.4 on page 118.
- Certain SQL statements provide the capability of dynamic SQL, as discussed in Section 2.5.4 on page 25.
- A transaction control statement helps group other SQL statements into atomic sequences, guaranteeing to apply the effects of all statements in the transaction, or of none, to the database. These statements are summarised in Section 2.7 on page 28, and defined in Section 5.6 on page 139.
- Connection statements manage connections between SQL clients and SQL servers. These statements are summarised in **Connections** on page 19, and defined in Section 5.7 on page 142.
- The diagnostic statement retrieves information on the outcome of other SQL statements (see Section 5.10 on page 150).

2.5.3 Embedded Constructs

An embedded SQL program is a compilation unit that consists of constructs of a programming language (called the host language), and SQL constructs from the following list:

- an SQL declare section, which defines program variables used in embedded SQL statements
- an optional DECLARE AUTHORIZATION statement, which specifies an authorization identifier to be used to check the privileges for statements in the program
- a DECLARE CURSOR statement, which defines a cursor
- a WHENEVER statement, which tells the program what action to take when executing an embedded SQL statement produces outcomes other than unqualified success
- an executable SQL statement.

Within the SQL declare section, the application declares all host-language variables it uses in conjunction with embedded SQL statements:

- An **embedded host variable** is a normal variable that an embedded SQL statement uses to either obtain a value from, or convey a value to, the program.

- An **indicator variable** is an integer host-language variable associated with another embedded host variable in an embedded SQL statement. It indicates whether the input or output value of the associated embedded host variable is the null value, and whether it contains a truncated character string.

These topics are discussed further in Chapter 4.

2.5.4 Dynamic SQL

Dynamic SQL statements provide a way to execute SQL statements whose specifics may not be known at compile-time. The application may generate the SQL statement dynamically during program execution. There are three ways to use dynamic SQL:

- **One-time execution**

  The EXECUTE IMMEDIATE statement is used for a one-time execution of a statement other than a cursor specification. The program prepares an executable SQL statement at execution time and immediately executes it.

- **Prepare and execute a statement**

  The PREPARE statement is used to generate an executable object from an SQL statement, other than a cursor specification, and to assign it to an identifier. The EXECUTE statement is then used to associate input values with the parameters in the prepared statement and to execute it as though it had been coded when the program was written.

- **Prepare and execute a cursor specification**

  A variant of the cursor mechanism is used to create cursors dynamically. The program uses a dynamic DECLARE CURSOR declarative to associate a cursor with a statement identifier, then uses PREPARE to prepare a cursor specification and to assign the prepared statement to the same identifier. The DESCRIBE statement is then used to retrieve the attributes of the columns of the derived table.

  The application uses the dynamic OPEN statement to associate input values with parameters in the prepared statement and open the dynamic cursor. Then dynamic forms of the FETCH, DELETE and UPDATE statements can be used to retrieve and update rows in the table. A dynamic CLOSE statement is used to close the cursor.

A variety of data definition statements and data manipulation statements can be prepared and executed with dynamic SQL. Dynamic SQL is explained in Section 5.5 on page 124.

2.5.5 Return Status

Any SQL statement can result in unqualified success or can produce a variety of errors or warnings. Section 4.5 on page 90 discusses these outcomes and ways that application programs can sense error and warning outcomes. The diagnostic statement mentioned in Section 2.5 on page 24 is one method of error detection.

2.6 Access Control

2.6.1 Users

A server distinguishes between different **users**. Each user is identified by a **user name**. User names are associated with a database in an implementation-defined way that identifies users entitled to use the database.

Every program is executed on behalf of a user. The **current user** is the user on behalf of whom the current program is executed. The identity of the user, and optionally authentication information, are specified implicitly or explicitly in the CONNECT statement.

Any implementation may use additional criteria to restrict database access.

2.6.2 Ownership

Every object in a schema is owned by the owner of the schema in which it resides. Only the owner of a schema can add tables and indexes to that schema. A schema can be owned by its creator; or a user can create and populate a schema and specify another user who becomes the owner of the schema.

2.6.3 Authorisation Identifiers

Every database operation performed by an SQL statement is contingent on authorisation to perform that operation. Authorisation involves checking the privileges (see Section 2.6.4) of an **authorisation identifier**.

Invoker's Rights

By default, authorisation checking relates to the user who invokes the SQL statement. When a connection is established, the invoker is the user name specified in the CONNECT statement, or an implementation-defined user name if one is not specified. This can be overridden by an authorisation identifier provided in the SET SESSION AUTHORIZATION executable statement (if the current user is authorised to execute this statement).

Definer's Rights

An embedded SQL program can instead be written to specify that authorisation checking relates to the definer of the program. The program specifies this by using a DECLARE AUTHORIZATION statement containing an authorisation identifier.

For example, such a program can perform operations on a table to which the definer has access without regard for whether the invokers have access. The program might make these operations available unconditionally to its invokers or it might define its own access criteria.

Programs can make their behaviour contingent on the identity or authorisation of the current user by using several pseudo-literals:

- SYSTEM_USER always represents the operating system's identification of the user.
- SESSION_USER is the identity of the invoker, as defined above under **Invoker's Rights**.
- CURRENT_USER (or the equivalent USER) is the user name on which access is based; it is the invoker (SESSION_USER) if the program is operating under invoker's rights, or the definer (specified in DECLARE AUTHORIZATION) if the program is operating under definer's rights.

These pseudo-literals are discussed as syntactic elements of SQL in **Authorisation Pseudo-literals** on page 50.

2.6.4 Privileges

A **privilege** authorises a user to perform a given category of action on a specified object. SQL statements report an error if the current user does not have appropriate privilege for the operation. The privileges are:

- INSERT, DELETE, SELECT and UPDATE, which apply to all tables
- the REFERENCES privilege, which applies only to base tables
- the USAGE privilege, which applies to character sets, collations and translations.

Privileges are derived from the following sources:

- The owner of an object other than a viewed table has all privileges that apply to an object of that type.
- The owner of a viewed table has all applicable privileges that the user has on all tables from which the viewed table is derived. (The owner of a viewed table may lack some privileges on the table despite being the owner, if it was created from a table owned by another user.)
- If a user holds a **grantable privilege** on an object, it means that user can grant that privilege to, or revoke it from, any user.
  - — All the privileges the owner of an object other than a viewed table holds on it are grantable.
  - — A privilege the owner of a viewed table holds on the view is grantable if the owner holds that grantable privilege on all the tables from which the view is derived.
  - — Whenever a user grants a privilege on an object, the user may specify WITH GRANT OPTION, which means that the privilege granted to the recipient is a grantable privilege (thereby allowing the recipient to grant the same privilege in turn).
  - — The holder of a grantable privilege can grant it to, or revoke it from, PUBLIC, specifying a minimum privilege for all current and future users on the object. A user can perform an operation if either that user or PUBLIC has been granted the applicable privilege.

A user's privileges also determine which rows in certain system views are visible to the user.

Each granted privilege references the grantor. This reference appears in the system views. The grantor can revoke the privilege. (For the privileges that an owner of an object has on that object, the grantor is the fictitious user **_SYSTEM**. This user name appears in the system views, indicating that the privilege cannot be revoked.)

A specific privilege granted by multiple grantors is effectively multiple privileges, each of which must be revoked before the recipient loses that specific privilege.

Column Privileges

The REFERENCES and UPDATE privileges can be specified on individual columns of a table. Granting these privileges on an entire table results in separate grants on each column of the table. (The recipient will also have the privilege on any column of the table subsequently created.) The grantor can revoke the privilege on some or all of the columns as well as on the table.

2.7 Transactions

A **transaction** is a sequence of executable SQL statements that is atomic with respect to recovery and concurrency. With one exception described below, changes that an application makes to a database can be perceived by that application but cannot be perceived by other applications unless and until the original transaction ends with a COMMIT statement. (Many aspects of transactions are implementation-defined; see Section 7.4 on page 183 and Section 7.5 on page 184.)

A transaction starts (becomes active) when a program that does not already have an active transaction executes an SQL statement that operates on a database. The program can end the transaction by executing one of these statements:

- COMMIT enacts all changes made to the database during the transaction.
- ROLLBACK cancels all changes made to the database during the transaction.

If the program ceases execution without ending the transaction, then according to undefined criteria, either a COMMIT or ROLLBACK occurs.

It is implementation-defined whether a transaction can span SQL servers. That is, it is implementation-defined whether the application can execute CONNECT or SET CONNECTION while a transaction is active. If an implementation lets a transaction span SQL servers, the scope of a COMMIT or ROLLBACK statement is implementation-defined.

The execution of an SQL statement within a transaction has no effect on the database other than the effect stated in the description of that SQL statement.

2.7.1 Transaction Attributes

A transaction has the following attributes:

- **Access mode**

  A transaction's access mode is either read-only or read-write. The **read-only** access mode applies only to tables. Even if the relevant tables and cursors are updatable, a DELETE, INSERT or UPDATE statement fails if it is within a transaction with the read-only access mode. Changes to metadata (data definition statements) are also disallowed in such transactions.

  In a transaction with the **read-write** access mode, the above restriction does not apply, but updates to metadata and data may be prevented by other restrictions in this specification such as a non-updatable cursor or table or lack of access rights.

- **Isolation level**

  A transaction's isolation level is either READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ or SERIALIZABLE. The isolation level specifies the degree to which the operations on metadata or data interact with the effects of concurrent transactions discussed in Section 2.7.2 on page 29.

- **Size of Diagnostics Area**

  The size of the diagnostics area (the number of diagnostic records that it can contain) is also a transaction attribute.

In SQL, the application does not specify the attributes when starting a transaction. Instead, the SET TRANSACTION statement specifies the attributes to be used in the next transaction.

2.7.2 Concurrent Transactions

Concurrent transactions are transactions begun by different programs that gain access to the same metadata or data and that overlap in time. All concurrent transactions are by default guaranteed to be serialisable; that is, any interaction between the transactions occurs only in ways that guarantee that the result is the same as some serial sequence of the same transactions. This corresponds to the SERIALIZABLE isolation level defined in the International Standard. It is the highest isolation level.

At lower isolation levels, one or more of the following phenomena may occur during the execution of the following transactions:

- **Phantoms**

  Selecting rows a second time with the same search conditions may retrieve a different set of rows. The second set may include ‘‘phantom’’ rows that another, concurrent transaction inserted after the first retrieval.

- **Non-repeatable reads**

  Reading the same row a second time may produce a different result, if the row is updated or deleted by another concurrent transactions.

- **Dirty reads**

  All transactions perceive changes to the database even before they are committed.

The various transaction isolation levels are defined below as guarantees that certain of the above phenomena are not possible:

Isolation Level	Dirty Read	Non-repeatable Read	Phantom
SERIALIZABLE	Not possible	Not possible	Not possible
REPEATABLE READ	Not possible	Not possible	Possible
READ COMMITTED	Not possible	Possible	Possible
READ UNCOMMITTED	Possible	Possible	Possible

Table 2-1 Database Phenomena Possible at Various Transaction Isolation Levels

When an application specifies an isolation level, the implementation prepares to conduct the next transaction so that the corresponding phenomena are not possible in that transaction, from the point of view of the application, based on database operations performed by other concurrent applications.

Specification of higher isolation levels may result in reduced concurrency.[8]

It is implementation-defined whether all four isolation levels are available to the application. In all cases, however, the implementation provides at least the guarantees specified in the table above — that is, provides an isolation level at least as high as the application requested.

8. For example, a primitive but valid method of guaranteeing serialisability would be to actually serialise concurrent transactions. Users and implementors may evaluate the consequences of each isolation level by performance measurement of the database product.

Additional Guarantees

Regardless of the transaction isolation level, all transaction work is guaranteed to be atomic (which means that the results of the transaction's execution are either all committed or all rolled back) and it is guaranteed that no updates are lost. In addition, it is guaranteed that none of the phenomena listed above occur (as in the SERIALIZABLE isolation level) during the following operations:

- the implied reading of metadata definitions that must be performed in the execution of an SQL statement
- the checking of integrity constraints
- the execution of referential actions associated with referential constraints.

Chapter 3

Common Elements

This chapter defines common syntactic elements of SQL. Definitions of SQL syntax in the remainder of this specification rely on the definitions in this chapter.

3.1 Notation and Language Structure

3.1.1 Notation

The following syntactic notation is used to define the SQL syntax, both in the **SYNOPSIS** sections throughout this specification, and in Appendix A:

- Words shown in upper case, such as COUNT, are literal syntactic elements of SQL and must appear as shown.
- Words in italics, including hyphens (for example, *search-condition*), are descriptive terms for syntactic elements. The term may appear in the text to refer to the element. When two elements with the same syntactic form have different semantic rules, they may have separate names such as *field-name-1* and *field-name-2*.
- Square brackets [ ] enclose syntax that is optional.
- Vertical bars separate mutually-exclusive options.
- An ellipsis (. . .) means that the syntax enclosed by the immediately preceding pair of braces or brackets may be repeated.
- Braces { } delimit a range of syntax to which vertical bars or ellipses apply.
- All other symbols (such as parentheses) are literal syntactic elements of SQL and must appear as shown.

Errors and Warnings

Any SQL statement can result in unqualified success or can result in a variety of warning or error outcomes. Section 4.5 on page 90 discusses these outcomes. The preferred way for an application to test the outcome of an SQL statement is to test the SQLSTATE status variable.

Anywhere this specification contains a statement imposing a requirement on the application for proper invocation of an SQL statement, it means the implementation checks that it was called properly. The specification also indicates the SQLSTATE value by which the implementation reports a violation of this rule. For example:

The cursor must be open ('**24**000').

This imposes a requirement on the application's use of the described SQL statement, and asserts that an X/Open-compliant implementation of the described statement verifies that the cursor is open and sets SQLSTATE to '**24**000' if the cursor is not open.

Syntax Errors

Violation of any syntactic assertion in this specification that does not specify a SQLSTATE value is a **Syntax error** ('**42**000'). The general rule in Section 5.2.1 on page 99 notes that this SQLSTATE applies to the syntax rules in the **SYNOPSIS** and **DESCRIPTION** sections of SQL statement descriptions.

3.1.2 Language Structure

An embedded SQL program resembles words, symbols and blank space. The technical terms for these language elements are non-delimiter tokens, delimiter tokens, and separators, respectively. These technical terms are defined below.

SQL Character Set

By default, an embedded SQL program is composed of characters from a character set called the **SQL character set**. (Syntax specified later in this section can be used to indicate that parts of an SQL program use different character sets.) All X/Open-compliant implementations support the SQL character set. It contains at least the 26 English upper-case letters, the 26 English lower-case letters, the 10 decimal digits, a blank character, a newline indication, and the following symbols required by the SQL language:

```
, ( ) < > . : = * + - / ? _ % |
```

The SQL character set is a proper subset of the X/Open Portable Character Set (see the referenced X/Open **Internationalisation Guide**).

The internal coding of these characters is undefined. The transfer format for these characters is discussed separately, depending on the interchange method; see the X/Open **RDA** Specification.

Separators and SQL Comments

A **separator** is a space, a newline indication, or an SQL comment. (Spaces and newline indications are components of the implementation's character set; SQL comments are defined below.) A separator may be followed by another separator or a token. If the newline indication is a character, it cannot be used to form a token.

An **SQL comment** begins with two or more consecutive hyphens and runs up to the next newline indication.[9]

Tokens

A **token** is either a non-delimiter token or a delimiter token.

A **non-delimiter token** is one of the words in an embedded SQL program. A non-delimiter token is a user-defined name, a host identifier, a keyword, a numeric-literal or a national character string literal. Two non-delimiter tokens may not be run together; a separator or a delimiter token must appear between them (in conformance with the syntax rules that apply to the context).

9. Consecutive hyphens inside a delimited user-defined name or character-string literal do not begin an SQL comment.

A **delimiter token** is a character-string-literal, a delimited user-defined name, one of the following characters:

, ( ) < > . : = * + – ? and /

or one of the following character-pairs:

<> >= <= and ||

A user-defined name or a keyword may contain lower-case letters. In these tokens, a lower-case letter is equivalent to the corresponding upper-case letter.

Literals are described in Section 3.4 on page 48, and host identifiers in Chapter 4; user-defined names and keywords are described below.

User-defined Names

A **user-defined name** is a non-delimiter token that is selected by the writer of the embedded SQL program as a name. The types of objects that can be named are listed in **Types of User-defined Name** on page 34.

The first character of a user-defined name must be a letter.[10] Any subsequent character of a user-defined name must be a letter, a digit or the underscore character.

The maximum length of a user-defined name is 18 characters. For certain types of user-defined name, some implementations impose further restrictions; see Section 7.3 on page 183.

A user-defined name must not be one of the reserved words listed in Section 3.1.6 on page 37.

Delimited User-defined Names

User-defined names may be enclosed in double quotes ("). These are called **delimited** user-defined names. The text within the double quotes is called the **body** of the user-defined name. To specify a double quote as a component character of the user-defined name, two consecutive double quotes appear in the body.

Delimited user-defined names are special in three respects:

- They can contain characters that the syntax of SQL would not permit or would interpret specially if they occurred without delimitation, including blanks in any or all positions.
- They can be identical to a reserved keyword.
- They are treated in a case-sensitive manner, allowing the application to specify upper-case and lower-case letters without any implicit conversion by the implementation.

The value of a delimited user-defined name is the body (defined above) of the name, after converting each instance of two consecutive double quotes to a single double quote. (The value is visible to applications through the system views in Chapter 6.) The length limit specified in

10. Section 3.1.5 on page 36 describes user-defined names that use character sets other than the SQL character set. In some character sets, a ‘‘letter’’ may be a syllable or an ideogram. In each such character set, the list of characters available for use in user-defined names is implementation-defined. Characters that are SQL delimiter tokens or separators are never in this list.

When a user-defined name is in a character set that has corresponding upper-case and lower-case characters, (1) an upper-case character is considered equivalent to the corresponding lower-case character and (2) lower-case characters are converted to the corresponding upper-case character before a character string is stored in the metadata. There is no other situation in any character set in which such an equivalence or automatic conversion occurs in SQL. Equivalence or automatic conversion does not apply to database data.

User-defined Names applies to this value.

Types of User-defined Name

The following syntactic elements are user-defined names:[11]

unqualified-base-table-name
: The name of a base table. It can be qualified, as discussed in Section 3.1.3 on page 35.

unqualified-character-set-name
: The name of a character set. The syntax defined in Section 3.1.5 on page 36 for specifying the character set cannot be applied to the name of a character set; character set names are always in the SQL character set.

unqualified-collation-name
: The name of a collation.

unqualified-column-name
: The name of a column, which must be unique within its table. It can be qualified, as discussed in Section 3.1.4 on page 36.

unqualified-conversion-name
: The name of a conversion.

correlation-name
: An alias for a *table-name*. It is specified by being paired with the *table-name* in a FROM clause. Its function is described in Section 3.11.5 on page 76. It is used in *table-reference*s, which are defined in Section 3.11.3 on page 75.

cursor-name
: The name of a cursor. It must be unique within a host program.

unqualified-index-name
: The name of an index. It can be qualified, as discussed in Section 3.1.3 on page 35.

product-name
: A name that identifies an implementation's SQL version. It appears only in a *vendor-escape-clause* (see Section 7.2 on page 181).

statement-name
: A name that identifies an SQL statement that is prepared for execution.

unqualified-translation-name
: The name of a translation.

user-name
: The name of a user (not necessarily the login name). It must be unique within the names of authorised users of the database.

vendor-name
: A name that identifies a supplier of an SQL implementation. It appears only in a *vendor-*

11. The *user-defined-name*s in this document are generally called identifiers in the International Standard. As presented in this section, this document uses the suffix *–name* in syntactic elements for classes of user-defined-name. The prefix *unqualified–* denotes the element without qualification; when the prefix is omitted, it denotes the user-defined-name with any optional qualification allowed for that class. By contrast, the International Standard applies a suffix *–name* to qualified identifiers and *–identifier* to unqualified identifiers.

escape-clause (see Section 7.2 on page 181).

unqualified-viewed-table-name
The name of a viewed table. It can be qualified, as discussed in Section 3.1.3.

The *host-identifier* syntactic element described in Chapter 4 is not a user-defined name as specified in this section.

3.1.3 Format of Object Qualification

Any of the syntactic elements defined above that use the prefix ''*unqualified-*'' may be uniquely identified (qualified) by preceding it with an *object-qualifier*, which specifies the catalog and schema in which the object resides.[12] An *object-qualifier* has the following syntax:[13]

```
[catalog-name.]schema-name
```

The *catalog-name* and *schema-name* are each syntactically a *user-defined-name*. Implementation-defined object naming systems may also be supported. In all cases, use of the *object-qualifier* is an application option.

This specification defines the additional SQL syntax elements listed in the right-hand column of the following table as the corresponding element from the left-hand column preceded by: *object-qualifier*.

Unqualified Element	**Qualified Element**
unqualified-base-table-name	*base-table-name*
unqualified-character-set-name	*character-set-name*
unqualified-collation-name	*collation-name*
unqualified-conversion-name	*conversion-name*
unqualified-index-name	*index-name*
unqualified-translation-name	*translation-name*
unqualified-viewed-table-name	*viewed-table-name*

Since SQL syntax usually allows object qualification, the elements in the right-hand column are the elements that typically appear in this specification.

A *table-name* is either a *base-table-name* or a *viewed-table-name*.

Default Qualification

For each connection, there is a default catalog name and a default schema name, as described in Section 2.4.1 on page 19. In any identifier that could be qualified, if there is no catalog name, then the default catalog is implicit; if there is no schema name, then the default schema is implicit.

The application can change the default catalog name and default schema name by using the SET CATALOG and SET SCHEMA statements described in Section 5.8 on page 147.

12. Catalogs are an optional feature, as discussed in **Three-part Naming** on page 20. An application can determine whether the implementation supports catalog names by examining the CATALOG_NAME attribute in the SERVER_INFO system view (see *SERVER_INFO* on page 166).

13. The only object qualification scheme specified in the previous issue of this specification provided that the *object-qualifier* is always a *schema-name* that is identical to the *user-name* of the owner of the schema.

The initial values of the default catalog name and default schema name are implementation-defined.[14]

3.1.4 Format of Column Qualification

A *column-identifier* names a column and must be unique within the names of the columns of the associated table. A *column-identifier* may be uniquely identified by being qualified by a *table-name* or *correlation-name*.

The construct [{*table-name* | *correlation-name*}.]*unqualified-column-name* is referred to as a *column-name*.

3.1.5 Specifying the Character Set

Certain types of user-defined name (listed below) can use character sets other than the SQL character set. The character set that is used for such a name is defined by the first applicable element from the following list:

- The user-defined name can indicate the character set they use.
- The syntax SQL NAMES ARE in an SQL declare section (see Section 4.2 on page 84) can declare a default character set for other such user-defined names in the compilation unit.
- Otherwise, the default character set for such user-defined names is an implementation-defined character set.

Format

The following syntax specifies a character set for an applicable user-defined name:

```
_character-set-name user-defined-name
```

That is, the *user-defined-name* is preceded by the underscore character, a *character-set-name* (which never begins with the underscore character and always uses the SQL character set), and one or more separators.

Applicability

A character set can be specified for the following syntactic elements:

unqualified-base-table-name	*correlation-name*	*statement-name*
catalog-name	*cursor-name*	*unqualified-translation-name*
unqualified-collation-name	*unqualified-index-name*	*user-name*
unqualified-column-name	*product-name*	*vendor-name*
unqualified-conversion-name	*schema-name*	*unqualified-viewed-table-name*

The same syntax is also valid for the *character-string-literal* defined in Section 3.4 on page 48.

14. For philosophical consistency with previous versions of this specification, X/Open recommends that the initial value of the default schema name be the value of USER.

3.1.6 Keywords

Keywords are predefined words that are required literally in some situations in SQL syntax. The syntax definitions in this specification show keywords in upper-case letters.

Reserved Words

The following keywords are also **reserved words**: Portable applications must not use them as user-defined names. An implementation may allow the use of any or all of these keywords, but flags such usage:

ABSOLUTE *	CROSS *	GLOBAL *	NOT	SOME
ACTION *	CURRENT	GO	NULL	SPACE *
ADD	CURRENT_DATE	GOTO	NULLIF *	SQL
ALL	CURRENT_TIME	GRANT	NUMERIC	SQLCA *
ALLOCATE	CURRENT_TIMESTAMP	GROUP	OCTET_LENGTH	SQLCODE *
ALTER	CURRENT_USER	HAVING	OF	SQLERROR
AND	CURSOR	HOUR	ON	SQLSTATE *
ANY	DATE	IDENTITY *	ONLY	SQLWARNING
ARE *	DAY	IMMEDIATE	OPEN	SUBSTRING
AS	DEALLOCATE	IN	OPTION	SUM
ASC	DEC	INCLUDE	OR	SYSTEM_USER
ASSERTION *	DECIMAL	INDEX	ORDER	TABLE
AT *	DECLARE	INDICATOR	OUTER	TEMPORARY *
AUTHORIZATION	DEFAULT	INITIALLY *	OUTPUT *	THEN *
AVG	DEFERRABLE *	INNER	OVERLAPS	TIME
BEGIN	DEFERRED *	INPUT	PAD *	TIMESTAMP
BETWEEN	DELETE	INSENSITIVE *	PARTIAL *	TIMEZONE_HOUR *
BIT *	DESC	INSERT	POSITION	TIMEZONE_MINUTE *
BIT_LENGTH *	DESCRIBE	INT	PRECISION	TO
BOTH	DESCRIPTOR	INTEGER	PREPARE	TRAILING
BY	DIAGNOSTICS	INTERSECT *	PRESERVE *	TRANSACTION
CASCADE	DISCONNECT	INTERVAL	PRIMARY	TRANSLATE
CASCADED *	DISTINCT	INTO	PRIOR *	TRANSLATION *
CASE *	DOMAIN *	IS	PRIVILEGES	TRIM
CAST	DOUBLE	ISOLATION	PROCEDURE *	TRUE *
CATALOG *	DROP	JOIN	PUBLIC	UNCOMMITTED
CHAR	ELSE *	KEY	READ	UNION
CHARACTER	END	LANGUAGE *	REAL	UNIQUE
CHARACTER_LENGTH	END-EXEC *	LAST *	REFERENCES	UNKNOWN *
CHAR_LENGTH	ESCAPE	LEADING	RELATIVE *	UPDATE
CHECK	EXCEPT *	LEFT	REPEATABLE	UPPER
CLOSE	EXCEPTION	LEVEL	RESTRICT	USAGE *
COALESCE *	EXEC	LIKE	REVOKE	USER
COLLATE	EXECUTE	LOCAL *	RIGHT	USING
COLLATION	EXISTS	LOWER	ROLLBACK	VALUE
COLUMN	EXTERNAL *	MATCH *	ROWS *	VALUES
COMMIT	EXTRACT	MAX	SCHEMA	VARCHAR
COMMITTED	FALSE *	MIN	SCROLL *	VARYING
CONNECT	FETCH	MINUTE	SECOND	VIEW
CONNECTION	FIRST *	MODULE *	SECTION	WHEN *
CONSTRAINT *	FLOAT	MONTH	SELECT	WHENEVER
CONSTRAINTS *	FOR	NAMES *	SERIALIZABLE	WHERE
CONTINUE	FOREIGN	NATIONAL	SESSION *	WITH
CONVERT *	FOUND	NATURAL *	SESSION_USER	WORK
CORRESPONDING *	FROM	NCHAR	SET	WRITE *
COUNT	FULL *	NEXT *	SIZE *	YEAR
CREATE	GET	NO *	SMALLINT	ZONE *

* X/Open SQL does not use these keywords, but they are defined in the International Standard.

Non-reserved Words

The following keywords are defined in X/Open SQL but are not reserved words. Applications may use them as user-defined names:

CLASS_ORIGIN	RETURNED_LENGTH
CONNECTION_NAME	RETURNED_SQLSTATE
DATA	ROW_COUNT
LENGTH	SCALE
MESSAGE_LENGTH	SERVER_NAME
MESSAGE_TEXT	SUBCLASS_ORIGIN
MORE	TYPE
NAME	UNNAMED
NULLABLE	

(Some implementations may reserve additional keywords; see Section 7.3 on page 183.)

3.2 Generic Data Types

A data type is a set of representable values. A data type restricts the contents or representation of a value.

There are four **generic data types**: character string, date/time, interval and numeric. Non-null values within a generic data type can be assigned to one another and compared to one another. Generally, values cannot be assigned to or compared with values of another generic data type. (The null value is treated specially in comparisons and assignment; see Section 3.7 on page 54.) However, the application may explicitly convert values from one generic data type to another using the CAST function (see Section 3.9.6 on page 65).

Within each generic data type, there are a variety of **named data types**. A named data type has specific characteristics and SQL syntax. The application identifies a named data type using an SQL keyword and, for some named data types, may specify additional attributes such as length and precision. Named data types are classified as shown below, and are defined in the following sections.

Generic Data Type	**Subclassification**	**Named Data Types**
Character String (see Section 3.2.1 on page 40)	Fixed-Length	CHARACTER NATIONAL CHARACTER
	Variable-Length	CHARACTER VARYING NATIONAL CHARACTER VARYING
Numeric (see Section 3.2.2 on page 41)	Exact Numeric	DECIMAL INTEGER NUMERIC SMALLINT
	Approximate Numeric	DOUBLE PRECISION FLOAT REAL
Date/Time (see Section 3.2.3 on page 42)		DATE TIME TIMESTAMP
Interval (see Section 3.2.4 on page 44)	Long Interval	INTERVAL YEAR INTERVAL MONTH INTERVAL YEAR TO MONTH
	Short Interval	INTERVAL DAY INTERVAL HOUR INTERVAL MINUTE INTERVAL SECOND INTERVAL DAY TO HOUR INTERVAL DAY TO MINUTE INTERVAL DAY TO SECOND INTERVAL HOUR TO MINUTE INTERVAL HOUR TO SECOND INTERVAL MINUTE TO SECOND

Table 3-1 Classification of Data Types

Notes on the Tables

The named data types are presented in tables in the following sections. The tables specify minimum values for ranges, precisions and lengths that application developers can assume all compliant implementations support. Implementations may support larger ranges, precisions and lengths than shown.

The tables show the basic syntax of each named data type. Information about optionality of the syntax, and about valid synonyms for keywords shown in the table, appears following the table.

3.2.1 Character String

A character string is a sequence of characters taken from a specific character set (see below). A character string has a length, which is the number of characters in the sequence.

A character string is either a fixed-length or variable-length character string. For variable-length strings, the length may be zero.

The following named character-string data types are defined:

Named Data Type	**Description**
CHARACTER(*n*) $1 \leq n \leq 254$	Character string of fixed length *n*.
CHARACTER VARYING(*n*) $1 \leq n \leq 254$	Variable-length character string with a maximum string length *n*.
NATIONAL CHARACTER(*n*) $1 \leq n \leq 254$	Character string in the national character set of fixed length *n*.
NATIONAL CHARACTER VARYING(*n*) $1 \leq n \leq 254$	Variable-length character string in the national character set with a maximum string length *n*.

Table 3-2 Named Character String Data Types

The application may omit the parenthesised length specification (*n*) following CHARACTER. The default length is 1.

X/Open SQL also allows CHAR as a synonym for CHARACTER, NCHAR as a synonym for NATIONAL CHARACTER and VARCHAR as a synonym for CHARACTER VARYING. However, the forms NATIONAL VARCHAR and NVARCHAR are invalid.

Character Set

The character set used by character-string variables and columns of tables in the database is defined by the first applicable element from the following list:[15]

- The elements can indicate the character set they use. This is done by preceding the element by an underscore, then a character set name, then one or more separators. This is the same syntax given for specifying the character set of user-defined names in Section 3.1.5 on page 36.

15. The list is similar in structure, but not identical, to the list given for user-defined names in Section 3.1.5 on page 36.

However, if the data type of an element is NATIONAL CHARACTER, NATIONAL CHARACTER VARYING or one of the synonyms defined above, then its character set is defined to be the national character set and the explicit syntax for specifying a character set cannot be used.

- Other elements use the default character set if one has been defined for the schema.
- Otherwise, the default character set for such elements is an implementation-defined character set.

3.2.2 Numeric

Each **numeric** value is either exact or approximate.

- An exact numeric value has a precision and scale. The **precision** is a positive integer that determines the total number of significant decimal digits. The **scale** is a non-negative integer representing the number of significant decimal digits to the right of the decimal point.
- An **approximate numeric** value consists of a mantissa and an exponent. The **mantissa** is a signed numeric value and the **exponent** is a signed integer value that specifies the magnitude of the mantissa. An approximate numeric value has a **precision**, which represents the number of significant bits in the mantissa.

The following named numeric data types are defined:

Named Data Type	Description
DECIMAL(*p*,*s*) $1 \leq p \leq 15;\ 0 \leq s \leq p$	Exact numeric, signed, precision *p*, scale *s*.
DOUBLE PRECISION	Approximate numeric, signed, mantissa precision 47 bits; value zero or absolute value 10^{-38} to 10^{38}.
FLOAT(*p*) $1 \leq p \leq 47$	Approximate numeric, signed, mantissa precision at least *p*; value zero or absolute value 10^{-38} to 10^{38}.
INTEGER	Exact numeric, signed, precision 10 digits, scale zero; value from –2,147,483,648 through 2,147,483,647
NUMERIC(*p*,*s*) $1 \leq p \leq 15;\ 0 \leq s \leq p$	Exact numeric, signed, precision *p*, scale *s*.
REAL	Approximate numeric, signed, mantissa precision 21 bits; value zero or absolute value 10^{-38} to 10^{38}.
SMALLINT	Exact numeric, signed, precision 5 digits, scale zero; value from –32,768 through 32,767.

Table 3-3 Named Numeric Data Types

Following DECIMAL and NUMERIC, the application may omit the scale, writing (*p*) as a synonym for (*p*,0), or omit the precision and scale specifications entirely. The application may likewise omit the precision specification following FLOAT. The default scale is 0. Default precisions for DECIMAL, NUMERIC, and FLOAT data are discussed in Section 7.1 on page 175.

X/Open SQL also allows DEC as a synonym for DECIMAL and INT as a synonym for INTEGER.

3.2.3 Date/Time

A date/time value contains some or all of the fields YEAR, MONTH, DAY, HOUR, MINUTE and SECOND. These fields always occur in the order listed here, which is from most significant to least significant. Each field is an integer except that the SECOND field can have an integer fractional seconds component. Date/time values are not character strings and cannot be used interchangeably with character strings in comparison or assignment unless explicitly converted by use of the CAST function.

The three classes of date/time data type are as follows:

DATE	Contains the YEAR, MONTH and DAY fields.
TIME	Contains the HOUR, MINUTE and SECOND fields.
TIMESTAMP	Contains the YEAR, MONTH, DAY, HOUR, MINUTE and SECOND fields.

A date/time data item may have the null value, but if a date/time data item is non-null, then all fields required by the item's class must be non-null.

Each field that is present in a date/time value has a non-negative value that denotes a date (using the Gregorian calendar) and/or a time (using a 24-hour clock). The limit on the value of each field is that normally imposed by the Gregorian calendar and the 24-hour time system:

$0001 \leq \text{YEAR} \leq 9999$
$01 \leq \text{MONTH} \leq 12$
$01 \leq \text{DAY} \leq 31$, constrained by MONTH and YEAR
$00 \leq \text{HOUR} \leq 23$
$00 \leq \text{MINUTE} \leq 59$
$00 \leq \text{SECOND} < 62$[16]

A date/time is only meaningful in conjunction with a time zone.[17] However, date/time values in X/Open SQL (in a database or in an application program) do not specify a time zone. Neither clients nor servers are required to account for different time zones or convert date/time values from one time zone to another. If the application or the user cannot infer a time zone (for example, from the context of the user's session) then the date/time value is ambiguous.[18]

16. The normal constraint on the value of SECOND, including any fractional part, is $00 \leq \text{SECOND} < 60$. However, as many as two leap seconds are added to certain minutes. No implementation is required to test a date/time value whose SECOND field is 60 or greater to ensure that the designated minute actually contained leap seconds.

17. Time zones are political divisions of the earth's surface within which the time is the same and approximates the time as it would be measured based on the sun. However, the law may specify different time offsets within a time zone based on political subdivisions or varying over the course of the year.

18. Applications that have to reconcile date/time values that represent times in different time zones should associate time zone information with each date/time value. However, the application should do so in a way that can accommodate database products in which date/time values contain time zone information.

The following named date/time data types are defined:

Named Data Type	**Description**
DATE	Date/time value describing a date. Its length is 10 and it contains a YEAR, MONTH and DAY field (constrained as described in Section 3.2.3 on page 42) in the format: `YYYY–MM–DD`
TIME(*p*)	Date/time value describing the time in an unspecified day, with precision *p*. Its length is 8 (or 9+*p* if $p > 0$) and it contains an HOUR, MINUTE, and SECOND field (constrained as described in Section 3.2.3 on page 42) in the format: `HH:MM:SS[.F]` where `F` is the fractional part.
TIMESTAMP(*p*)	Date/time value describing both a date and a time, with precision *p*. Its length is 19 (or 20+*p* if $p > 0$) and it contains a YEAR, MONTH, DAY, HOUR, MINUTE, and SECOND field (constrained as described in Section 3.2.3 on page 42) in the format: `YYYY–MM–DD HH:MM:SS[.F]` where `F` is the fractional part.

Table 3-4 Named Date/Time Data Types

The application may omit the parenthesised specification of seconds precision (*p*) following TIME and TIMESTAMP. The default seconds precision is 0 for TIME and 6 for TIMESTAMP.

3.2.4 Interval

An interval value is one of the following:

- one or both of the fields YEAR and MONTH
- one or more consecutive fields from the set: DAY, HOUR, MINUTE and SECOND.

Interval Precision

Named interval data types (which are enumerated in Table 3-5 on page 45) have an **interval precision**, which is a list of all the fields that values of that data type may contain. For example, in a value of type INTERVAL HOUR TO SECOND, the date / time precision is HOUR, MINUTE, SECOND.

Leading Precision

Conceptually, an interval is a signed numeric quantity comprising a specific set of fields. The fields have the same ordering, relative significance, and Gregorian-calendar constraints as described above for date / time values except as described below.

In all intervals, the high-order field is not constrained by the Gregorian calendar. Named interval data types have a **leading precision**, which is the number of decimal digits that the high-order field can accommodate.

For example, in an INTERVAL HOUR TO MINUTE, the MINUTE field is constrained as described above but the HOUR field is constrained only by the leading precision. If interval arithmetic generates an intermediate value of 01:61, the implementation converts it to 02:01 before making it visible to the application.

Seconds Precision

Any interval data type that has a SECOND field has a **seconds precision**. This is the number of decimal digits allowed in the fractional part of the SECOND value.

In the named data type INTERVAL SECOND, the SECOND field is the only field and has both a leading precision and a seconds precision.

Interval Qualifier

The syntactic element *interval-qualifier* follows every use of the keyword INTERVAL to specify the interval precision (the set of available fields). Valid *interval-qualifiers* are listed in Table 3-5 on page 45.

An *interval-qualifier* specifies or implies the interval precision, leading precision, and seconds precision (if applicable) of the value.

18. Syntax using the *interval-qualifier* also appears in **Subtracting Two Date/Times** on page 58, and as a qualification for dynamic parameters (see Section 3.9.5 on page 64 and in the COLUMNS system view (see *COLUMNS* on page 160).

Named Interval Data Types

A named interval data type consists of the word INTERVAL followed by one of the following *interval-qualifier*s:

interval-qualifier	**Description of Named Data Type**
YEAR(*p*)	Interval class describing a number of years, with leading precision *p*. It contains a YEAR field in the format: `YYYY`
MONTH(*p*)	Interval class describing a number of months, with leading precision *p*. It contains a MONTH field in the format: `MM`
YEAR(*p*) TO MONTH	Interval class describing a number of years and months, with leading precision *p*. Its format is: `YYYY-MM`
DAY(*p*)	Interval class describing a number of days. Its leading precision is *p*. It contains a DAY field in the format: `DD`
HOUR(*p*)	Interval class describing a number of hours, with leading precision *p*. It contains an HOUR field in the format: `HH`
MINUTE(*p*)	Interval class describing a number of minutes, with leading precision *p*. It contains a MINUTE field in the format: `MM`
SECOND(*p*, *s*)	Interval class describing a number of seconds, with leading precision *p* and seconds precision *s*. It contains a SECOND field in the format: `SS[.F]`
DAY(*p*) TO HOUR	Interval class describing a number of days and hours, with leading precision *p*. Its format is: `DD HH`
DAY(*p*) TO MINUTE	Interval class describing a number of days, hours and minutes, with leading precision *p*. Its format is: `DD HH:MM`
DAY(*p*) TO SECOND(*s*)	Interval class describing a number of days, hours, minutes, and seconds, with leading precision *p* and seconds precision *s*. Its format is: `DD HH:MM:SS[.F]`
HOUR(*p*) TO MINUTE	Interval class describing a number of hours and minutes, with leading precision *p*. Its format is: `HH:MM`
HOUR(*p*) TO SECOND(*s*)	Interval class describing a number of hours, minutes and seconds, with leading precision *p* and seconds precision *s*. Its format is: `HH:MM:SS[.F]`
MINUTE(*p*) TO SECOND(*s*)	Interval class describing a number of minutes and seconds, with leading precision *p* and seconds precision *s*. Its format is: `MM:SS[.F]`

Table 3-5 Named INTERVAL Data Types

The application may omit the parenthesised specification of leading precision (*p*). The default leading precision is 2, even if the high-order field is YEAR. The application may omit the parenthesised specification of seconds precision (*s*). The default seconds precision is implementation-defined. In the case of INTERVAL SECOND(*p*, *s*), the application may omit the seconds precision, writing (*p*) and implying the implementation-defined default seconds precision. The application can also omit both precision specifications.

Length of an Interval

The three types of precision determine the maximum length of any interval value. Length is expressed in positions. This is the same as a number of characters of the string representation of the interval (using a single-byte character set). The rules for computing an interval's maximum length are as follows:

- Allow two positions for every field in the interval that is not the high-order field.
- For the high-order field, allow the number of positions that is the leading precision.[19]
- Add one position for each separator between fields.
- If the seconds precision is nonzero, add that number of positions, plus one for the decimal point.

Applying these rules to each named interval data type produces the following lengths, where p is the leading precision and s is the seconds precision (specified or implied):

interval-qualifier	**Length of Named Data Type**
YEAR	p
MONTH	p
YEAR TO MONTH	$3 + p$
DAY	p
HOUR	p
MINUTE	p
SECOND	p (or $p + s + 1$ if $s > 0$)
DAY TO HOUR	$3 + p$
DAY TO MINUTE	$6 + p$
DAY TO SECOND	$9 + p$ (or $10 + p + s$ if $s > 0$)
HOUR TO MINUTE	$3 + p$
HOUR TO SECOND	$6 + p$ (or $7 + p + s$ if $s > 0$)
MINUTE TO SECOND	$3 + p$ (or $4 + p + s$ if $s > 0$)

Table 3-6 Lengths of Named INTERVAL Data Types

19. This algorithm, derived from the International Standard, does not allow for the minus sign that is present in the string representation of negative intervals. Application writers are advised that such a string representation could be one character longer than the ''maximum'' length.

3.3 Rules for Determining Data Types

Data Types of Table Columns

Each column of a base table is given a named data type when the column is created (that is, when the table is created or when the column is added to the table).

The data types of a derived table's columns may be generic data types or named data types:

- Columns of derived tables, defined by a *query-specification*, derive their data types from the *expressions* in the *query-specification* that define columns.
- Columns of a viewed table inherit the data types of the corresponding result columns of the *query-specification* that defines the view.
- Columns defined by a cursor specification inherit the data types of the corresponding result columns of the first *query-specification* in the cursor specification.

Data Types of Expressions

If an *expression* (see Section 3.9 on page 57) contains a single operand, the data type of the *expression* is the data type of that operand.

The data type of the result of an arithmetic operation depends on the data types of the two operands and is determined as follows:

- If the data type of either operand is approximate numeric, the data type of the result is approximate numeric with implementation-defined precision and range of magnitude.
- If the data type of the result is approximate numeric, the result must be within the implementation-defined range of magnitude.
- If the data type of both operands is exact numeric, the data type of the result is exact numeric with implementation-defined precision and range, and scale dependent on the operation as follows:
  - — If the operation is addition or subtraction, the scale is the larger of the scales of the two operands.
  - — If the operation is multiplication, the scale is the sum of the scales of the two operands.
  - — If the operation is division, the scale is implementation-defined.
- If the data type of the result is exact numeric and the operator is not division, the result must be exactly representable within its data type. If the data type of the result is exact numeric and the operator is division, the result must be representable within its data type without losing any leading significant digits.

The data type of a result of a set function depends on the function and on the type of its argument. (See Table 3-9 on page 63.)

The data type of a result of a date/time or interval computation is specified in Section 3.9.2 on page 57.

The data type of a result of the various string operations is specified in Section 3.9.3 on page 59.

The data type of an expression can be explicitly converted to another data type using the CAST function.

3.4 Literals

A *literal* is a sequence of characters representing a value. The classification of *literals* is according to the values that they represent.

Character-string Literals

A *character-string-literal* represents a character string and consists of a sequence of characters from a certain character set (see below) delimited at each end by the single quote character.

That is, it has the following format:

```
'{character}...'
```

Within the delimiters, a single quote character is represented by two consecutive single quote characters. The value of the *literal* is the value of the sequence of characters within the delimiters. The data type of a *character-string-literal* is fixed-length character string.

The rules for determining the character set of a *character-string-literal* is the same as for *user-defined-name*s (see Section 3.1.5 on page 36). For example, a *character-string-literal* can be preceded by syntax explicitly specifying a character set.

A separate form of *character-string-literal* is the national character string literal:

```
N'{character}...'
```

This is a *character-string-literal* where the character set is always the national character set. The character set cannot be specified explicitly for a national character set literal.

Numeric Literals

A *numeric-literal* represents a number and consists of a character string whose characters are selected from the digits 0 through 9, the plus sign, the minus sign, the decimal point and the character **E**.

A *numeric-literal* is either an *exact-numeric-literal* representing an exact numeric value or an *approximate-numeric-literal* representing an approximate numeric value.

An *exact-numeric-literal* has the following format:

```
[+|−]{unsigned-integer[.unsigned-integer]
      |unsigned-integer.
      |.unsigned-integer}
```

where *unsigned-integer* is defined as:

```
{digit}...
```

The data type of an *exact-numeric-literal* is exact numeric, its precision is the number of digits that it contains, and its scale is the number of digits to the right of the decimal point. The value of an *exact-numeric-literal* is derived from the normal mathematical interpretation of the specified notation.

An *approximate-numeric-literal* has the format *mantissa*E*exponent*, where the *mantissa* is an *exact-numeric-literal* and the *exponent* has the form:

```
[+|−]unsigned-integer
```

The data type of an *approximate-numeric-literal* is approximate numeric and its precision is the precision of its *mantissa*.

The value of an *approximate-numeric-literal* is the product of the value represented by the *mantissa* with the number obtained by raising the number 10 to the power represented by the *exponent*.

Date/Time Literals

A date/time literal represents a date/time value and consists of one of the keywords shown below, followed by text delimited at each end by a single quote character. It can have the following formats:

```
DATE 'date-value'
TIME 'time-value'
TIMESTAMP 'date-value space time-value'
```

The quoted text contains the fields specified in Section 3.2.3 on page 42 for the respective date/time subtype, in the order and with the limits described there, using the delimiters shown in Table 3-4 on page 43 ('**22**007').

A *date-value* has the following format:

```
year-value – month-value – day-value
```

A *time-value* has the following format:

```
hour-value : minute-value : second-value
```

where *year-value*, *month-value*, *day-value*, *hour-value* and *minute-value* are *unsigned-integer*s and *second-value* has the following format:

```
unsigned-integer[.unsigned-integer]
```

and –, : and *space* above represent single characters determined by the character set in use.

Interval Literals

An interval literal represents a date/time value and consists of the INTERVAL keyword followed by text delimited at each end by a single quote character. It can have the following formats:

```
INTERVAL [+|–] 'interval-value' interval-qualifier
```

The text of *interval-value* must be a valid representation of a value of the named interval data type specified by *interval-qualifier*. The text contains a decimal value for every field implied by *interval-qualifier*.

- If the interval precision includes the fields YEAR and MONTH, values of these fields are separated by the minus sign.
- If the interval precision includes the fields DAY and HOUR, values of these fields are separated by a space.
- If the interval precision includes HOUR and lower-order fields, values of these fields are separated by the colon.
- No field value can be more than two digits long except that:
  - — The value of a YEAR field can be 4 digits long.
  - — The value of the high-order field can be as long as the leading precision specified in *interval-qualifier*.
  - — The SECOND field can have a fractional part whose maximum length is specified or implied by *interval-qualifier*.

3.4.1 Pseudo-literals

A *pseudo-literal* is a syntactic element that can be used as a literal. Rather than a constant value, it returns variable information obtained from the implementation.

Current Date and Time

The following keywords denote date / time values:[20]

`CURRENT_DATE`
: A DATE value denoting the current date.

`CURRENT_TIME[(`*precision*`)]`
: A TIME value denoting the current time.

`CURRENT_TIMESTAMP[(`*precision*`)]`
: A TIMESTAMP value denoting the current date and time.

If specified, *precision* determines the seconds precision of the time or timestamp value returned.

If a single SQL statement references several instances of CURRENT_DATE, CURRENT_TIME and CURRENT_TIMESTAMP, they all return values that denote the same date and time.

Authorisation Pseudo-literals

Several pseudo-literals describe users of the database system. These pseudo-literals represent character strings using characters from the character set SQL_TEXT:

```
CURRENT_USER
SESSION_USER
SYSTEM_USER
USER
```

The contents of these pseudo-literals are defined and their uses are discussed in the overview of authorisation in Section 2.6.3 on page 26.

SYSTEM_USER, SESSION_USER, CURRENT_USER and USER are explicitly permitted in a small number of cases where pseudo-literals in general are not valid.

20. The International Standard classifies these three pseudo-literals as *date/time value functions*. In X / Open SQL, they are valid anywhere other date / time literals are, with one exception noted for column constraints in Section 5.3.7 on page 107.

3.5 Assignment

Character String Assignments

The following rules apply when assigning a non-null character string of length *S* (source) to a table column or host variable whose data type is fixed-length character string of length *D* (destination) or variable-length character string of maximum length *D*.

1. If the assignment is to a table column and *S* is larger than *D*, then the rightmost *S*–*D* characters of the source string must all be blanks ('**22**001') and only the leftmost *D* characters of the source string are assigned.
2. If the assignment is to a host variable and *S* is larger than *D*, the leftmost *D* characters of the character string are assigned ('**01**004').[21]
3. If *S* is smaller than *D*, the character string is assigned to the leftmost *S* character positions of the destination field. If the destination's data type is fixed-length character string, the remaining *D*–*S* positions are blank-filled.
4. If *S* is equal to *D*, the character string is assigned to the destination field.

Character-String Storage for C

For C, a character string of length *D* requires an array of *D*+1 elements. The elements are numbered 0 through *D*. The application must place a null byte (\0) in the position following the last one that contains character data, in order to ensure correct interpretation by the database ('**22**024'). When assigning a value to such a host variable, the database places a null byte in the *D*th position.

Numeric Assignments

When assigning a non-null numeric value to an **exact numeric** table column or host variable, there must be a representation of the numeric value in the data type of the destination field that does not cause the whole part of the number (that is, the leading significant digits) to be truncated ('**22**003'). The fractional part of the number (that is, the trailing significant digits) may be truncated as necessary and it is implementation-defined whether warning flags or indicator variables are set.

When assigning a non-null numeric value to an **approximate numeric** table column or host variable, the numeric value must be within the range of magnitude of the destination field ('**22**003'). The result is an approximation of the source numeric value that has the precision of the destination field.

21. Using the classification in Section 4.5.1 on page 90, truncation of non-blank data on assignment to the database is an error (and the SQL statement fails), whereas truncation on extracting data from the database is just a warning, whether or not the truncated data is blank.

    Indicator variables can detect truncation, as described in Section 3.8 on page 56.

Date/Time and Interval Assignment

When assigning a non-null date / time value to a column, the data type of the target location must also be date / time.

Its subtype (DATE, TIME or TIMESTAMP) must be the same as that of the source ('**22**007').

When assigning a non-null value to a **date/time** table column, the leading precision of the target must be sufficient to represent the value of the source ('**22**008').

When assigning a non-null value to an **interval** table column, the leading precision of the target must be sufficient to represent the value of the source ('**22**015').

Truncation of a fraction in a SECOND field is subject to the rules stated above for **Numeric Assignments**.

3.6 Comparison

Numeric

All numbers are comparable and are compared according to their algebraic value.

Character

The rules for comparing character strings depend on a specific collation. The collation that governs any string expression is derived from its component character strings (see Section 2.4.4 on page 23), or can be specified explicitly.

Before comparing two character strings, if the strings are of unequal length in characters and if the governing collation has a pad attribute of **pad space** (see **Pad Attribute** on page 23), space characters are conceptually appended to the shorter string to match the length of the longer string. If the governing collation has a pad attribute of **no pad**, no such padding occurs.

A string comparison starts at the start of the strings and ends on detecting either an inequality or the end of one of the strings (whichever is detected first). The governing collation defines the method and the result of comparing two strings.

The pad attribute is relevant only when comparing two strings of different lengths in characters that are identical through the entire length of the shorter string. Under a collation whose attribute is **no pad**, the longer string is greater than the shorter string in all such cases. Under **pad space**, the remaining characters of the longer string are compared to the spaces in the conceptual padding of the shorter string. The resulting sort order of the strings depends on the result of this comparison under the governing collation.

The rules of a given collation may dictate that two character strings compare equal even if they are of different length or contain different sequences of characters. If this is the case in the MAX and MIN functions, in references to a grouping column, or in the operations specified by the DISTINCT and UNION keywords, one of the equal values is chosen, using undefined criteria.

Date/Time

Two date/time values are comparable if their subtype (DATE, TIME or TIMESTAMP) is identical. When comparing two TIME or TIMESTAMP values, the implementation conceptually extends the seconds precision of the value with the lower precision by adding trailing zeros.

Interval

Two interval values are comparable only if they have some fields in common. Interval types with YEAR or MONTH fields are not comparable to interval types with DAY, HOUR, MINUTE or SECOND interval types. Interval types with different interval precision are conceptually converted to the same precision before comparison by adding fields as required.[22]

22. Added low-order fields have the value zero. Added high-order fields have the value zero unless the previous high-order field exceeds the limit imposed by the Gregorian calendar; in this case its value is normalised, resulting in a nonzero value in the new field. For example, to compare an INTERVAL MINUTE containing the valid value 75 to an INTERVAL HOUR TO MINUTE, the invalid value 0:75 is converted to the valid value 1:15.

3.7 Null Values

Every data type includes the null value, which is distinct from all non-null values. The null value represents a value that is not known or not applicable.

There is no *literal* that represents the null value. However, in some situations, SQL syntax uses the keyword NULL to represent the null value.

A table may constrain certain columns (using NOT NULL) to contain only non-null values. SQL inserts null values into tables in several situations when the application fails to specify or imply non-null values for insertion into the table. In these cases, the destination columns must not be constrained by NOT NULL.

The application and the database can communicate null values in embedded host variables by using an associated *indicator-variable* (see Section 3.8 on page 56).

Null Values in Computations

If the value of any operand in an *expression* is the null value then the value of the *expression* is the null value. (In particular, subtracting a null value from a null value evaluates to null and not to zero.)

If any operand in an *expression* is allowed to contain the null value, the result of that *expression* is conceptually allowed to contain the null value.

Assigning Null Values

In a database operation that would assign a null value to a table column, the column must be allowed to contain null values (that is, NOT NULL must not have been specified).

In a database operation that would assign a null value to a host variable, the application must provide an associated indicator variable (see Section 3.8 on page 56) ('**22**002'). The content of the host variable is undefined.

Comparing Null Values Explicitly

Generally, comparing a null value with any value, even another null value, evaluates to the unknown truth value. The only way to test for the presence or absence of null values is the IS [NOT] NULL predicate (see Section 3.10.6 on page 69).

Implicit Comparisons

Certain SQL comparisons and predicates imply comparison of values. The following table describes how these comparisons handle null values:

Context	**Treatment**
FOREIGN KEY	If any column of a FOREIGN KEY is null, the referential constraint is met.
GROUP BY	Null grouping column values are treated as equal.
ORDER BY	Null values are treated as equal and are either greater than or less than all non-null values. (The choice is implementation-defined.)
PRIMARY KEY	Null values are not allowed.
SELECT DISTINCT	Null values are treated as equal.
Set functions	Null values are eliminated regardless of whether DISTINCT is specified except that COUNT(*) includes all null values.
UNIQUE columns	Null values are treated as not equal, so multiple rows with null values in the same columns are allowed.
UNIQUE INDEX	Null values are treated as equal, so the table cannot have multiple rows with null values in the column.

Table 3-7 Treatment of Null Values in Various Contexts

3.8 Indicator Variables

A host variable may be accompanied by an indicator variable. The indicator variable contains information about a value that the host variable does not contain. (See Section 4.1.2 on page 80, for rules on the SQL syntax for indicator variables.)

Implementations use indicator variables in these cases:

- The database indicates that the associated host variable represents the null value by setting the required indicator variable to −1.
- The database sets the required indicator variable to a positive value if the associated host variable's data type is character string and the executable SQL statement truncates the value it assigns to the variable ('**01**004'). The positive indicator value is typically the total length of the original character string the database would have assigned to the host variable except for truncation.
- If neither of these cases occurs (if the database returns a non-null value that is not a truncated character string), the database sets the indicator variable to 0.
- Negative values other than −1 are reserved for future definition by X/Open.

Applications set any indicator variable to −1 to indicate a null value, and to 0 to indicate a non-null value, in the associated host variable.

3.9 Expressions

An *expression* represents a single value. It consists of one of the following:

- a *host-variable-reference*
- a *literal* (see Section 3.4 on page 48), including *pseudo-literal*s
- a *column-name*
- a *set-function-reference* (see Section 3.9.4 on page 63)
- a *dynamic-parameter* (see Section 3.9.5 on page 64)
- one of the scalar functions CHAR_LENGTH, CHARACTER_LENGTH, EXTRACT, LOWER, OCTET_LENGTH, POSITION, SUBSTRING, TRIM or UPPER (see Section 3.9.3 on page 59); or the CAST function (see Section 3.9.6 on page 65)
- any valid combination of these primary components connected by operators.

Any expression whose result is a character string has an implicit collation (see Section 2.4.4 on page 23), which indicates the manner of performing any comparison on the string. Any expression whose result is a character string can explicitly specify a collation by following the expression with the word COLLATE and then a *collation-name* (see Section 3.9.7 on page 66).

3.9.1 Arithmetic Operators

Arithmetic operators combine numeric expressions and yield a numeric value. The following arithmetic operators are allowed (in descending order of precedence):

+, –	unary plus and minus (negation)
*, /	multiplication and division
+, –	addition and subtraction

The operand following a unary plus or minus must not be a signed numeric literal.

Operations with the same precedence are executed from left to right. Parentheses may be used to depart from the above precedence order since, when used in an *expression*, they determine the binding of *expression* components to arithmetic operators.

Arithmetic errors occur in evaluating an expression (that does not involve date / time or interval values) on a violation of the following rules:

- No computed value, as a final or intermediate result, may exceed the range of the data type ('**22**003').
- Division by zero may not be performed ('**22**012').

3.9.2 Date/Time and Interval Arithmetic

The following table lists the only valid arithmetic operations involving values whose generic data type is date / time or interval:

Operand 1	**Operator**	**Operand 2**	**Result Type**
Date / Time	–	Date / Time	(see below)
Date / Time	+ or –	Interval	Date / Time
Interval	+	Date / Time	Date / Time
Interval	+ or –	Interval	Interval
Interval	* or /	Numeric	Interval
Numeric	*	Interval	Interval

Operands cannot be combined arithmetically unless their data types are comparable (as defined in Section 3.6 on page 53). If either operand is the null value, then the result of any arithmetic operation is the null value.

If arithmetic involves two date / time or interval values with a defined scale, then the scale of the result is the larger of the scales of the two operands.

Interval arithmetic that involves date ranges that span discontinuities in calendars[23] produces implementation-defined results.

Arithmetic errors occur in date / time and interval arithmetic, on a violation of the following rules:

- No computed date / time value, as a final or intermediate result, can exceed the leading precision of the data type ('**22**008').
- No computed interval value, as a final or intermediate result, can exceed the leading precision of the data type ('**22**015').
- Division by zero may not be performed ('**22**012').

Subtracting Two Date/Times

One date / time value may be subtracted from another to produce an interval that is the signed difference between the stated dates or times. However, the application program must specify a named interval data type for the result by using an *interval-qualifier*. The required syntax is:

```
(date-time-1 – date-time-2) interval-qualifier
```

EXTRACT Function

The EXTRACT function extracts a single numeric field from a date / time or interval value. It has the form:

```
EXTRACT (field-name FROM expression)
```

where *field-name* is one of YEAR, MONTH, DAY, HOUR, MINUTE or SECOND. The *expression* must be of type date / time or interval and must contain the field specified by *field-name*.

The data type of the result is exact numeric. The precision and scale are implementation-defined except that, for *field-name*s other than SECOND, the scale is 0.

23. Discontinuities are decisions to add units to, or remove units from, a calendar at a specified point in time. The International Standard lists leap seconds (see Section 3.2.3 on page 42) as an example of discontinuity. Another example is the addition of 10 days to the Gregorian calendar in the 18th century. The acceptance and timing of a discontinuity depends on the locale.

If *expression* is null, then the result is the null value. Extracting any field from a negative interval produces a negative value. All other uses of EXTRACT produce a non-negative value.

3.9.3 String Operations

The following table lists the valid operations on character strings. In the table, *s1* and *s2* stand for character-string operands and *i* and *j* stand for *arithmetic-expression*s. Additional valid syntax and details of the respective operations follow the table.

Simplified Syntax	**Result Returned**
s1 | | *s2*	A character string formed by concatenating *s1* and *s2*.
CHAR_LENGTH(*s1*)	An exact numeric indicating the length of *s1*.
CHARACTER_LENGTH(*s1*)	Same as CHAR_LENGTH.
CONVERT(*s* USING *c*)	A string resulting from applying conversion *c* to string *s*.
LOWER(*s*)	A copy of *s* with any upper-case letters converted to lower case.
OCTET_LENGTH(*s1*)	Same as CHAR_LENGTH.
POSITION(*s1* IN *s2*)	An exact numeric indicating the character position of the first occurrence of *s1* in *s2*.
SUBSTRING(*s1* FROM *i* [FOR *j*])	A string extracted from *s1*, starting at character position *i* and continuing for *j* characters.
TRANSLATE(*s* USING *t*)	A string resulting from applying translation *t* to string *s*.
TRIM(*s1*)	A string consisting of *s1* with leading and/or trailing pad characters removed.
UPPER(*s*)	A copy of *s* with any lower-case letters converted to upper case.

Table 3-8 String Operations

Long Strings

An exact numeric return value is specified above for CHAR_LENGTH, CHARACTER_LENGTH, OCTET_LENGTH and POSITION. Applications can use integer values to retrieve these return values, and elsewhere to represent the length of, or position within, a string. However, if it is possible that the string length could exceed the capacity of an integer variable:

- COBOL applications should use DECIMAL(*p*, 0), where *p* is large enough for the data of interest.
- C applications should do one of the following:
  — use **long** and be prepared to handle overflow exceptions
  — use **double** and be prepared for the result to be inexact without notification.

Concatenation

The result of the concatenation operation is the value of the string operand preceding the || operator, followed by the value of the string operand following the || operator.

If the data type of both operands is a fixed-length string, then the data type of the result is a fixed-length string. If the data type of either operand is a variable-length string, then the data type of the result is a variable-length string. If the value of either operand is null, the result is the null value.

The length of the result is the sum of the lengths of the operands except when this is excessive:

- If the data type of both operands is a fixed-length string and if the length of the resulting string exceeds the implementation-defined maximum for fixed-length strings, the operation fails ('**42**000').
- If the data type of either operand is a variable-length string and if the length of the resulting string exceeds the implementation-defined maximum for variable-length strings, then:
  - — If all the characters in the positions beyond the maximum length are the space character, the result is truncated on the right to the maximum length (without warning).
  - — Otherwise, the operation fails ('**22**001').

Capitalisation Functions

The LOWER function returns a copy of its string argument, in which any upper-case letter is replaced with the corresponding lower-case equivalent. The UPPER function returns a copy of its string argument, in which any lower-case letter is replaced with the corresponding upper-case equivalent.

Upper-case letters, lower-case letters, and the correspondence referred to above are defined by the character set associated with the string argument. The string result is in the same character set. The expressions UPPER(LOWER(*s*)) and LOWER(UPPER(*s*)) do not necessarily yield *s* in all cases.

CHAR_LENGTH

The CHAR_LENGTH function returns the length of the value (the number of characters) of its single character-string operand.

If the value of the operand is null, then the function returns the null value.

The keyword CHARACTER_LENGTH can be used instead of CHAR_LENGTH. For single-byte character sets, the keyword OCTET_LENGTH can be used instead of CHAR_LENGTH.

CONVERT

The CONVERT function returns a character string that is the value of the string operand with a specified conversion applied. Converting a string changes it to use a different form-of-use (see Section 2.4.3 on page 21).

The names and effects of any available conversions are implementation-defined.

POSITION

The POSITION function returns an integer indicating the first position at which the value of one character string occurs within another. The first position in a character string is position number 1. POSITION returns the value 0 in several cases specified below where there is no such occurrence. The function has the form:

```
POSITION (search-string IN source-string)
```

where *search-string* and *source-string* are both *string-expression*s.

The function performs the following tests in sequence:

- If the value of either operand is null, the function returns the null value.
- If *source-string* has a length of zero, the function returns zero.
- If the characters of *search-string* occur consecutively within *source-string*, then the function returns the character position of the start of the first such occurrence.
- Otherwise (if *search-string* does not occur within *source-string*), the function returns zero.

All comparisons of characters within the two operands are case-insensitive.

SUBSTRING

The substring function returns a character string which is a portion (zero-length, partial, or complete) of its character-string operand. The function has the form:

```
SUBSTRING (source-string FROM start-position [FOR string-length])
```

where *source-string* is a *string-expression* and *start-position* and *string-length* are both *arithmetic-expression*s.

The data type of the result is variable-length character string, with the same maximum length as *source-string* (or as the fixed length of *source-string*, if it is a fixed-length string). The data type of both numeric operands is exact numeric with a scale of zero.

If *string-length* is omitted, then its value is assumed to be:

```
CHAR_LENGTH(source-string) + 1 − start-position
```

This denotes the entire remainder of *source-string* at and beyond *start-position*.

The SUBSTRING function performs the following steps in sequence:

- If the value of any operand is null, the function returns the null value.
- If *string-length* is negative, or if *start-position* is greater than the number of characters in *source-string*, the function fails ('**22**011').
- Otherwise, the function returns a character string containing *string-length* characters of *source-string*, starting at the character specified by *start-position* and in the same sequence that the characters appear in *source-string*. (To the extent that some of these positions are before the start or beyond the end of *source-string*, then no character is returned. If all the characters specified are thus, then the result is the null string.)

TRANSLATE

The TRANSLATE function returns a character string that is the value of the string operand with a specified translation applied. A translation might be defined to simply translate text from one character set to another. Other translations might perform more complex translations. The LOWER and UPPER string functions might be implemented as translations.

The names and effects of any available translations are implementation-defined. An application can query the TRANSLATIONS system view (see *TRANSLATIONS* on page 172) to determine what translations are accessible. An application can execute the CREATE TRANSLATION and DELETE TRANSLATION statements to control user-defined translations.

Each translation has a source character set and a target character set. These are implementation-defined and available through the TRANSLATIONS system view. The string operand must be in the source character set.

The TRANSLATE function must not apply a translation to a string operand so as to produce a character that is not in the repertoire of the character set defined as the target character set of the translation ('**22**021').

TRIM

The TRIM function returns a character string that is the value of a string operand with certain pad characters removed. The complete syntax of TRIM is as follows:

```
TRIM ([[LEADING | TRAILING | BOTH] [trim-character] FROM]
        source-string)
```

where *trim-character* and *source-string* are both *string-expression*s.

The brackets above indicate optional syntax. However, if the word FROM is specified, then some valid syntax must appear between the opening parenthesis and the word FROM. If neither LEADING, TRAILING nor BOTH is specified, then BOTH is assumed.

The value of *trim-character* is the trim character: the single character that is to be removed from the start and/or end of *source-string*. If *trim-character* is omitted, then the space character is the trim character.

The data type of the result is variable-length character string, with the same maximum length as *source-string* (or as the fixed length of *source-string*, if it is a fixed-length string).

The TRIM function performs the following steps in sequence:

- If the value of any operand is null, the function returns the null value.
- If the length of the value of *trim-character* is not 1, the operation fails ('**22**027').
- Otherwise, the function returns the value of *source-string* after trimming characters from it as follows:
  - — If LEADING is specified: Any consecutive occurrences of the trim character at the start of *source-string* are removed.
  - — If TRAILING is specified: Any consecutive occurrences of the trim character at the end of *source-string* are removed.
  - — If BOTH is specified or assumed: Both leading and trailing occurrences of the trim character are removed.

3.9.4 Set Functions

A set function can appear in an *expression* in two contexts within a *query-specification* (see Section 3.11.1 on page 71):

- Within a *select-sublist*, the set function analyses the entire virtual table.
- Within the *search-condition* in a HAVING clause, the *query-specification* specifies a grouped table and the set function analyses the current group.

The set function returns a single value as a result.

A *set-function-reference* requires an argument in parentheses. The rules for a valid argument depend on the form used and are discussed below. The general format of a *set-function-reference* is:

```
set-function-name ([ALL | DISTINCT] argument)
```

For the COUNT function, X/Open SQL supports these two forms:

```
COUNT(*)
COUNT(DISTINCT argument)
```

The *set-function-name* may be any of those shown in the following table:

Function	**Result**	**Data Type of Result**
AVG	Average of the values in the column.	Same generic data type as the argument; implementation-defined attributes.
COUNT	Total number of values in (cardinality of) the column.*	Exact numeric, scale 0, implementation-defined precision and range.
MAX	Largest value in the column.	Same as the argument.
MIN	Smallest value in the column.	Same as the argument.
SUM	Sum of the values in the column.	**Exact numeric argument:** Exact numeric result; same scale as argument; implementation-defined precision and range. **Approximate numeric argument:** Approximate numeric result; implementation-defined precision and range of magnitude.

Table 3-9 Set Functions

Functions AVG and SUM can be applied only to numeric columns.

If DISTINCT is specified, the set function calculates the result after excluding duplicate values of the *argument*. In this case, *argument* must be a simple *column-name* and the set function cannot be combined with other terms using binary arithmetic operators. The keyword DISTINCT cannot be used in a query or sub-query which itself uses the form SELECT DISTINCT; see Section 3.11 on page 71.

* COUNT(*) does not refer to columns but returns the cardinality of the entire virtual table (or, if it occurs in a HAVING clause, the cardinality of the current group).

If ALL is specified (or in the case of COUNT(*)), duplicate values are retained. In this case, the *argument* may be an *expression* but it must contain at least one *column-name* and may not contain any *set-function-references*.

If neither ALL nor DISTINCT is specified, ALL is assumed.

In both cases, any null *argument* values are eliminated before the function is applied. (If any null values are eliminated, SQLSTATE is set to ('**01**003').) However, COUNT(*) counts all values, even duplicate and null values.

A *set-function-reference* may be specified only in an expression in a SELECT clause or in an *expression* that is contained within the *search-condition* of a HAVING clause. If it appears in a *sub-query* of a HAVING clause, its argument must be a correlated reference.

An *expression* directly contained within the *search-condition* of a WHERE clause must not reference a column derived from a set function.

A *column-name* specified in the *argument* of a *set-function-reference* must not reference a column derived from a set function. If such a *column-name* is a correlated reference (see Section 3.11.5 on page 76), then all of the following must be true:

- The argument must consist solely of that *column-name*.
- The *set-function-reference* must be contained in a *sub-query* of a HAVING clause.
- The correlated reference must be to a *table-reference* contained in the FROM clause of the outer query that contains the HAVING clause.

Set Function Exceptions

The sum of the column values on which a call to AVG or SUM operates must be within the range of the data type of the result ('**22**003').

All set functions return the null value if applied to an empty set, except that COUNT(*) returns the value 0.

3.9.5 Dynamic Parameters

A dynamic parameter is represented by a question mark (?) and identifies a parameter in a dynamically prepared SQL statement.

If the context in which the dynamic parameter is used requires an interval, then the question mark must be followed by a suitable *interval-qualifier* to specify the interval precision. There are two exceptions, in which no *interval-qualifier* is required:

- a quantity added to, or subtracted from, a DATE is assumed to be of type INTERVAL YEAR(*p*) TO MONTH
- a quantity added to, or subtracted from, a TIME or TIMESTAMP is assumed to be of type INTERVAL YEAR(*p*) TO SECOND(*s*)

where *p* and *s* above are the implementation-defined maxima for the respective precision.

3.9.6 CAST Function

The CAST function explicitly converts data of one data type to another data type. The CAST function has the form:

```
CAST({expression | NULL}) AS data-type)
```

where *data-type* is one of the named data types specified in Section 3.2 on page 39.

The resulting data type of the CAST function is *data-type*. Applying the CAST function to NULL, or to an expression that has the null value, yields the null value of the target data type.

Casting an expression from a source data type (specified by the data type of *expression*) to a target data type (specified as *data-type*) is subject to the rules set out below. Conversions not specifically permitted below are invalid.

Source and Target Data Types Identical

Use of the CAST function is essentially an assignment, subject to the rules in Section 3.5 on page 51.

Character Source

The value of *expression* must conform to the natural limits imposed on intervals ('**22**006') and on date / time values ('**22**007') by the Gregorian calendar (see Section 3.2.3 on page 42); and must in other respects be a valid literal representation of a value of the target data type ('**22**018').

Character Target

There must exist a representation of a literal with the value of *expression* in the character set defined by the implementation ('**22**018').

For conversions to the character class, the length of the converted value must not exceed the length of the target (for CHARACTER) or the maximum string length (for CHARACTER VARYING) ('**22**001').

For conversions to the CHARACTER data type, the value of the source expression is padded on the right with spaces if the length of the converted value is less than the length of the target data type. For conversions to the CHARACTER VARYING data type, this blank padding does not take place.

Numeric Source

If the source data type is exact numeric, any conversion to an interval data type must not result in the loss of leading significant digits ('**22**015').

Date/Time Source

The CAST function can be used to cast an expression of one date / time data type to another with the following effects:

- When casting a DATE to a TIMESTAMP, the HOUR, MINUTE and SECOND fields of the target are set to zero. Other fields are set to the corresponding values in the source expression.
- When casting a TIME to a TIMESTAMP, the YEAR, MONTH and DAY fields of the target are set to the respective values obtained by evaluating CURRENT_DATE. Other fields are set to the corresponding values in the source expression.

- When casting a TIMESTAMP to a DATE or TIME, the fields of the target are set to the corresponding values in the source expression. Some fields of the source expression are not present in the target.

Interval Source

The value of the source expression must be representable as an exact numeric value without the loss of leading significant digits ('**22**003').

3.9.7 Specifying a Collation

Section 2.4.4 on page 23 lists the rules that determine the collation of a character string. The rule with the highest precedence is that the collation can be specified explicitly in the embedded SQL program.

To specify a collation explicitly, the program element is followed by the word COLLATE and then the name of a collation:

```
element COLLATE collation-name
```

where *element* is one of the following:

- A reference to a column of a table whose data type is character string.
- A *character-string-literal* defined in Section 3.4 on page 48, and the pseudo-literals defined in **Authorisation Pseudo-literals** on page 50).
- Any of the expressions defined in Section 3.9 on page 57 whose result is a character string:
  — the CAST, CONVERT, LOWER, SUBSTRING, TRANSLATE, TRIM and UPPER functions
  — any of the set functions defined in Section 3.9.4 on page 63, when used in a context such that its result is a character string
  — a dynamic parameter (see Section 3.9.5 on page 64) whose data type is character string.
- A *sub-query* (see Section 3.11.4 on page 76) whose data type is character string. (The COLLATE syntax appears outside the parentheses of the *sub-query*.)
- A host variable (see Section 4.1.2 on page 80) whose data type is character string.

3.10 Search Conditions and Predicates

A **predicate** is an assertion about a relationship between values. The valid SQL predicates are described in the remainder of this section. These predicates specify or imply comparisons between two values. In any such comparison, the data types of these values must be comparable (as defined in Section 3.6 on page 53). A **search condition** is a combination of one or more predicates using the **logical operators** AND, OR and NOT.

Search conditions, predicates and logical operators in SQL use tri-state logic, dealing with the truth values true, false and unknown. The unknown truth value often results from comparisons with a null value.

Uses of Search Conditions

In query specifications and in sub-queries (see Section 3.11 on page 71), the WHERE clause uses a *search-condition* to qualify its selection of rows. The HAVING clause uses a *search-condition* to qualify its selection of groups. The CHECK clause in the CREATE TABLE statement uses a *search-condition* to restrict the values that rows of that table may contain. The predicates in a *search-condition* are evaluated once for each row or group selected. Qualifying rows or groups in a SELECT are those for which the *search-condition* evaluates to true.

A *column-name* or *expression* specified in a *search-condition* is directly contained in that *search-condition* if the *column-name* or *expression* is not specified within a *set-function-reference* or a *sub-query* of that *search-condition*.

Logical Operators

The logical operators AND, OR and NOT operate on the truth values true (T), false (F) and unknown (U), as follows:

AND	T	F	U
T	T	F	U
F	F	F	F
U	U	F	U

OR	T	F	U
T	T	T	T
F	T	F	U
U	T	U	U

NOT	
T	F
F	T
U	U

The order of precedence among the logical operators is NOT (highest), followed by AND, followed by OR. The order of evaluation at the same precedence level is from left to right. The application may use parentheses to change this order since, when used in a *search-condition*, they determine the binding of predicates to logical operators.

3.10.1 Comparison Predicate

A comparison predicate compares two values and has the form:

```
expression-1 comparison-operator {expression-2 | (sub-query)}
```

where *comparison-operator* may be any of the following:

=	equal to
<>	not equal to
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to

Within the context of a comparison predicate, a *sub-query* must result in either a single value or an empty set ('**21**000'). If the result of the *sub-query* is an empty set, the result of the predicate is unknown. If the result of the *sub-query* is a single value, the same rules apply as for *expression-2*.

If either *expression-1* or *expression-2* evaluates to the null value, the result of the predicate is unknown. Otherwise, the result is true or false, depending on the outcome of the comparison.

3.10.2 Quantified Comparison Predicate

A quantified comparison predicate compares a value with a number of derived values and has the form:

```
expression comparison-operator {ALL | ANY | SOME} (sub-query)
```

where *comparison-operator* is as described in Section 3.10.1 on page 67.

If ALL is Specified

The result of the predicate is true if the *sub-query* results in an empty set or if the comparison is true for every value returned by the *sub-query*. The result of the predicate is false if the comparison is false for at least one value returned by the *sub-query*. Otherwise, the result of the predicate is unknown.

If SOME or ANY is Specified

The result of the predicate is true if the comparison is true for at least one value returned by the *sub-query*. The result of the predicate is false if the *sub-query* results in an empty set or if the comparison is false for every value returned by the *sub-query*. Otherwise, the result of the predicate is unknown.

3.10.3 BETWEEN Predicate

A BETWEEN predicate tests whether a value is within a range of values and has the form:

```
expression-1 [NOT] BETWEEN expression-2 AND expression-3
```

The predicate (without NOT) is equivalent to:

```
expression-1 >= expression-2
      AND expression-1 <= expression-3
```

Using the keyword NOT negates the result in the manner of the NOT logical operator.

3.10.4 IN Predicate

An IN predicate compares a value with a list of values or with a number of derived values and has the form:

```
expression [NOT] IN {(value [, value]...) | (sub-query)}
```

where *value* is a *dynamic-parameter*, a *literal* or a *host-variable-reference*.[24]

The operator IN is equivalent to the operator = ANY. Thus, when NOT is not specified, the result of the predicate is true if the implied equality comparison is true for at least one specified or derived comparison value; false if the implied equality comparison is false for every specified or derived comparison value or if the *sub-query* results in an empty set; and unknown otherwise. Using the keyword NOT negates the result in the manner of the NOT logical operator.

24. Any character-string *pseudo-literal* (for example, USER) is also valid here, but not useful.

3.10.5 LIKE Predicate

A LIKE predicate compares a column value with a pattern and has the form:

`column-name [NOT] LIKE pattern-value [ESCAPE escape-character]`

where *pattern-value* and *escape-character* are each a *dynamic-parameter*, a *character-string-literal*, a *host-variable-reference*, or the pseudo-literals defined in **Authorisation Pseudo-literals** on page 50.

The column *column-name* must reference a character string column. If *host-variable-reference* is specified, it must reference a character string variable.

If *escape-character*, *pattern-value*, or the value of the column referenced by *column-name* is the null value, the result of the predicate is unknown. If NOT is not specified, the result of the predicate is true or false depending on whether or not the value of the column referenced by *column-name* conforms to the specified pattern. Using the keyword NOT negates the result in the manner of the NOT logical operator.

Pattern Syntax

Within the character string represented by *pattern-value*, characters are interpreted as follows:

The underscore character (_) stands for any single character.

The percent character (%) stands for any sequence of zero or more characters.

All other characters stand for themselves.

For example, `LIKE '%A%'` is true for any column value that contains the character A. `LIKE 'B__'` is true for any column value that is three characters long and starts with the character B.

Escape Character

The ESCAPE clause introduces a single character as the escape character for *pattern-value*. The escape character is not treated as part of the pattern, and causes the following character in *pattern-value* to be taken literally (without the special effects for _ and % described above). For example, `LIKE 'e%%' ESCAPE 'e'` is true for any column value that begins with a percent sign.

To include the *escape-character* literally within *pattern-value*, it must appear twice in succession.

DIAGNOSTICS

The length of *escape-character* must be exactly 1 character ('**22**019').

The character following an occurrence of *escape-character* in *pattern-value* must be a character whose effect can be defeated by *escape-character*; namely, the underscore character, the percent character, or another *escape-character* ('**22**025').

3.10.6 NULL Predicate

A NULL predicate compares a value against the null value and has the form:

`column-name IS [NOT] NULL`

The result of the IS NULL predicate is true if the value of the column referenced by *column-name* is the null value, and false otherwise. Using the keyword NOT negates the result in the manner of the NOT logical operator.

This predicate is the only way to test for the presence or absence of null values. An application cannot identify null values by comparing them with an embedded host variable whose *indicator-variable* identifies it as a null value because the comparison always returns the unknown truth value.

3.10.7 EXISTS Predicate

An EXISTS predicate tests for the existence of a row satisfying some condition and has the form:

```
EXISTS(sub-query)
```

The result of the predicate is true if the *sub-query* does not result in an empty set, otherwise the result of the predicate is false.

3.10.8 OVERLAPS Predicate

An OVERLAPS predicate tests two ranges of dates and times to see if the ranges overlap. It has the form:

```
row-value-constructor-1 OVERLAPS row-value-constructor-2
```

Row value constructors are defined in Section 3.12 on page 78.

The degree of both *row-value-constructor*s must be two. Therefore, another way to illustrate the OVERLAPS predicate is:

```
(a1, b1) OVERLAPS (a2, b2)
```

The values *a1* and *a2* must be date/time values whose data type is comparable (as defined in Section 3.6 on page 53). Each parenthesised pair of values defines a range of dates or times. The values *a* and *b* in a pair must relate to one another in one of the following ways:

- If *a* and *b* are date/time values whose data type is comparable, then the pair defines the range of dates or times between *a* and *b*, (either of which may be the earlier).
- If *a* is a date/time value and *b* is an interval value that could be added to it (as defined in Section 3.9.2 on page 57), then the pair defines the range between *a* and $a + b$. (This sum is earlier or later than *a* depending on the sign of *b*.)

The result of the OVERLAPS predicate is true if the two ranges have any date or time in common, including either endpoint of the ranges. If the two ranges do not overlap, the result is false. If any of the values is null, then the result is undefined.

3.11 Queries

A *query-specification* is the basic data retrieval construct. Evaluating a *query-specification* may involve conceptually joining several specified tables and may involve arranging and extracting information from the table(s) based on specified criteria. (See Section 3.11.1.)

A *query-specification* is used in the INSERT statement. It is also a component of a *query-expression*. A *query-expression* is an expression involving either one *query-specification* or several combined with the UNION operator. (See Section 3.11.3 on page 75.) The *query-expression* syntactic element appears in the DECLARE CURSOR and CREATE VIEW statements.

Queries can occur in other contexts, such as predicates. Such a query is a *sub-query* (see Section 3.11.4 on page 76).

3.11.1 Query Specifications

FUNCTION

Specify a derived table.

SYNOPSIS

A *query-specification* has the form:

```
SELECT [ALL | DISTINCT] select-list
      FROM table-reference [, table-reference]...
      [WHERE search-condition]
      [GROUP BY column-clause [, column-clause]...]
      [HAVING search-condition]
```

where *column-clause* is defined as:

```
column-name [COLLATE collation-name]
```

and where *select-list* is defined as:

```
* | select-sublist [, select-sublist]...
```

and *select-sublist* is defined as:

```
expression [[AS] unqualified-column-name]
      | {table-name | correlation-name}.*
```

and *table-reference* is defined as:

```
table-name [[AS] correlation-name]
      | joined-table
```

and *joined-table* is as defined in Section 3.11.2 on page 73.

DESCRIPTION

The result of evaluating a *query-specification* can be explained in terms of a multi-step algorithm. The order of steps in this algorithm follows the mandatory order of the clauses (FROM, WHERE, and so on) of the SELECT statement:

Step 1 For each *table-reference* that is a *joined-table*, conceptually join the tables as specified to form a single table, as described in Section 3.11.2 on page 73.

Step 2 Form a Cartesian product of all the *table-reference*s, so that each row of the result is a concatenation of one row from each of the tables in the order specified, and there is one row in the result for each combination of rows of the referenced tables.

Step 3 Eliminate all rows that do not satisfy the *search-condition* in the WHERE clause.

Step 4 Arrange the resulting rows into groups.

— If there is no GROUP BY clause, there is a single group and there are no grouping columns.

— If there is a GROUP BY clause specifying grouping columns, then form groups so that all rows within each group have equal values for the grouping columns according to the specified or implicit collation.

Collation is specified by use of the COLLATE syntax. If this is not used, the groups are formed according to the collation that the table specifies for the grouping columns. The COLLATE syntax can be used only on grouping columns whose data type is character string.

— If a grouping column contains any null values, these values are considered equal and give rise to only a single group.

Step 5 If there is a HAVING clause, eliminate all groups that do not satisfy its *search-condition*. The *expressions* in the HAVING clause must be single-valued per group. Thus each *column-name* directly contained within the *search-condition* must reference a grouping column or be a correlated reference (see Section 3.11.5 on page 76), and if a *sub-query* contained in the *search-condition* contains a correlated reference, it must be to a grouping column or must be specified as the argument of a set function reference.

Step 6 Generate result rows based on the result columns specified by the *select-list*.

The *select-list* ''*'' is equivalent to a sequence of *expressions* in which each *expression* is a *column-name* that references a column resulting from Step 2. The columns are referenced once each, in the order of their ordinal position at the end of Step 2.

If the *select-sublist* ''{*table-name* | *correlation-name*}.*'' is specified, *table-name* or *correlation-name* must denote a table referenced by the FROM clause. The form *select-sublist* ''{*table-name* | *correlation-name*}.*'' is equivalent to a sequence of *expressions* in which each *expression* is a *column-name* that references a column of the table denoted by *table-name* or *correlation-name*. The columns are referenced once each, in the order of their ordinal position in the denoted table.

The above uses of the * character may produce different results on different X/Open SQL implementations; see Section 7.8 on page 185.

Each implicit or explicit *expression* in the *select-list* represents a result column.

If GROUP BY was specified, each group generates one result row and the result columns must be single-valued per group. Therefore, any column referenced by the *select-list* must either be a grouping column or be referenced within the *argument* of a *set-function-reference*. If there are no groups, the result of the *query-specification* is an empty set.

All result columns have names, derived as follows:

- If an AS clause (a *select-sublist*) is specified for the column, then its *unqualified-column-name* specifies the name of the result column.
- Otherwise, if *expression* for the column consists only of a *column-name*, then the *column-name* specifies the name of the result column.
- Otherwise, the name of the result column is undefined, except that it is guaranteed to be different from the name of any column, other than itself, of a table referenced by a *table-reference* in the SQL statement.

If GROUP BY was not specified, each row generates one result row unless a result column is derived from a set function, in which case the *query-specification* results in only a single row and any column referenced by the *select-list* must be referenced within the *argument* of a *set-function-reference*.

Step 7 In the case of SELECT DISTINCT, eliminate duplicate rows from the result.

The form SELECT DISTINCT cannot use the keyword DISTINCT again in set functions within the *query-specification*. For example, if you use SELECT DISTINCT to eliminate duplicate rows from the result, you cannot also use DISTINCT to eliminate duplicate rows from the argument of a set function. However, the keyword DISTINCT may reappear in a *sub-query* (see Section 3.11.4 on page 76).

Updatability

A *query-specification* is updatable if and only if the following conditions are satisfied:

- The FROM clause contains a single *table-reference*, either to a base table or to a derived table that is updatable.
- Neither the GROUP BY clause nor the HAVING clause is present.
- DISTINCT is not specified.
- All the result columns are derived from *column-names* and no *column-name* appears more than once. That is, either the *select-list* contains no explicit *expressions* or every explicit *expression* it contains consists of a single distinct *column-name*.
- If there is a WHERE clause and it contains a *sub-query*, then the *sub-query* must not reference any table referenced in the FROM clause (including indirect references).

3.11.2 Joined Tables

The JOIN syntax provides methods of combining information in tables. Varieties of combinations, defined below, are the natural join, the inner join, the left outer join, and the right outer join. (Specifying two or more tables in the FROM clause of a query specification also conceptually joins the tables, in the same manner as an inner join.)

FUNCTION

Specify combinations of tables within a *query-specification*.

SYNOPSIS

A *joined-table* is defined as:

```
qualified-join | (joined-table)
```

and *qualified-join* is defined as:

```
table-reference-1
   [NATURAL]
   [INNER | LEFT [OUTER] | RIGHT [OUTER]]
   JOIN table-reference-2
   {ON search-condition | USING (column-name[, column-name ...])}
```

DESCRIPTION

Every use of *qualified-join* (that is, every case of combining two tables using the JOIN operator) must specify exactly one of NATURAL, ON, and USING to indicate one of the following operations:

NATURAL — NATURAL JOIN produces a non-null table when the two tables being joined have common columns (columns with the same name). The operation is meaningful when common columns contain identical information in the case where the respective rows are related. For example, two tables may contain different sets of information on persons, but a common column called NAME indicates in both tables the person to which the information in that row pertains. In this example, if the NAME column contains the same value in rows of two tables, it means those two rows are related.

The result set of a NATURAL JOIN contains one row for each case where all common columns in the two tables are equal. The result set contains all the columns from both tables, except that the matching common columns appear only once in the result set.

Performing a NATURAL JOIN on the two tables of personal information discussed in the example above produces a result set with one composite row for each person, based on the value of the common NAME column.

Every common column in the two tables must have a data type that permits the values in the two tables to be compared.

USING — USING introduces a list of *column-name*s. The operation is conceptually the same as NATURAL JOIN, but JOIN USING takes the specified columns, instead of all common columns, as the basis for joining the rows.

Specifying the columns explicitly instead of using the entire set of common columns is useful when both tables contain common columns (for example, a REMARKS column) which might not be identical even though the rows are related. Specifying the columns explicitly also guards against undesired rejection of rows in cases where columns have identical names by accident.

Every specified column in the two tables must have a data type that permits the values in the two tables to be compared.

ON — ON introduces a *search-condition*. From a Cartesian product of the two tables (defined in Section 3.11.1 on page 71), the result set contains only those rows for which the *search-condition* is true. The *search-condition* cannot reference common columns unless it qualifies them, such as by table name.

OUTER JOIN

The above discussion describes the effect if INNER is specified, or if neither INNER, LEFT OUTER, or RIGHT OUTER is specified. (The keyword OUTER is always optional.) The INNER JOIN includes rows in the result set based on the respective criteria listed above. LEFT OUTER JOIN includes in the result table additional rows from the table specified to the left of the word JOIN (that is, *table-reference-1*). RIGHT OUTER JOIN includes additional rows from the table to the right (*table-reference-2*):

- In LEFT OUTER JOIN, rows from *table-reference-1* that were not the basis for any row of the INNER JOIN appear at the end of the result set. All information from *table-reference-1* thus is present in at least one row of the result set.
- In RIGHT OUTER JOIN, rows from *table-reference-2* that were not the basis for any row of the OUTER JOIN appear at the end of the result set. All information from *table-reference-2* thus is present in at least one row of the result set.

Both OUTER JOINs add rows to the result set based on a row of only one of the joined tables. In each row, the columns that are defined only in the other table contain the null value. These rows are combined with the result of the INNER JOIN as in the UNION ALL operator discussed in Section 3.11.3.

3.11.3 Query Expressions

A *query-expression* is one or a combination of *query-specifications* defined as follows:

```
query-expression UNION [ALL] query-expression
      | (query-expression)
      | query-specification
```

Parentheses control the order of evaluation of any UNION operators.

UNION of Tables

The UNION or UNION ALL operator forms a union of two derived tables specified as the operands. The union is a derived table that contains all the rows of the two operands. In the UNION ALL form, duplicate rows are retained; if ALL does not follow UNION, then duplicate rows are eliminated from the union. The two operands must have the same number of columns.

Corresponding columns in a UNION or UNION ALL operation must have compatible data types. The following list shows all valid combinations of operand columns and, for each combination, shows the data type of the resulting column:

- If the data type of one operand is a character string, then the data type of the other operand must be character string.

  If either of the operand data types is variable-length character string, then the result data type is variable-length character string whose maximum length is the largest maximum length of either operand.

  Otherwise, the result data type is fixed-length character string whose length is the largest length of either of the operands.

- If both of the operand data types are exact numeric, then the result data type is exact numeric, with implementation-defined precision, and whose scale is the largest scale of either of the operands.

- If the data type of one operand is approximate numeric, then the data type of the other operand must be numeric. The result data type is approximate numeric with implementation-defined precision.

- If the data type of one operand is date/time, then the data type of the other operand must be date/time and the subtypes must be identical. The result data type is date/time with this same subtype.

- If the data type of one operand is interval, then the data type of the other operand must also be interval. Either both operands' interval precisions must specify YEAR or MONTH, or both operands' interval precisions must specify DAY, HOUR, MINUTE or SECOND. The result data type is interval and its interval precision is selected to accommodate the date/time fields of either operand. For example, the union of INTERVAL DAY with INTERVAL MINUTE TO SECOND produces a result of data type INTERVAL DAY TO SECOND.

The result column allows null values if and only if either operand column allows them.

When these corresponding columns have the same name, the result column takes that name. Otherwise, the name of the result columns is undefined, except that it is guaranteed to be different from the name of any column, other than itself, of a table referenced by a *table-reference* in the SQL statement.

UNION may be specified several times; the application can use parentheses to indicate the order in which the union operations are performed. If two or more UNION operations are specified without parentheses, the operations are performed from left to right. The result set of the cursor is the result of the final UNION.

Updatability

A *query-expression* that contains the keyword UNION is not updatable. A *query-expression* that is a single *query-specification* is updatable depending on whether the *query-specification* is updatable. This is defined in Section 3.11.1 on page 71.

3.11.4 Sub-queries

FUNCTION

Provide a multi-set of values or rows within a predicate.

SYNOPSIS

A *sub-query* has the same syntax as a *query-specification* (see Section 3.11.1 on page 71) but is enclosed in parentheses.

DESCRIPTION

A *sub-query* is a limited form of *query-specification*. A *sub-query* can be used only on the right hand side of a comparison predicate, quantified predicate, or IN predicate, or as the subject of an EXISTS predicate. A *sub-query* can be nested in *expression*s within another *sub-query*. The result of the *sub-query* is substituted into the predicate of the outer query.

A *sub-query* used in a context where each result row is compared to a scalar value (in the quantified comparison and IN predicates) must have a single result column. A *sub-query* used in a comparison predicate must have a single result row and column.

A *sub-query* in an EXISTS predicate is used to derive a multi-set of rows, but it is the existence of one or more result rows, rather than the values in these rows, that is significant; therefore, in this case, the result columns can only be specified using a single asterisk (the normal case), a single asterisk qualified by a *table-name* or *correlation-name*, or a single *column-name*.

A *sub-query* is subject to the same restriction as for *query-specification* regarding multiple use of the keyword DISTINCT.

DIAGNOSTICS

When a *sub-query* is used in a position that requires at most a single value, its evaluation must produce a result of no more than one row ('**21**000').

3.11.5 Correlation

The *column-name* element has been defined as:

```
[{table-name | correlation-name}.]unqualified-column-name
```

A qualifier is added to a *unqualified-column-name* to show which table, and which reference to that table, the *column-name* relates to. The reasons for using a qualifier are (1) to override the scope rules, given below, that assume a default table reference; and (2) to differentiate among simultaneous references to the same table in cases described below. In addition, labelling references to tables with *correlation-name*s may increase the readability of the application program even in cases where their use is not strictly required.

In any *table-reference* in a FROM clause, the *table-name* can be followed by a *correlation-name*. If a *correlation-name* appears, it is the name that specifies this table reference. If it does not appear, the *table-name* itself specifies this table reference. This name of the table reference is the name that can be used to qualify a *unqualified-column-name*.

Scope of Table Reference Names

The scope of a name that denotes a table reference is the entire innermost *sub-query*, *query-specification* or SELECT statement that contains the FROM clause.

Within a FROM clause, *table-names* that denote table references must be unique, and *correlation-names* must be unique and distinct from the unqualified *table-names* that denote table references.

The scope of a *table-name* in an UPDATE or searched DELETE statement is the entire statement.

If a *column-name* contains a qualifier, the *column-name* must be within the scope of that qualifier. If the *column-name* is within the scope of more than one qualifier with the specified name, the name refers to the qualifier with the most local scope. The specified table reference must be to a table that contains a column with the specified *unqualified-column-name*.

If a *column-name* does not include a qualifier, the *column-name* must be within the scope of, and is assumed to be qualified by, a qualifier whose associated table contains a column with the specified *unqualified-column-name*. If there is more than one possible qualifier, there must be only one with the most local scope and that one is assumed.

Multiple References to the Same Table

There are two notable cases:

- **A FROM clause references the same table twice.**

  A typical example of this is joining a table to itself (joining rows of the same table). Since there are two simultaneous distinct references to the same table, two distinct, explicit qualifiers are needed to distinguish between the two references. Thus, at least one of the *table-reference*s must contain a *correlation-name*.

- **A** *sub-query* **and an outer query reference the same table.**

  If the *sub-query* refers to both references — the table reference within the *sub-query* and the reference to the same table in the outer query — then the references are simultaneous, and qualifiers must be used to distinguish between the two occurrences. Thus, at least one of the *table-references* must contain a *correlation-name*.

  In this example, only the reference in the *sub-query* to the occurrence in the outer query needs an explicit qualifier for unique identification. The reference to the occurrence within the *sub-query* could be implicit because of the rule that the reference with the most local scope is assumed.

  If the value of a *sub-query* depends on the value of a column in a row of an outer query, the *sub-query* is a ‘‘correlated *sub-query*’’ and the reference to the column of the outer query is a ‘‘correlated reference’’. A correlated *sub-query* has to be evaluated once for each row of the outer query.

  A correlated reference need not be explicitly qualified unless required for uniqueness of reference. However, it may improve clarity to associate a *correlation-name* with the relevant *table-name* in the FROM clause of the outer query, and to qualify the correlated reference with this name.

If the *sub-query* refers only to its own occurrence, then this reference and the reference in the outer query are not simultaneous and do not need to be distinguished. This case is not a correlated reference.

3.12 Row Value Constructor

FUNCTION

Specify the value of each column of a table row.

SYNOPSIS

A *row-value-constructor* has the form:

```
(column-spec[, column-spec]...)
```

where each *column-spec* can be an *expression*, NULL, or DEFAULT.

DESCRIPTION

There are two cases in X/Open SQL syntax where the application uses a *row-value-constructor* to specify a complete row of a table:

- One form of the INSERT statement, which inserts a row into a table, lets the application specify values of each column in the new row.
- The OVERLAPS predicate, which compares two ranges of dates or times, deals with each range as a row containing two columns of type date/time or interval.

These valid contexts for a *row-value-constructor* impose restrictions on the number and values of the *column-spec*s.

The **degree** of a *row-value-constructor* is the number of *column-spec*s it contains.

Chapter 4

Embedded Aspects

This chapter begins the discussion of the actual contents of an embedded SQL program.

A host program can contain two categories of SQL constructs:

- **Declaratives**, specified in the data declaration part of a program, let the host program and the database system exchange data values and other information. The SQL declarative is the SQL declare section (see Section 4.2 on page 84).
- **Statements** appear in the procedural part of a program and specify operations on the contents of a database (metadata and data). This chapter discusses the following SQL statements that are usually non-executable:
  - — the DECLARE AUTHORIZATION statement (see Section 4.3 on page 86)
  - — the DECLARE CURSOR statement (see Section 4.4.1 on page 88)
  - — the dynamic DECLARE CURSOR statement (see Section 4.4.2 on page 89)
  - — the WHENEVER statement (see Section 4.6 on page 94).

  The executable SQL statements are discussed in Chapter 5.

4.1 Embedded SQL Host Program

4.1.1 Embedded SQL Constructs

SQL constructs are distinguished from programming language statements by starting with the prefix EXEC SQL and ending with a host-language-dependent special terminator. The prefix EXEC SQL must be specified on a single line of the source program and the keywords must be separated only by one or more spaces.

The **SYNOPSIS** sections of this specification generally do not include the prefix EXEC SQL nor the terminator. Section 4.2 on page 84 breaks this general rule in order to illustrate that the SQL declare section includes a header (BEGIN DECLARE SECTION) and a trailer (END DECLARE SECTION), *both of which* have the prefix EXEC SQL and a terminator. The escape clause (see Section 7.2 on page 181) is another exception; escape clauses do not use the prefix EXEC SQL nor the terminator but have their own delimiter syntax.

The rules for the placement of host-language comments in SQL constructs are those of the host language. SQL constructs embedded in host-language comments are treated as comments.

An SQL comment may appear anywhere in an SQL construct that a separator may appear (except between the keywords in EXEC SQL, EXEC SQL BEGIN DECLARE SECTION, and EXEC SQL END DECLARE SECTION). SQL comments are ignored during the compilation of the SQL construct.

Lower-case letters are allowed for SQL keywords and user-defined names.

The only token that may be split across lines is a character-string literal. This is achieved using the continuation syntax of the host language.

Special Rules for COBOL

- The SQL terminator is END-EXEC.
- SQL constructs must not be contained within library text processed by a COPY statement and must not be modified by a REPLACE statement.
- SQL constructs must be specified in Area B.
- An SQL construct must not be specified on a debugging line.
- SQL statements must be specified in the program's Procedure Division and may appear anywhere an imperative statement is allowed. A paragraph name may precede EXEC SQL on the line, subject to the usual restrictions in COBOL.

Special Rules for C

- The SQL terminator is a semicolon (;).
- SQL constructs must not be contained within an **#include** file and must not be modified by a **#define** directive.
- SQL statements may be specified anywhere a C statement may be specified within a function block. A label may precede EXEC SQL on the line, subject to the usual restrictions in C.

Additional constraints on the location of SQL declaratives are listed in Section 4.2 on page 84. Additional constraints on the use of SQL statements governed by a WHENEVER statement are listed in Section 4.6 on page 94.

4.1.2 Embedded Host Variables and Indicator Variables

Embedded host variables are normal host-language variables that executable SQL statements use to exchange data with the database.

A host variable in an SQL statement is immediately preceded by a colon (:). The term *embedded-variable-name* is thus defined as:

```
:host-identifier
```

For COBOL, a *host-identifier* is a data-name. For C, a *host-identifier* is a variable identifier. Any *host-identifier* used must be declared in the SQL declare section. It may be the same as an SQL keyword or user-defined name. (Some implementations restrict the names of these variables; see Section 7.3 on page 183.)

An *embedded-variable-name* used as a *descriptor-name* in an ALLOCATE DESCRIPTOR, DEALLOCATE DESCRIPTOR, GET DESCRIPTOR, SET DESCRIPTOR statement or USING clause identifies a host variable whose value identifies an SQL descriptor area. An *embedded-variable-name* used in the *get-descriptor-information* clause of a GET DESCRIPTOR statement identifies a host variable that is receiving a value from an SQL descriptor area. In all other contexts, an *embedded-variable-name* identifies a host variable that is supplying a value.

Indicator Variables

An embedded host variable may be accompanied by an indicator variable. An indicator variable is a host-language variable capable of holding a two-byte integer value. It indicates whether the associated embedded host variable contains the null value and whether the implementation truncated a character string value. (See Section 3.8 on page 56.)

Any indicator variable immediately follows its associated embedded host variable, except that the keyword INDICATOR may introduce the indicator variable. Like an embedded host variable, an indicator variable is preceded by a colon.

The term *host-variable-reference* refers to an embedded host variable with an optional indicator variable. Its syntax is:

```
embedded-variable-name [[INDICATOR ]indicator-variable]
```

where each of *embedded-variable-name* and *indicator-variable* is a *host-identifier* (defined above) preceded by a colon.

A *host-variable-reference* used in the INTO clause of a FETCH or SELECT statement identifies a host variable to which a column value is to be assigned. In all other contexts, a *host-variable-reference* identifies a host variable that is supplying a value.

Scope of Host Variables

It is implementation-defined whether the scope of a host variable follows the scope rules of the host language or the textual order of embedded SQL constructs. (For host variables referenced by a cursor specification, the scope rule applies to the OPEN statements rather than the DECLARE CURSOR statement.)

4.1.3 Data Types and Embedded Host Variables

An embedded host variable must have a description that corresponds to one of the SQL data types. The data type of the host variable is considered to be that corresponding data type. All embedded host variables are conceptually allowed to contain null values.

Here is the most nearly equivalent SQL data type for several COBOL data types:

COBOL Data Type	**X/Open SQL Equivalent**
PIC X(*n*); $1 \le n \le 254$	CHARACTER(*n*)*
PIC S9(5) or PIC S9(5) COMP-3 or PIC S9(4) COMP or PIC S9(4) BINARY	SMALLINT
PIC S9(10) or PIC S9(10) COMP-3 or PIC S9(9) COMP or PIC S9(9) BINARY	INTEGER

PIC S9(*m*)V9(*n*)
or PIC S9(*m*)V9(*n*) COMP-3
or PIC S9(*m*)V9(*n*) COMP
$m,n \ge 0$; $0 < m+n \le 15$
Note: V is optional if $n = 0$.
	DECIMAL(*m*+*n*,*n*)
	or
	NUMERIC(*m*+*n*,*n*)

(COMPUTATIONAL is a valid synonym for COMP and PICTURE is a valid synonym for PIC.) The CHARACTER VARYING, DATE, DOUBLE PRECISION, FLOAT, INTERVAL, REAL, TIME and TIMESTAMP data types of X/Open SQL have no direct equivalent in COBOL.

COBOL data definitions using the PACKED-DECIMAL storage type are not currently recognised as having equivalent SQL data types.

Here is the most nearly equivalent SQL data type for several C data types:

C Data Type	**X/Open SQL Equivalent**
char[*n*]; $2 \le n \le 255$	CHARACTER(*n*−1)*
short or **int**	SMALLINT (see below)
long	INTEGER
float	REAL
double	DOUBLE PRECISION

The DATE, DECIMAL, INTERVAL, FLOAT, NUMERIC, TIME and TIMESTAMP data types of X/Open SQL have no direct equivalent in C. The CHARACTER VARYING data type of X/Open SQL can be expressed in C using a special syntax defined in **Special Rules for C** on page 85.

* This host-language data type can also be used to represent values of type DATE, TIME, TIMESTAMP and INTERVAL (and REAL and DOUBLE PRECISION in COBOL, and DECIMAL and NUMERIC in C) after using the CAST function (see Section 3.9.6 on page 65) to obtain the string representation. The length must be sufficient to hold a value of the respective type (as defined for date/time values in Table 3-4 on page 43 and for intervals in **Length of an Interval** on page 46). (The length must be one greater than this in C due to the null terminator.)

Any C **int** variables used must be portable: that is, their absolute values must be less than 32768.

If an application uses a C **char** variable to assign a value to a column of a table, it must terminate the character string with a null byte (\0). When the implementation assigns a value to such a host variable, it pads the value with blanks and terminates it with a null byte (\0).

Some implementations may support larger ranges than shown in the above tables. The tables state the largest values that application writers may safely assume that all implementations support.

4.2 SQL Declare Section

FUNCTION

Define SQL-specific aspects of the compilation unit.

SYNOPSIS

```
EXEC SQL BEGIN DECLARE SECTION sql-terminator
      [SQL NAMES ARE character-set-name]
      [host-variable-definition]...
EXEC SQL END DECLARE SECTION sql-terminator
```

DESCRIPTION

A compilation unit may contain one or more SQL declare sections. They define the default character set and define host variables used in SQL statements.

The header (BEGIN DECLARE SECTION) and trailer (END DECLARE SECTION) are separately delimited as shown. The header and trailer, including their delimiters, must each be specified on a single line of the source program, and their keywords must be separated from each other only by one or more spaces.

Special Rules for COBOL

- The declare section must be placed in the Working Storage or Linkage Section of the Data Division.
- The declare section must not contain any COPY statements, REPLACE statements, or debugging lines, and must not be modified by any REPLACE statement.

Special Rules for C

- The declare section may be placed anywhere C variables may be declared.
- The declare section must not contain any **#include** or **#define** directives and must not be modified by any **#define** directive.

SQL NAMES Clause

The optional SQL NAMES clause defines a default character set for every user-defined name and character-string literal in the compilation unit (except those that explicitly declare their own character set, using the syntax discussed in Section 3.1.5 on page 36).

If an SQL NAMES clause does not appear in any SQL declare section in the compilation unit, user-defined names and character-string literals that do not declare their own character set are assumed to use an implementation-defined character set that contains at least every character in the **SQL character set** (see Section 2.4.3 on page 21).

Only one SQL NAMES clause is allowed anywhere in the compilation unit.

Host Variable Declarations

Every host variable that a statement references, including indicator variables and SQLSTATE, must be declared in exactly one SQL declare section, which must textually precede all SQL statements that reference that variable.

These variable declarations resemble the form of variable declarations in the host language, but they contain SQL-specific syntax and are subject to SQL-specific rules, listed below for COBOL and C.

Host variables declared in SQL declare sections are normal host program variables and may be referenced by host program statements without restriction.

Special Rules for COBOL

- Each declared variable must be an elementary data item with a level number of 01 or 77.
- A data item may contain PICTURE, SIGN, SYNCHRONIZED, USAGE and VALUE clauses. (For the uses of various PICTURE clauses, see Section 4.1.3 on page 82.) It may not contain the BLANK, JUSTIFIED, OCCURS or REDEFINES clauses.
- The following ISO 1989: 1985 COBOL Standard extensions are not allowed:

  — GLOBAL and EXTERNAL clauses
  — lower-case letters, except in non-numeric literals
  — the interchangeability of the separators comma, semicolon and space
  — picture string continuation
  — blank lines preceding continuation lines
  — the ALL figurative constant in the VALUE clause
  — symbolic character in the VALUE clause.
- To specify the character set of a character-string value, the syntax PIC X(*n*) or PICTURE X(*n*) can be preceded by the following syntax:

  `CHARACTER SET [IS]` *`character-set-name`*

  If this syntax is omitted, the variable uses the character set specified in the SQL NAMES clause (see above); if there is no SQL NAMES clause, the variable is assumed to use an implementation-defined character set. If the resulting character set is a multi-byte character set, then the effective length of the variable in the COBOL program is implicitly multiplied by the number of bytes of a character in that character set.

Special Rules for C

- A C variable must not be a pointer variable, and must not be part of a structure or union.
- The storage classes **auto**, **const**, **extern**, **static**, and **volatile** may be specified.
- A declarator may specify an initial value for the identifier being declared.
- In the specification of a **char** array, any of the following can be coded in place of the C keyword **char**:

  char `CHARACTER SET [IS]` *`character-set-name`*
  `VARCHAR`
  `VARCHAR CHARACTER SET [IS]` *`character-set-name`*

  A variable or array defined using the word **char** can be referenced in executable SQL statements as a variable of type CHAR.

  Use of the keyword VARCHAR lets the variable or array be referenced in executable SQL statements as a variable of type VARCHAR. The array uses a null termination byte to indicate the length of the meaningful data in the VARCHAR variable. The declared length of the C array must be 2 or greater.

  Use of the CHARACTER SET clause specifies a character set for its contents. If this syntax is omitted, the variable is assumed to use the character set specified in the SQL NAMES clause (see above); if there is no SQL NAMES clause, the variable is assumed to use an implementation-defined character set. If the resulting character set is a multi-byte character set, then the effective length of the variable or array in the C program is implicitly multiplied by the number of bytes of a character in that character set.

4.3 DECLARE AUTHORIZATION Statement

FUNCTION

Define an authorization identifier to be used to check the privileges for statements in the program.

SYNOPSIS

```
DECLARE AUTHORIZATION authorization-identifier [FOR STATIC ONLY]
```

P where *authorization-identifier* is defined as:

`character-string-literal`

DESCRIPTION

A DECLARE AUTHORIZATION statement provides an authorization identifier to be used when checking the privileges of the statements contained in a program.

The existence of a DECLARE AUTHORIZATION statement in the program indicates that the program is to be executed with definer's rights, as discussed in Section 2.6.3 on page 26.

If this statement is specified, it must be the first SQL statement in the program.

If FOR STATIC ONLY specified, *authorization-identifier* is only used to check static SQL. Any dynamic SQL executed using EXECUTE or EXECUTE IMMEDIATE statements uses the current authorization identifier for the session.

If FOR STATIC ONLY is not specified, *authorization-identifier* is used to check both static and dynamic SQL.

4.4 Cursors

A **cursor** is a means for a host program to gain access to a table, one row at a time.

A cursor is defined by a DECLARE CURSOR statement (see Section 4.4.1 on page 88) or by a dynamic DECLARE CURSOR statement (see Section 4.4.2 on page 89). Declaring a cursor defines the following:

- It defines a name for the cursor.
- It defines the **result set** of the cursor as a derived table based on a *query-expression* and it may define an ordering of the rows of that table. If it does not define an ordering, then the rows of the result set occur in an undefined order.

Statements that Affect Cursors

Executable SQL statements that operate on a cursor defined by DECLARE CURSOR are as follows: OPEN opens a cursor and CLOSE closes a cursor. While a cursor is open, FETCH extracts the row of the result set pointed to by the cursor, and updates the cursor position; positioned UPDATE updates the row of the result set pointed to by the cursor; and positioned DELETE deletes the row pointed to by the cursor.

Cursors defined by dynamic DECLARE CURSOR statements are operated on by the corresponding dynamic cursor statements: Dynamic OPEN, dynamic CLOSE, dynamic FETCH, dynamic positioned UPDATE, and dynamic positioned DELETE.

All these executable statements refer to the cursor using its declared name.

Openness and Cursor Position

During execution, information associated with a cursor is whether it is open, and, if it is open, its position:

A cursor is **open** when the application has used the OPEN statement on it and has not yet used the CLOSE statement on it. Otherwise the cursor is **closed**.

The **position** of an open cursor is either before a certain row, on a certain row, or after the last row. A cursor may be before the first row or after the last row even though the result set is empty. If the position is on a row, then that row is the **current row** of the result set.

If the cursor position is on or before a row that gets deleted through that cursor, then the cursor assumes a position before the next row. If such a row does not exist, then the cursor is positioned after the last row.

Other Cursor Effects

If a cursor is open, and the current transaction makes a change to the database other than through that cursor, then whether the change will be visible through that cursor before it is closed is undefined.

If an error occurs during the execution of an SQL statement that identifies an open cursor, then any effect on the position or state of that cursor is undefined, except where this specification explicitly defines an effect.

4.4.1 DECLARE CURSOR Statement

FUNCTION

Define a cursor.

SYNOPSIS

```
DECLARE cursor-name CURSOR FOR cursor-specification
```

where *cursor-specification* is defined as:

```
query-expression
  [ORDER BY sort-specification[, sort-specification]. ..]
  [FOR {READ ONLY
    | UPDATE [OF unqualified-column-name[, unqualified-column-name]. ..]}]
```

and *sort-specification* is defined as:

```
{unqualified-column-name | unsigned-integer} [ASC | DESC]
```

and *query-expression* is defined in Section 3.11.3 on page 75.

DESCRIPTION

A DECLARE CURSOR statement defines and names a cursor for subsequent use within the compilation unit.

A compilation unit may define several cursors, provided they have distinct names. The DECLARE CURSOR statement defining a cursor must textually precede any SQL statements that reference that cursor.

Updatability of the cursor

The application may specify FOR READ ONLY or FOR UPDATE, subject to restrictions below, to indicate whether it will perform updates using the cursor. Knowledge that a cursor will not be used for update may let the implementation optimise its processing of the cursor.

The cursor is updatable, and therefore can appear in positioned DELETE and positioned UPDATE statements, if and only if all of the following are true:

1. The *query-expression* is updatable (see Section 3.11.3 on page 75).
2. The application does not specify ORDER BY.
3. The application does not specify FOR READ ONLY.

If the cursor is not updatable because conditions (1) or (2) above are not met, then the application may also specify FOR READ ONLY but it must not specify FOR UPDATE. If conditions (1) and (2) above are both met, FOR UPDATE is implicit (with no specified columns). The application may specify FOR UPDATE or FOR READ ONLY.

A FOR UPDATE clause may specify a list of columns. If it does, these are the only ones that a positioned UPDATE statement can update using this cursor. Any columns specified must be columns of the table referenced by the (single) FROM clause of the *query-specification* (but need not be columns of the result set of the cursor). If the FOR UPDATE clause does not specify columns, it implicitly refers to all columns of the *query-specification*.

Ordering of Rows

An application can control the ordering of rows in the result set by using ORDER BY. If the application does not do so, the ordering is undefined. ORDER BY introduces one or more *sort-specification*s. Each identifies a column, either by its name or by an unsigned integer that represents its ordinal position within the result set. (To specify a column whose name is not known, the application must use the unsigned integer.[25])

The rows are ordered in ascending order of that column's values unless the application specifies DESC (descending), in which case they are ordered in descending order. The application may redundantly specify ASC for ascending.

If there is more than one *sort-specification*, then the rows are ordered first according to the first *sort-specification*. Where the specified column contains duplicate values, those rows are then ordered according to the second *sort-specification*, and so on. Rows that have duplicate values in the columns specified by all the *sort-specification*s appear in the result set together, but in an undefined order.

Ordering is determined by the comparison rules defined in Section 3.6 on page 53. (See Section 3.7 on page 54 for a description of implicit comparison methods on the null value.)

Implementation

No code is generated for a DECLARE CURSOR statement. Instead, the application effectively executes the statement each time it opens the defined cursor. This is also the point at which any referenced host variables are evaluated.

4.4.2 Dynamic DECLARE CURSOR Statement

FUNCTION

Define a dynamic cursor.

SYNOPSIS

DECLARE *cursor-name* CURSOR FOR *statement-name*

DESCRIPTION

A dynamic DECLARE CURSOR statement associates *cursor-name* with *statement-name*. The description of the DECLARE CURSOR statement in Section 4.4.1 on page 88 applies to the dynamic DECLARE CURSOR statement too.

The dynamic DECLARE CURSOR statement must be preceded by a PREPARE statement for this *statement-name* in the text of an embedded host program.

To refer to the *cursor-name* by positioned DELETE or positioned UPDATE statements, the *embedded-variable-name* specified in the associated PREPARE statement must contain a FOR UPDATE clause.

It is implementation-defined whether multiple dynamic DECLARE CURSOR statements can reference the same *statement-name*.

25. To ensure portability, an application should use unsigned integers to specify columns whose names are implementation-generated, even if the actual names are known to the application.

4.5 SQL Statement Outcomes

Information on the outcome of executable SQL statements notifies applications of the actual effects and may let them take compensatory action to convert failure to success. X/Open-compliant implementations predictably return specified error information.

X/Open specifies the SQLSTATE and SQLCODE status variables that an application can include in its SQL declaratives. The implementation uses these variables to return status information to the application.

In addition, the diagnostics area may be able to report multiple outcome records for a single executable SQL statement. Each record may include a text description of the outcome.

4.5.1 Outcome Categories

The effect of an executable SQL statement can fall into four general categories:

- **Success** means that the statement executed successfully without limitation.
- **Success with warning** means that the statement executed successfully but its effect was limited and may not be as the application desired.
- **No data** means that the statement executed successfully but that there were no rows that satisfied conditions in the statement, or the statement operated on no rows of the table. (This implies that the statement has made no change to the database.) **No data** occurs in these situations:
  - — The FETCH statement fetches no row. (This typically means that the implicit movement of the cursor failed because the cursor was already at the end of the set.)
  - — The SELECT statement produces an empty table as its result.
  - — The condition specified in an INSERT, searched DELETE or searched UPDATE statement is not satisfied.
- **Error** means that the statement did not execute successfully. The statement makes no change to the database. The statement's effect on associated host variables and on SQL descriptor areas is undefined unless this document explicitly specifies an effect.

4.5.2 SQLSTATE Status Variable

An application can define a host variable named SQLSTATE in its SQL declare section. SQLSTATE describes the result (or primary exception code) of the most recently executed SQL statement.

A list of valid SQLSTATE values appears in Appendix B.

If a C program defines SQLSTATE, its data type must be **char**[6]. If a COBOL program defines SQLSTATE, its data type must be PIC X(5).

Fields of SQLSTATE

SQLSTATE is a 5-character string that an SQL statement returns to indicate status. SQLSTATE can contain only digits and capital letters. The first two characters of SQLSTATE indicate a class; the following three characters indicate a subclass. Class codes are unique, but subclass codes are not; the meaning of a subclass code depends on the class code that accompanies it.

The initial character of the class and subclass indicates the source document that defines that return condition:

- Class codes starting with 0-4 or A-H mean that the result condition is defined in the International Standard (or, for class ’**HZ**’, in ISO RDA). In this case, subclass codes for conditions specified in the same standard also start with 0-4 or A-H. The meaning of all other subclass codes is implementation-defined.
- Class codes starting with 5-9 or I-Z denote implementation-specific conditions. In this case, subclass codes can start with any character. The meaning of all subclass codes is implementation-defined, except that implementations may not define ’000’ but return ’000’ when there is no subclass information.

Application Use of SQLSTATE

Applications can test just the class (first 2 characters) of SQLSTATE, to determine the category of result. Class ’**00**’ indicates **Success**; class ’**01**’ indicates **Success with warning**; class ’**02**’ indicates **No data**; and other class codes each describe a general group of **Errors** (for example, ’**07**’ indicates a dynamic SQL error). An application can test the entire string (the 2-character class code and the 3-character subclass code) to obtain a more precise error report.

4.5.3 Diagnostics Area

The diagnostics area holds a sequence of diagnostic records, with information pertaining to the most recently executed SQL statement. This information can be obtained using the GET DIAGNOSTICS statement.

The first record (exception number 1) corresponds to the SQLSTATE value set by the most recent SQL statement other than GET DIAGNOSTICS. That is, all executable SQL statements set SQLSTATE as though they had been followed by:

```
GET DIAGNOSTICS EXCEPTION 1 :SQLSTATE = RETURNED_SQLSTATE
```

(However, the scope of GET DIAGNOSTICS and of SQLSTATE may differ. Since the diagnostics area is global to the application, exception number 1 may not correspond to the value of SQLSTATE in the compilation unit that executes the GET DIAGNOSTICS statement.) The association between other exception numbers and other exceptions raised by that SQL statement is undefined.

The number of records in the diagnostics area is a transaction attribute, settable using SET TRANSACTION (see Section 5.6.4 on page 140).

There is only one diagnostics area for the application, regardless of the number of connections or compilation units.

Multiple Non-success Outcomes — Primary Outcome

Multiple conditions may occur that would produce outcomes other than **Success**. If there are two or more status records, they occur in an undefined order, except that the *primary outcome*, which is placed in the first record of the diagnostics area and is also assigned to SQLSTATE, is defined as the highest-ranking condition using the following ranking:

1. **Errors.** If two or more status records describe the same condition, then the standardised or X/Open-defined SQLSTATE (classes from ’**03**’ to and including ’**HZ**’) outranks the implementation-defined SQLSTATE.[26]

26. In addition, records that indicate a failure or possible failure of the transaction outrank records that indicate statement failure.

2. **Warnings** values in class '**01**'.
3. **Implementation-defined No Data values** in class '**02**'.

If the implementation generates two or more status records that all have the highest rank of any status record generated, then it is undefined which of these is the first record.

In the remainder of the diagnostics data structure, an application cannot assume that errors precede warnings. Applications should scan the entire diagnostic data structure to obtain complete information on the non-success outcome of a CLI function.

Access Violations

Some users are not entitled to know whether a table or other object exists. In this case, SQLSTATE reports an access violation. If the user is entitled to more precise information, the diagnostics area may contain additional result indications. See **Access Violations** on page 99.

Omissions from the Diagnostics Area

Implementations are required to place a record corresponding to the outcome that SQLSTATE reports in record 1 of the diagnostics area. However, despite the hierarchy of outcomes presented above, implementations are not required to populate the diagnostics area with other outcomes that might apply to the execution of the SQL statement.

Textual Outcome Description

The MESSAGE_TEXT field of a diagnostic record may contain informative text on the statement outcome that the record reports. However, even when the text is meaningful, it is not standardised and is not a portable way for applications to test result status. Implementations may return a zero-length string for diagnostic text.

4.5.4 SQLCODE Status Variable

An application can define a host variable named SQLCODE in its SQL declare section. This must be a signed, 4-byte integer capable of holding a value in the INTEGER range. The value of SQLCODE indicates the outcome of the most recently executed SQL statement:

- An SQL statement that achieves **Success** or **Success with warning** sets SQLCODE to 0.
- The implementation reports the **No data** outcome by setting SQLCODE to +100. (Some implementations use positive numbers for certain **error** conditions; see Section 7.6 on page 184.)
- Negative values of SQLCODE indicate the **Error** outcome, but do not portably describe the reason for the error.

SQLSTATE provides more precise information than SQLCODE. X/Open does not specify SQLCODE values for all SQL statement outcomes. Applications should test SQLSTATE, not SQLCODE, to determine the status of an executable SQL statement. If a C program defines SQLCODE, its data type must be **long**. If a COBOL program defines SQLCODE, its data type must be PIC S9(9) COMP.

4.5.5 Application Usage

Preferred Error Detection Methods

An application should test for exceptions using one of these methods:

- Examine the status variable SQLSTATE.

  If SQLSTATE indicates an outcome other than unqualified success ('**00**000'), execute GET DIAGNOSTICS. This statement retrieves from the diagnostics area (see Section 4.5.3 on page 91) one or more error or warning indications that apply to the most recently executed SQL statement. The indications are in the SQLSTATE (5-character) format. Some implementations may also provide diagnostic text for some or all indications.

- Use the WHENEVER statement (see Section 4.6 on page 94) to force subsequent executable SQL statements to effect a transfer of control (GOTO) in the **Success with warning**, **No data** and **Error** categories. The WHENEVER statement adds code to each executable SQL statement to effect the transfer of control if any selected result category occurs.

Application Coding Options

X/Open intends this specification to support even application programs written before SQLSTATE was developed. Different compilation units may define different destination variables to hold status information. The implementation returns status in variables based on the following rules:

- If the application defines either the SQLSTATE or SQLCODE status variable in the SQL declare section, the implementation sets the defined variable.
- If the application defines both SQLSTATE and SQLCODE in the SQL declare section, the implementation sets both. When at least one of the status variables is defined in the SQL declare section, the implementation does not return information in any SQLCODE or SQLSTATE variables defined outside the SQL declare section.
- When neither of the status variables is defined in the SQL declare section, the implementation assumes there is an SQLCODE variable defined outside the SQL declare section and uses it. The implementation does not check the application to ensure that SQLCODE is a valid variable name, so this option can produce compilation or execution errors. (If the application defines an SQLSTATE variable outside the SQL declare section, the implementation does not use it.)

4.5.6 Other Effects of Errors

Some errors may close an open cursor, or end an active transaction with an implicit ROLLBACK statement.

Some distributed implementations do not correctly notify the application of the outcome of COMMIT in cases where communication errors occur. See Section 7.5 on page 184.

4.6 WHENEVER Statement

FUNCTION

Enable or disable subsequent executable SQL statements from transferring control if they produce exceptions.

SYNOPSIS

```
WHENEVER {SQLERROR | SQLWARNING | NOT FOUND}
      {{GOTO | GO TO} host-label | CONTINUE}
```

DESCRIPTION

The WHENEVER statement affects executable SQL statements that textually follow it. The keyword GOTO (or the equivalent GO TO) indicates that applicable SQL statements may transfer control to *host-label*. The keyword CONTINUE specifies no special transfer of control following applicable SQL statements; this form is typically used to cancel the effect of a WHENEVER . . . GOTO statement.

Each WHENEVER statement relates to a single statement outcome class. The keywords NOT FOUND, SQLERROR and SQLWARNING correspond to the outcomes **No data**, **Error** and **Success with warning** respectively, as specified in Section 4.5 on page 90. Such a WHENEVER statement has no effect on invocations of SQL statements that produce a different outcome. Up to three WHENEVER statements, one for each of the three outcomes, can be in force for each SQL statement. The WHENEVER statement relating to a given outcome class remains in force until another WHENEVER statement relating to the same outcome class appears.

For example, if an executable SQL statement produces **No data**, then it may transfer control depending on the nearest WHENEVER NOT FOUND statement that precedes it textually. If this statement used GOTO, the executable SQL statement transfers control to the *host-label* specified in that WHENEVER statement. If this statement used CONTINUE, or if there is no such statement textually preceding the executable SQL statement, then there is no special transfer of control.

No code is generated for a WHENEVER statement at the point at which it is declared. Rather, appropriate code is effectively generated immediately following those executable SQL statements within its scope.

4.6.1 Special Rule for C

- *host-label* must be a label defined in the same program block or in a block of a higher scope.

4.6.2 Special Rules for COBOL

- *host-label* must be a section name or unqualified paragraph name.
- Executable SQL statements which could transfer control because of prior use of a WHENEVER . . . GOTO statement:
  - — must not be within the scope of an IF statement terminated by its explicit scope terminator END-IF
  - — must not immediately precede a scope terminator (for example, ELSE, NOT ON SIZE ERROR or END-ADD), except for a terminating period
  - — are essentially conditional statements; ISO 1989: 1985 COBOL Standard implementations that treat such SQL statements as conditional statements further restrict the locations where they can appear.

4.7 Multiple Compilation Units

An embedded SQL host program may invoke other embedded SQL host programs written in the same host language. In these cases, the following rules apply:

- The scope of a cursor declaration is limited to a single compilation unit. The same *cursor-name* must not be used in more than one compilation unit.
- The scope of a *statement-name* is limited to a single compilation unit. The same *statement-name* must not be used in more than one compilation unit.
- The scope of a *descriptor-name* is limited to a single compilation unit. The same *descriptor-name* must not be used in more than one compilation unit.
- The scope of an exception declarative is limited to a single compilation unit.
- A host variable can be used in SQL statements only in that compilation unit in which it is declared.
- The flow of control may freely cross the boundaries of compilation units. Crossing a unit boundary has no effect on transactions or the states of cursors.
- The limit on the number of open cursors applies to the application as a whole, not to the individual compilation units.

In this context, a COBOL source program contained in another COBOL source program is considered a separate compilation unit.

Chapter 5

Executable SQL Statements

This chapter gives a synopsis of each SQL statement's syntax, describes its behaviour, and lists its associated diagnostics, such as SQLSTATE values.

5.1 Classification of SQL Statements

The following table shows the classification and function of each executable SQL statement:

Name	Function	Section
Create/Drop Schema		
CREATE SCHEMA	Create a schema.	**5.3.6**
DROP SCHEMA	Drop a schema.	**5.3.13**
Other Data Definition Statements		
ALTER TABLE	Add or destroy a column in an existing base table.	**5.3.2**
CREATE CHARACTER SET	Create a character set.	**5.3.3**
CREATE COLLATION	Create a collation.	**5.3.4**
CREATE INDEX	Create an index on a base table.	**5.3.5**
CREATE TABLE	Create a base table.	**5.3.7**
CREATE TRANSLATION	Create a translation.	**5.3.8**
CREATE VIEW	Create a viewed table.	**5.3.9**
DROP CHARACTER SET	Destroy a character set.	**5.3.10**
DROP COLLATION	Destroy a collation.	**5.3.11**
DROP INDEX	Destroy an index.	**5.3.12**
DROP TABLE	Destroy a base table.	**5.3.14**
DROP TRANSLATION	Destroy a translation	**5.3.15**
DROP VIEW	Destroy a view.	**5.3.16**
GRANT	Grant privileges on a table.	**5.3.17**
REVOKE	Revoke privileges on a table.	**5.3.18**
Data Manipulation Statements		
CLOSE	Close a cursor.	**5.4.1 ***
Positioned DELETE	Delete a row of a table.	**5.4.2 ***
Searched DELETE	Delete rows based on a *search-condition*.	**5.4.3**
FETCH	Advance the cursor and fetch a row of a table.	**5.4.4**
INSERT	Insert rows into a table.	**5.4.5**
OPEN	Open a cursor.	**5.4.6**
SELECT INTO	Assign table values to host variables.	**5.4.7**
Positioned UPDATE	Update a row of a table.	**5.4.8 ***
Searched UPDATE	Update a table based on a *search-condition*.	**5.4.9**

* The section defines both the data manipulation statement and the corresponding dynamic SQL statement.

Name	Function	Section
Dynamic SQL Statements		
ALLOCATE DESCRIPTOR	Allocate an SQL descriptor area.	**5.5.3**
Dynamic CLOSE	Close a dynamic cursor.	**5.4.1 ***
DEALLOCATE DESCRIPTOR	Deallocate an SQL descriptor area.	**5.5.4**
Dyn. Positioned DELETE	Delete a row of a table.	**5.4.2 ***
DESCRIBE	Get information about a prepared statement.	**5.5.5**
EXECUTE	Execute a prepared statement.	**5.5.6**
EXECUTE IMMEDIATE	Execute an SQL statement in a host variable.	**5.5.7**
Dynamic FETCH	Fetch a row for a cursor.	**5.5.8**
GET DESCRIPTOR	Get information from an SQL descriptor area.	**5.5.9**
Dynamic OPEN	Associate input values and open a cursor.	**5.5.10**
PREPARE	Prepare a statement for execution.	**5.5.11**
SET DESCRIPTOR	Assign values to an SQL descriptor area.	**5.5.12**
Dyn. Positioned UPDATE	Update a table based on a *search-condition*.	**5.4.8 ***
Transaction Control Statements		
COMMIT	Successfully complete the current transaction.	**5.6.2**
ROLLBACK	Cancel the current transaction.	**5.6.3**
SET TRANSACTION	Set the attributes of the next transaction.	**5.6.4**
Connection Statements		
CONNECT	Connect the client with a server.	**5.7.6**
DISCONNECT	End a connection between a client and a server.	**5.7.7**
SET CONNECTION	Make a specified connection the current one.	**5.7.8**
Session Statements		
SET CATALOG	Set the default catalog name.	**5.8.1**
SET NAMES	Set the default character set	**5.8.2**
SET SCHEMA	Set the default schema name.	**5.8.3**
SET SESSION AUTHORIZATION	Set the session authorisation identifier	**5.8.4**
Diagnostic Statement		
GET DIAGNOSTICS	Get information from the diagnostics area.	**5.9**

* The section defines both the data manipulation statement and the corresponding dynamic SQL statement.

5.2 General Diagnostics

This section is important since the following material applies to many types of SQL statement. It is not repeated in the sections that define the applicable statements.

General diagnostics also appear in the following locations:

- Section 5.3.1 on page 102 for data definition statements
- Section 5.6.1 on page 139 for transaction control statements
- Section 5.7.5 on page 143 for connection statements.

5.2.1 Syntax Checking

Syntax checking of an SQL statement includes tests of all of the following:

- that the statement syntax conforms to the **SYNOPSIS** given in this chapter
- that the statement satisfies other syntactic rules stated in text in the **DESCRIPTION** sections
- access violations and adequacy of references to objects, as discussed below.

Syntax checking is typically done at compile-time. However, in some situations, such as sending an SQL statement to a remote server, run-time syntax checking occurs and SQLSTATE is set to ('**42**000') for syntax errors.

Access Violations

Access to each database object is subject to privileges (see the GRANT statement in Section 5.3.17 on page 115 and the REVOKE statement in Section 5.3.18 on page 116). Any applicable data definition statements, data manipulation statements, and dynamic SQL statements detect violations of these privileges. A violation is reported as a syntax error ('**42**000', but see also below).

A general rule implicit in the International Standard, and made more explicit in emerging standards, is that a database system *may* check security at compile-time, but *must* validate access again at execution. It is unacceptable for compiled database access routines to perpetuate the security policy that was in effect when they were compiled.

Disclosure of Access Violations

For certain types of access violation, the implementation is able to disclose additional information that some applications are not entitled to obtain. In all these cases, SQLSTATE is set to '**42**000'. However, the **DIAGNOSTICS** sections in this chapter specify a different value, from the following sets:

- diagnostics of invalid cardinality, which begin with '**21**S'
- diagnostics of existence of an object in a schema, which begin with '**42**S'.

If the application is entitled to know the exact cause of the failure, the codes listed above may appear in the diagnostics area.

Object References

The following diagnostics are examples of access violations that are issued when an object of the type given in the left-hand column is specified in an embedded SQL statement and does not exist in the specified or implied schema. The reporting of these diagnostics is subject to the rules specified above. Following the table are exceptions under which the diagnostics are not reported.

Object Type	Syntax Reference	Diagnostic
Catalog	Section 3.1.3 on page 35	'**42**S42'
Character set	Section 3.1.5 on page 36	'**42**S52'
Collation*	Section 3.9.7 on page 66	'**42**S62'
Conversion*	**CONVERT** on page 60	'**42**S72'
Schema	Section 3.1.3 on page 35	'**42**S32'
Table	[various]	'**42**S02'
Translation*	**TRANSLATE** on page 62	'**42**S82'

Table 5-1 Object Types with General Diagnostics for Existence Checking

Exceptions. It is not an error that a referenced object does not exist if either of the following applies:

- The reference occurs in a CREATE statement that is bringing the object into existence.
- The reference occurs in a statement that is enclosed in a CREATE SCHEMA statement and the referenced object is defined somewhere in the same CREATE SCHEMA statement.

5.2.2 Expression Errors

Any data manipulation statement or dynamic SQL statement with a *query-specification* can produce the expression errors specified in Section 3.9 on page 57.

5.2.3 Assignment Errors

Any SQL statement that assigns values to embedded host variables or to database columns can produce the data transfer errors described in Section 3.5 on page 51.

Certain SQL statements assign values to host variables or to DATA and INDICATOR fields of an SQL descriptor area. X/Open does not specify the sequence of these assignments nor the result value of an assignment in which an error occurs. If such an SQL statement produces an assignment error, the contents of all host variables or SQL descriptor area fields that should have been modified is undefined.

* The diagnostic is not raised if the referenced object name is in an implementation-defined list of built-in objects of that type. In the case of character sets, Section 1.3.4 on page 6 specifies character sets that are built-in on all X/Open-compliant implementations.

5.2.4 Constraint Checking

At the completion of any SQL statement that changes the contents of a table, all applicable constraints must be satisfied ('**23**000'). For example, a null value cannot be given to a column constrained by NOT NULL; a duplicate value cannot be given to a column constrained by UNIQUE; and any foreign key or primary key referential constraint or CHECK constraint that applies to the table or to any column must continue to be satisfied. Constraints are specified in the CREATE TABLE statement that created the table, and are discussed in Section 5.3.7 on page 107.

5.2.5 Read-only Transaction Violation

The following statements cannot be executed within a transaction whose access level is READ ONLY ('**25**000'):

- any data definition statement
- DELETE, INSERT and UPDATE.

The transaction access level is specified by the SET TRANSACTION statement (see Section 5.6.4 on page 140).

5.2.6 Connection Errors

Connection statements give the application access to multiple servers, some of which may be remote. X/Open SQL syntax does not distinguish between local and remote servers. Subsequent SQL statements can report failure if the application is disconnected from the server, either through its use of the DISCONNECT statement or because of communication errors, administrative action, or other events. See Section 5.7.5 on page 143 for details.

Communication errors are reported promptly; a success return indicates that the current statement completed without communication errors.

5.3 Data Definition Statements

A data definition statement modifies the metadata of the database. Data definition statements define tables and views; data manipulation statements store and retrieve data from them.

For the data manipulation statements to work correctly, the definitions of the tables they reference must exist at compile-time. For example, the CREATE TABLE or CREATE VIEW statements that define specific tables must have been executed before compiling a program that inserts data into those tables. If the metadata with respect to those tables subsequently changes, then the behaviour of the program is implementation-defined.

Applications should not mix data definition statements and data manipulation statements within transactions. See Section 7.4 on page 183.

5.3.1 General Diagnostics

Only the owner of a schema can alter the definition of the objects within the schema. Therefore, the current user must be the owner of the schema that contains the object referenced by ALTER TABLE, CREATE INDEX, DROP INDEX, DROP TABLE or DROP VIEW; or the owner of the schema dropped by DROP SCHEMA ('**42**000'). (The CREATE SCHEMA statement lets a user create and populate a schema to be owned by another user.)

5.3.2 ALTER TABLE

NAME

ALTER TABLE — Add or destroy a column in an existing base table.

SYNOPSIS

```
ALTER TABLE base-table-name
      ADD [COLUMN] column-definition
      | DROP [COLUMN] identifier {CASCADE | RESTRICT}
```

where *column-definition* is as defined in Section 5.3.7 on page 107.

DESCRIPTION

The ALTER TABLE statement adds a new column to, or drops an existing column from, *base-table-name*.[27]

Adding a Column

The ADD clause specifies the column name and data type of a new column, and may also specify a default value and column constraints, using the same syntax as in the CREATE TABLE statement (see Section 5.3.7 on page 107). The new column appears at the right-hand side of the table, so that a new value is added at the end of every existing row of the table. The column's initial value is the default value. (If ALTER TABLE does not specify a default value, the default value is the null value.)

Adding a column to a table has no effect on any existing *query-specification* contained in a view definition, since any implicit column references are resolved when the view definition is originally executed.

27. A former alternative syntax for the ALTER statement that specified several columns in parentheses is no longer supported. To add more than one column to an existing base table, use multiple ALTER TABLE statements.

Dropping a Column

In an ALTER TABLE statement that contains the DROP clause, the application must use the keyword CASCADE or RESTRICT. Doing so specifies the effect if the dropped column is referenced in the *query-specification* of a view definition or in the *search-condition* of a constraint:

CASCADE	All such viewed tables and constraints are also dropped.
RESTRICT	The statement is invalid.

A column cannot be dropped when it is the only column in the table.

DIAGNOSTICS

The ALTER TABLE statement must not specify any column twice and must not identify any existing columns in the base table ('**42**S21').

When adding a column that does not allow null values to a non-empty table, the ALTER TABLE statement must specify a non-null default value ('**23**000').

The **Success with warning** outcome occurs if a default value was specified (see Section 5.3.7 on page 107, DEFAULT clause) whose length exceeds 254 characters ('**01**00B'). The warning indicates that the application cannot retrieve the default value by querying the COLUMNS system view (see *COLUMNS* on page 160).

APPLICATION USAGE

The ALTER TABLE...ADD COLUMN statement, which sets the new column to its default value in all rows of the table, fails due to a constraint violation in the following cases:

- If the new column is constrained to be NOT NULL, the default value is null and the table is not empty, then an immediate violation of the NOT NULL constraint occurs because the statement would extend all existing rows to contain the new column with a null value.
- If the new column is constrained to be UNIQUE, the default value is non-null and the table has two or more rows, then an immediate violation of the UNIQUE constraint occurs because the statement would extend all existing rows to contain the new column with the same value.

5.3.3 CREATE CHARACTER SET

NAME

CREATE CHARACTER SET — Create a character set.

SYNOPSIS

```
CREATE CHARACTER SET new-char-set [AS]
      GET character-set-name
      [COLLATE collation-name
      | COLLATION FROM collation-source]
```

where *collation-source* is as defined in Section 5.3.4 on page 104.

DESCRIPTION

The CREATE CHARACTER SET statement creates a character set named *new-char-set* derived from an existing character set. The statement also defines the default collation for *new-char-set*.

If a collation is specified using the COLLATE or COLLATION FROM clause, then it is the default collation for *new-char-set*. The COLLATE syntax allows reference to an existing, named collation. The COLLATION FROM syntax allows use of the more general syntax for specifying a collation, as defined in Section 5.3.4.

If a collation is not specified in the CREATE CHARACTER SET statement, then the default collation for *new-char-set* is the same collation that would be achieved by use of the statement CREATE COLLATION ... FOR *new-char-set* FROM DEFAULT (see Section 5.3.4).

DIAGNOSTICS

The *new-char-set* must not identify an existing character set ('**42**S51').

5.3.4 CREATE COLLATION

NAME

CREATE COLLATION — Create a collation.

SYNOPSIS

```
CREATE COLLATION new-collation FOR character-set-name
      FROM collation-source
      [PAD SPACE
      | NO PAD]
```

where *collation-source* is as follows:

```
collation-name
| DEFAULT
| DESC(collation-name)
| EXTERNAL(character-string-literal)
| TRANSLATION translation-name [THEN COLLATION collation-name]
```

DESCRIPTION

The CREATE COLLATION statement creates a collation named *new-collation*, defined on the specified, existing character set. A collation is a sort order to be used when character strings of the specified character set are compared.

The *collation-source* specifies the collation. The following forms are valid:

- The *collation-source* can be a *collation-name* of an existing collation. The effect of a CREATE COLLATION statement that uses this form is basically to define a new name for the existing collation. (However, the pad attribute of the new collation may be defined to be different from that of the existing collation; see below.)

- The word DEFAULT specifies that the collation is to be performed using the order of characters as they appear in the character repertoire.

  When a CREATE CHARACTER SET does not specify a default collation for the new character set, the character set uses this same DEFAULT rule.

- The syntax DESC(*collation-name*) specifies that the new collation is the same collation as *collation-name* except that the order is reversed.

- The syntax EXTERNAL(*character-string-literal*) specifies an implementation-defined EXTERNAL collation. The set of valid values for the literal is implementation-defined.

  An implementation can define a certain collation as a built-in *collation-name*. This means it can be used in embedded SQL programs. By contrast, an implementation can also define a certain collation as a valid EXTERNAL collation. This means it must be assigned a *collation-name* using the CREATE COLLATION statement before it can be used.

- The TRANSLATION clause specifies that, in comparing two strings under the collation being defined, each string is first translated using *translation-name*.

— If the THEN COLLATION clause is present, then the results of the above translation are compared using *collation-name*.

— If the THEN COLLATION clause is not present, then the results of the above translation are compared using the default collation defined for the target character set of *translation-name*.

The CREATE COLLATION statement can specify a pad attribute for the new collation using either the PAD SPACE clause or the NO PAD clause. If neither clause is present, PAD SPACE is assumed, except that if *collation-source* uses any syntax that specifies a *collation-name*, then the pad attribute of that collation becomes the pad attribute of the collation being defined.

DIAGNOSTICS

The *new-collation* must not identify an existing collation ('**42**S61').

5.3.5 CREATE INDEX

NAME

CREATE INDEX — Create an index on a base table. Ensure uniqueness of the values in the specified columns.

SYNOPSIS

```
CREATE [UNIQUE] INDEX index-name
      ON base-table-name (unqualified-column-name [ASC | DESC]
      [, unqualified-column-name [ASC | DESC]]...)
```

DESCRIPTION

The CREATE INDEX statement creates an index named *index-name* on the existing base table *base-table-name*. The schema specified or implied by *index-name* must be the schema in which the base table resides.

The index key is constructed of columns from the specified base table in the given order of significance. When ASC is specified, the order of the referenced column is ascending. When DESC is specified, the order of the referenced column is descending. The default order is ascending.

UNIQUE indicates that at most one row is allowed in the table for each combination of values in the specified columns. For the purpose of this clause two null values are considered equal. The constraint is enforced when rows are inserted or updated and checked during execution of the CREATE INDEX statement.

DIAGNOSTICS

The *base-table-name* must identify a table that already exists ('**42**S02'). However, the *index-name* must not identify an existing index ('**42**S11').

Any specified column must already exist ('**42**S22'). (See the exception in **Object References** on page 100.)

5.3.6 CREATE SCHEMA

NAME

CREATE SCHEMA — Create a schema.

SYNOPSIS

```
CREATE SCHEMA [object-qualifier] [AUTHORIZATION auth-id]
      [DEFAULT CHARACTER SET character-set-name]
      [schema-element [schema-element]...]
```

where each *schema-element* is a CREATE TABLE, CREATE VIEW, CREATE INDEX or GRANT statement.

DESCRIPTION

The CREATE SCHEMA statement creates a schema, a unit that can contain tables, indexes and privileges.

OP If *object-qualifier* is of the form *schema-name*, then it specifies the name of the new schema. If it is of the form *catalog-name.schema-name*, then *schema-name* is the name of the new schema and the schema resides in the catalog *catalog-name*. (If the catalog is not specified, the catalog in which the schema resides is implementation-defined.) If *object-qualifier* is omitted, then the name of the new schema is the user name specified by the AUTHORIZATION clause, which must be present.

The AUTHORIZATION clause specifies, as *auth-id*, the owner of the new schema. If the AUTHORIZATION clause is omitted, the current user owns the new schema.

If the DEFAULT CHARACTER SET clause is present, then *character-set-name* is the name of the character set that all data within the schema will use, unless contrary declarations are made on a more local level, such as by the CREATE TABLE statement for specific columns of a table. If the clause is omitted, then the default character set for the schema is implementation-defined.

By finishing the CREATE SCHEMA statement with one or more *schema-element*s, the user can specify the initial contents of the schema. This can be some sequence of the following statements:

- the CREATE INDEX statement specified in Section 5.3.5 on page 105
- the CREATE TABLE statement specified in Section 5.3.7 on page 107
- the CREATE VIEW statement specified in Section 5.3.9 on page 111
- the GRANT statement specified in Section 5.3.17 on page 115.

The statements can appear in any order; in fact, the definition of a view or index can precede the definition of the underlying base table, and tables can reference columns in tables that appear later in the CREATE SCHEMA statement.

A CREATE SCHEMA statement need not specify any *schema-element*. Objects can be created and privileges granted by data definition statements executed subsequently.

DIAGNOSTICS

The *object-qualifier* must not identify an existing schema (’**42**S31’).

There are no other diagnostics specific to CREATE SCHEMA. However, if *schema-element*s are specified, these operations can fail with the diagnostics specified in the respective section. If any error occurs executing a *schema-element*, then CREATE SCHEMA returns the associated diagnostics and does not create the schema.

APPLICATION USAGE

Applications can use the AUTHORIZATION clause to specify *auth-id* as the owner of the created schema. The list of *schema-element*s in CREATE SCHEMA is the user's last chance to affect the set of objects in the schema, because the user will not own the schema when CREATE SCHEMA completes.

5.3.7 CREATE TABLE

NAME

CREATE TABLE — Create a base table.

SYNOPSIS

```
CREATE TABLE base-table-name-1
      (column-element [, column-element]...)
```

where *column-element* is defined as:

```
column-definition | table-constraint-definition
```

and *column-definition* is defined as:

```
unqualified-column-name data-type [DEFAULT default-value]
[column-constraint-definition [, column-constraint-definition]...]
      [COLLATE collation-name]
```

and *column-constraint-definition* is defined as:

```
CHECK (search-condition)
| NOT NULL
| PRIMARY KEY
| REFERENCES base-table-name-2 [(unqualified-column-name)]
| UNIQUE
```

and *default-value* is defined as:

```
literal | NULL
```

and *table-constraint-definition* is defined as:

```
UNIQUE (unqualified-column-name [, unqualified-column-name]...)
| PRIMARY KEY (unqualified-column-name [, unqualified-column-name]...)
| CHECK (search-condition)
| FOREIGN KEY referencing-columns
      REFERENCES base-table-name-2 [referenced-columns]
      [ON DELETE {NO ACTION | CASCADE | SET NULL | SET DEFAULT}]
```

and *referencing-columns* and *referenced-columns* are defined as:

```
(unqualified-column-name [, unqualified-column-name]...)
```

and where no two *column-definition*s begin with the same *unqualified-column-name*.

DESCRIPTION

The CREATE TABLE statement creates a base table named *base-table-name-1*. The current user becomes the owner of the base table. The owner has all privileges on the table, and can GRANT and REVOKE privileges on the table to other users.

The CREATE TABLE statement specifies each column of the created base table, giving its name in *unqualified-column-name* and its data type in the corresponding *data-type*. The column allows null values unless NOT NULL or PRIMARY KEY appears in that column definition.

In each *column-definition*, if *data-type* is a character-string data type, then the COLLATE clause can be used to specify a collation; if the COLLATE clause is omitted, the default collation associated with *data-type* is assumed.

It is implementation-defined which users can execute CREATE TABLE.

Table Constraints

A *table-constraint-definition* may contain UNIQUE, PRIMARY KEY, CHECK or FOREIGN KEY:

- If UNIQUE appears, then each row is constrained to contain a different value in the specified column (or specified columns taken together).
- PRIMARY KEY has the same meaning as UNIQUE, but also declares that the column names are the ones implicitly referenced in any FOREIGN KEY constraints (see below) that reference *base-table-name-1*. PRIMARY KEY may not occur more than once in a CREATE TABLE statement. Designating a column as the PRIMARY KEY also implicitly constrains the column to be NOT NULL.
- If CHECK appears, then each row is constrained to contain values that satisfy the *search-condition*.
- If FOREIGN KEY appears, then each row is constrained in that the *referencing-columns* must contain values found in *base-table-name-2*. The valid values are contained in the *referenced-columns* or, if this is omitted, in the primary key of *base-table-name-2*.

Each *unqualified-column-name* specified in UNIQUE, PRIMARY KEY or *referencing-columns* must identify a column of *base-table-name1*. In each case, if a list is specified, it must not identify the same column more than once.

Multiple unique (UNIQUE or PRIMARY KEY) clauses cannot apply to the same set of columns.

ON DELETE Actions

As described above, specifying FOREIGN KEY for one or more *referencing-columns* in the table being defined restricts their valid values to those existing in certain columns of a table *base-table-name-2*. If those values in *base-table-name-2* were deleted, all *referencing-columns* that contained the deleted values would immediately violate the FOREIGN KEY constraint.[28]

When creating a table with a FOREIGN KEY, the application can specify what happens in this case by using an ON DELETE clause. This specifies the effect of certain positioned DELETE or searched DELETE statements applied to *base-table-name-2*. ON DELETE is followed by exactly one of the following:

NO ACTION
: The positioned DELETE or searched DELETE fails with '**23**000' and makes no change to the database. (This was the only option in the previous issue of this specification, which did not specify the ON DELETE clause.)

28. The table being defined could also violate this constraint if the values in *base-table-name-2* were modified. The International Standard includes an ON UPDATE clause, which works in the same manner as the ON DELETE clause specified here, to cover this case. X/Open does not require implementations to support the ON UPDATE clause.

CASCADE
: All rows where any *referencing-columns* are affected are also deleted. If some columns of a deleted row are themselves the object of FOREIGN KEY references, the deletion of the row may cause additional rows to be deleted, possibly including rows in other tables.

SET NULL
: Affected *referencing-columns* are set to the null value.*

SET DEFAULT
: Each affected column of *referencing-columns* is set to its respective default value.*

If the CREATE TABLE statement contains a FOREIGN KEY clause that does not contain an ON DELETE clause, ON DELETE NO ACTION is assumed.

Column Constraints

A *column-constraint-definition* is a distinct syntactic form that achieves the same purposes as a *table-constraint-definition* but applies to a single column. This form may contain CHECK, NOT NULL, PRIMARY KEY, REFERENCES or UNIQUE.

- The NOT NULL constraint constrains the column to not contain null values. NOT NULL may not occur more than once in a *column-definition*.
- The PRIMARY KEY and UNIQUE constraints have the same effect as described above for table constraints. In the case of column constraints, the application cannot specify both PRIMARY KEY and UNIQUE.[29]
- The REFERENCES column constraint is equivalent to a FOREIGN KEY table constraint (see above). It constrains each row to contain only values contained in a specified base table.
- If CHECK appears, then each row is constrained to contain a value that satisfies the *search-condition*. The *search-condition* may only contain literals and references to the column being defined.

The DEFAULT clause specifies a value that is given to the column if an INSERT statement does not furnish a value. All columns have a default value; if the application does not use the DEFAULT clause, the column's default value is the null value.

The specified or implied default value can be NULL, even if the column is constrained by NOT NULL, but it is then a constraint violation to insert a row without overriding the default value.

Any specified default value is subject to the following rules:

- If the data type of the column is character string, *default-value* must be a character string literal or *pseudo-literal* and its length must not exceed the column's length or maximum length; blanks are appended to a literal if necessary to make it the length of a fixed-length column.

---

* Doing so may itself violate an integrity constraint, such as a NOT NULL or CHECK constraint. In this case, the positioned DELETE or searched DELETE fails as described for the NO ACTION case.

29. **Implementation Note:** UNIQUE columns may specify NOT NULL, and PRIMARY KEY columns imply NOT NULL. In X/Open SQL, a UNIQUE index (which is functionally similar to a unique constraint) may be created including columns which are defined to allow null values. This means X/Open-compliant implementations cannot implement unique constraints using UNIQUE indexes unless they enforce the NOT NULL constraint separately.

If *default-value* is a *pseudo-literal* such as CURRENT_TIMESTAMP or CURRENT_USER, the column's default value reflects the value the *pseudo-literal* has at the time of the particular insertion, not the value it had at the time of the CREATE TABLE statement.

- If the data type of the column is exact numeric, *default-value* must be a numeric literal for which there is a valid representation in the column that does not lose any significant digits.
- If the data type of the column is approximate numeric, *default-value* must be a numeric literal.

In the CHECK clause, the *search-condition* may not contain a set function reference, a subquery, CURRENT_DATE, CURRENT_TIME or CURRENT_TIMESTAMP.

FOREIGN KEY lets one table reference itself or another table. The REFERENCES keyword can appear in a *column-constraint-definition* to apply to the column being defined, or in a *table-constraint-definition* to apply to one column or to a set of columns taken together. Referencing and referenced columns must be equal in number and in data type. The sequence of *referenced-columns* must be the same as that in a UNIQUE or PRIMARY KEY list in the CREATE TABLE statement that defines *base-table-name-2*.

DIAGNOSTICS

The *base-table-name-1* must not identify an existing base table or viewed table ('**42**S01').

Any column specified in a constraint (referencing a table other than the one being created) must already exist ('**42**S22'). (See the exception in **Object References** on page 100.)

The current user must have the REFERENCES privilege on all *referenced-columns* ('**42**000').

The **Success with warning** outcome occurs if a default value was specified whose length exceeds 254 characters ('**01**00B'). The warning indicates that the application cannot retrieve the default value by querying the COLUMNS system view (see *COLUMNS* on page 160).

5.3.8 CREATE TRANSLATION

NAME

CREATE TRANSLATION — Create a translation.

SYNOPSIS

```
CREATE TRANSLATION new-translation
      FOR source-char-set TO target-char-set FROM
      { IDENTITY
      | translation-name
      | EXTERNAL(character-string-literal)
```

DESCRIPTION

The CREATE TRANSLATION statement creates a translation named *new-translation*, for use in the TRANSLATE function specified in **TRANSLATE** on page 62. A translation takes a character string in *source-char-set* and produces a string in *target-char-set*.

The CREATE TRANSLATION statement can specify the translation in one of the following ways:

- The keyword IDENTITY means that the translation is null; the translation being defined translates every character of the source string into the corresponding character from *target-char-set*. Although the characters are unchanged by the IDENTITY translation, their encodings may be different in the target character string.

- An existing *translation-name* can be specified. The effect of a CREATE TRANSLATION statement that uses this form is basically to define a new name for the existing translation.
- The syntax EXTERNAL(*character-string-literal*) specifies an implementation-defined EXTERNAL translation. The set of valid values for the literal is implementation-defined.

  An implementation can define a certain translation as a built-in *translation-name*. This means it can be used in embedded SQL programs. By contrast, an implementation can also define a certain translation as a valid EXTERNAL translation. This means it must be assigned a *translation-name* using the CREATE TRANSLATION statement before it can be used.

DIAGNOSTICS

The *new-translation* must not identify an existing translation ('**42**S81').

5.3.9 CREATE VIEW

NAME

CREATE VIEW — Create a viewed table.

SYNOPSIS

```
CREATE VIEW viewed-table-name
      [(unqualified-column-name [,unqualified-column-name]...)]
      AS query-expression
      [WITH CHECK OPTION]
```

where a given *unqualified-column-name* appears only once.

DESCRIPTION

The CREATE VIEW statement creates a viewed table *viewed-table-name* that is the result of evaluating *query-expression*.

The *query-expression* defines a derived table, as specified in Section 3.11.3 on page 75. The resulting viewed table is updatable if this derived table is updatable. The data types of the viewed table's columns are the same as those of the corresponding columns of the derived table. The viewed table's column names are the names of the corresponding columns of the derived table unless the CREATE VIEW statement contains *unqualified-column-name*s. (If the derived table contains any columns with undefined or duplicate names, then the CREATE VIEW statement must contain *unqualified-column-name*s.)

The current user becomes the owner of the viewed table. The owner has INSERT or DELETE privileges on the viewed table if the viewed table is updatable and if the owner has that privilege on the table from which the viewed table is derived. The owner has UPDATE privilege on a given column of the viewed table if the viewed table is updatable and if the owner has that privilege on the corresponding column of the table from which the viewed table is derived. Otherwise the owner only has SELECT privilege on the viewed table.

If any *query-specification* in *query-expression* contains a GROUP BY clause, *viewed-table-name* identifies a grouped view.

No *query-specification* in *query-expression* may reference any host variables, contain any dynamic parameters, or reference the viewed table being created.

WITH CHECK OPTION

The search conditions in the *query-specification* in the CREATE VIEW statement, combined with those in any underlying CREATE VIEW statement, are called the **view criteria**.

If WITH CHECK OPTION is specified, then any insert or update performed on the viewed table, or on any table derived from the viewed table, fails (setting SQLSTATE to '**44**000') if the resulting row violates the view criteria. That is, the insert or update fails if the effect of the change would be that the resulting row is no longer present in *viewed-table-name*.

If WITH CHECK OPTION is not specified, then *query-expression* determines the initial definition of the viewed table, but does not regulate the changes that can be made to the viewed table. In this case, the application may insert or update rows in the viewed table that it is subsequently unable to retrieve through that viewed table.

DIAGNOSTICS

The *viewed-table-name* must not identify an existing base table or viewed table ('**42**S01').

If CREATE VIEW contains *unqualified-column-name*s, the number of names specified must be the same as the degree of the derived table defined by *query-expression* ('**21**S02').

The current user must have SELECT privilege on every table referenced by *query-expression* ('**42**000'). If SELECT privilege is revoked at any time on any such table, the viewed table is dropped automatically.

5.3.10 DROP CHARACTER SET

NAME

DROP CHARACTER SET — Destroy a character set.

SYNOPSIS

```
DROP CHARACTER SET character-set-name
```

DESCRIPTION

The DROP CHARACTER SET statement removes the character set *character-set-name* from the database.

The statement is invalid if there is any of the following dependencies on the character set being dropped:

- If any table column of a character-string data type is defined to use *character-set-name*.
- If any named collation is defined for *character-set-name*.
- If any translation uses *character-set-name* as either its source character set or its target character set.
- If the SQL NAMES clause in an SQL declare section specifies *character-set-name* as the default character set for user-defined names and character-string literals in the compilation unit.

(A CASCADE option, in which the implementation would resolve any of the above dependencies, is not defined.)

DIAGNOSTICS

The *character-set-name* must identify a character set that already exists ('**42**S52').

5.3.11 DROP COLLATION

NAME

DROP COLLATION — Destroy a collation.

SYNOPSIS

```
DROP COLLATION collation-name
```

DESCRIPTION

The DROP COLLATION statement removes the collation *collation-name* from the database.

The statement is invalid if there is any of the following dependencies on the collation being dropped:

- If the collation is specified as the default collation for any character set.
- If the collation is specified as the collation to be used for any string expression in the database, such as in a constraint.
- If the collation is used in the definition of another collation.

(A CASCADE option, in which the implementation would resolve any of the above dependencies, is not defined.)

DIAGNOSTICS

The *collation-name* must identify a collation that already exists ('**42**S62').

5.3.12 DROP INDEX

NAME

DROP INDEX — Destroy an index.

SYNOPSIS

```
DROP INDEX index-name
```

DESCRIPTION

The DROP INDEX statement removes the index *index-name* from the database.

DIAGNOSTICS

The *index-name* must identify an index that already exists ('**42**S12').

5.3.13 DROP SCHEMA

NAME

DROP SCHEMA — Destroy a schema.

SYNOPSIS

```
DROP SCHEMA [object-qualifier] {CASCADE | RESTRICT}
```

DESCRIPTION

The DROP SCHEMA statement destroys the schema specified by *object-qualifier*.

The application must use the keyword CASCADE or RESTRICT. Doing so specifies the effect if the schema contains any object (table, index, or privilege):

CASCADE	All such objects are also destroyed.
RESTRICT	The statement is invalid.

DIAGNOSTICS

The *object-qualifier* must identify a schema that already exists ('**42**S32').

5.3.14 DROP TABLE

NAME

DROP TABLE — Destroy a base table.

SYNOPSIS

`DROP TABLE` *`base-table-name`* `{CASCADE | RESTRICT}`

DESCRIPTION

The DROP TABLE statement removes *base-table-name* from the database. It also drops any indexes based on that base table, and revokes any privileges that were granted on that base table.

The application must use the keyword CASCADE or RESTRICT. Doing so specifies the effect if FOREIGN KEY or CHECK constraints reference *base-table-name* or if viewed tables are based on *base-table-name*:

CASCADE	All such constraints and viewed tables are also dropped.
RESTRICT	The statement is invalid.

DIAGNOSTICS

The *base-table-name* must identify a table that already exists ('**42**S02').

5.3.15 DROP TRANSLATION

NAME

DROP TRANSLATION — Destroy a translation.

SYNOPSIS

`DROP TRANSLATION` *`translation-name`*

DESCRIPTION

The DROP TRANSLATION statement removes the translation *translation-name* from the database.

The statement is invalid if there is any of the following dependencies on the translation being dropped:

- If the translation is used in the definition of a collation.
- If the translation is used within any string expression in the database, such as in a constraint.

(A CASCADE option, in which the implementation would resolve any of the above dependencies, is not defined.)

DIAGNOSTICS

The *translation-name* must identify a translation that already exists ('**42**S82').

5.3.16 DROP VIEW

NAME

DROP VIEW — Destroy a view.

SYNOPSIS

`DROP VIEW` *`viewed-table-name`* `{CASCADE | RESTRICT}`

DESCRIPTION

The DROP VIEW statement removes *viewed-table-name* from the database. It also revokes any privileges granted on the viewed table.

The application must use the keyword CASCADE or RESTRICT. Doing so specifies the effect if other viewed tables are based on *viewed-table-name*:

CASCADE	All such other viewed tables are also dropped.
RESTRICT	The statement is invalid.

Revoking necessary SELECT privileges from the owner of a viewed table may implicitly drop a viewed table.

DIAGNOSTICS

The *viewed-table-name* must identify a table that already exists ('**42**S02').

5.3.17 GRANT

NAME

GRANT — Grant privileges on a table.

SYNOPSIS

```
GRANT {ALL PRIVILEGES | privilege [, privilege]...}
      ON privilege-object
      TO {PUBLIC | user-name [, user-name]...}
      [WITH GRANT OPTION]
```

where *privilege* is one of the following:

```
DELETE
INSERT
REFERENCES [(unqualified-column-name [, unqualified-column-name]...)]
SELECT
UPDATE [(unqualified-column-name [, unqualified-column-name]...)]
USAGE
```

and where *privilege-object* is defined as:

```
[TABLE] table-name
| CHARACTER SET character-set-name
| COLLATION collation-name
| TRANSLATION translation-name
```

and where a given *privilege* or *user-name* appears only once.

DESCRIPTION

The GRANT statement grants privileges on *table-name* to one or more users, provided the current user has such grantable privileges.

The GRANT statement grants one or more *privileges*:

DELETE	Grants the right to delete rows. (May be used only when *privilege-object* is a table.)
INSERT	Grants the right to insert rows. (May be used only when *privilege-object* is a table.)
REFERENCES	Grants the right to reference the specified columns from a FOREIGN KEY. Specifying REFERENCES without *unqualified-column-names* grants the right on all columns, including any column subsequently added to *privilege-object*. (May be used only when *privilege-object* is a table.)
SELECT	Grants the right to retrieve values. (May be used only when *privilege-object* is a table.)

UPDATE	Grants the right to update the specified columns. Specifying UPDATE without *unqualified-column-name*s grants the right on all columns, including any column subsequently added to *privilege-object*. (May be used only when *privilege-object* is a table.)
USAGE	Grants the right to use the object. (May be used only when *privilege-object* is a character set, collation or translation.)

Specifying GRANT ALL PRIVILEGES is equivalent to specifying all the grantable privileges that the current user has on *privilege-object*.

The application specifies PUBLIC to grant the specified privilege to all present and future users, or identifies specific users by *user-name*s.

The application specifies WITH GRANT OPTION to allow any users affected by this GRANT statement to further grant the *privilege*s to other users.

DIAGNOSTICS

The current user must hold at least one privilege on *privilege-object* ('**42**000').

Any specified column must already exist ('**42**S22'). (See the exception in **Object References** on page 100.)

A separate **Success with warning** outcome occurs ('**01**007') in the following cases:

- Any combination of explicitly specified privilege and recipient (including PUBLIC) is not grantable.
- ALL PRIVILEGES is specified and, for any specified recipient (including PUBLIC), the current user has no grantable privileges on *privilege-object*.

5.3.18 REVOKE

NAME

REVOKE — Revoke privileges on a table.

SYNOPSIS

```
REVOKE {ALL PRIVILEGES | privilege [, privilege]...}
      ON privilege-object
      FROM {PUBLIC | user-name [, user-name]...}
      {CASCADE | RESTRICT}
```

where *privilege* and *privilege-object* are the same as for the GRANT statement (see Section 5.3.17 on page 115) and where a given *privilege* or *user-name* appears only once.

DESCRIPTION

The REVOKE statement revokes privileges on *privilege-object* that are held by one or more specified users with the current user as grantor. Each *privilege* may be one of the following:

DELETE	Revokes the right to delete rows. (May be used only when *privilege-object* is a table.)
INSERT	Revokes the right to insert rows. (May be used only when *privilege-object* is a table.)
REFERENCES	Revokes the right to reference the specified columns from a FOREIGN KEY. (May be used only when *privilege-object* is a table.)
SELECT	Revokes the right to retrieve values. (May be used only when *privilege-object* is a table.)

UPDATE	Revokes the right to update the specified columns. (May be used only when *privilege-object* is a table.)
USAGE	Revokes the right to use the object. (May be used only when *privilege-object* is a character set, collation or translation.)

Specifying UPDATE or REFERENCES with *unqualified-column-name*s revokes the respective privilege on the specified columns. Applications can grant a privilege on a table at large and then revoke it on specific columns, indicating exceptions to the general rule. In this case the user retains privileges on all remaining columns of the table, and on all columns subsequently added to the table.

Specifying UPDATE or REFERENCES without *unqualified-column-name*s revokes the user's privilege on all columns in the table, whether granted on individual columns or on the table at large. It also revokes any privilege the user has on columns subsequently added to the table, which may have been granted by a previous GRANT statement that did not specify columns.

Specifying REVOKE ALL PRIVILEGES is equivalent to specifying all the grantable privileges that the current user has on *privilege-object*.

The FROM clause either identifies one or more users by their *user-name*s, or specifies PUBLIC to revoke privileges formerly extended to all users using GRANT ... PUBLIC.

The application must use the keyword CASCADE or RESTRICT. Doing so specifies the effect if there are any of the dependencies on the privilege being revoked that are defined below. If RESTRICT is specified, the statement is invalid. If CASCADE is specified, the following effects occur:

- *If another privilege is based on the privilege being revoked:* Any such privilege is also revoked.
- *Revoking a SELECT privilege that a view's owner needed in order to create the view:* Any such view is dropped.
- *Revoking a REFERENCES privilege on a column to which the user has made FOREIGN KEY references from another table:* Any such FOREIGN KEY reference is dropped from the other table.
- *Revoking a USAGE privilege on a character set, collation or translation from the owner of a table that references it:* Any column that contains such a reference is dropped.
- *Revoking a USAGE privilege on a collation from the owner of a table that specifies the collation as the collation for any column:* The definition of any such column is changed to instead specify the default collation for the character set of the column.

DIAGNOSTICS

The current user must hold at least one privilege on *privilege-object* ('**42**000').

To revoke the INSERT, DELETE or UPDATE privilege on a viewed table, the table must be updatable ('**42**000').

A separate **Success with warning** outcome ('**01**006') occurs in the following cases:

- For any combination of explicitly specified privilege and recipient (including PUBLIC), the recipient does not currently hold such a privilege that the current user granted to the recipient.
- ALL PRIVILEGES is specified and any specified recipient (including PUBLIC) currently holds no privileges on *privilege-object* that the current user granted to the recipient.

5.4 Data Manipulation Statements

A data manipulation statement operates on the data contents of the database or controls the state of a cursor. (Cursors are described in Section 4.4 on page 87.)

INSERT and UPDATE statements assign values to columns in tables. These assignments follow rules specified in Section 3.5 on page 51.

Applications should not mix data definition statements and data manipulation statements within transactions. See Section 7.4 on page 183.

5.4.1 CLOSE

NAME

CLOSE — Close a cursor.

SYNOPSIS

`CLOSE` *`cursor-name`*

DESCRIPTION

CLOSE closes the cursor *cursor-name*, so that the result set associated with it is no longer accessible.

DIAGNOSTICS

The cursor must be open ('**24**000').

5.4.2 Positioned DELETE

NAME

Positioned DELETE — Delete a row of a table.

SYNOPSIS

`DELETE FROM` *`table-name`* `WHERE CURRENT OF` *`cursor-name`*

DESCRIPTION

This form of DELETE deletes the row from which the current row of the result set of *cursor-name* is derived. If *table-name* is a viewed table, DELETE deletes the corresponding row of the base table from which the viewed table is derived.

The cursor must be updatable.

The *table-name* must be the (single) table referenced by the FROM clause of the *query-specification* that defines the result set of the cursor.

DIAGNOSTICS

The current user must have DELETE privilege on *table-name* ('**42**000').

The cursor must be open and positioned on a row in its result set ('**24**000').

5.4.3 Searched DELETE

NAME

Searched DELETE — Delete rows of a table that satisfy a *search-condition*.

SYNOPSIS

`DELETE FROM` *`table-name`* `[WHERE` *`search-condition`*`]`

DESCRIPTION

This form of DELETE deletes any rows from *table-name* that satisfy *search-condition*. If *table-name* is a viewed table, DELETE deletes the corresponding rows of the base table from which it is derived. If no row satisfies *search-condition*, DELETE deletes zero rows. If *search-condition* is omitted, DELETE deletes all the rows in the table.

table-name must be updatable and not referenced (directly or indirectly) by a FROM clause of any *sub-query* contained in the *search-condition*.

The scope of the *table-name* is the entire searched DELETE statement.

If *search-condition* is specified, then DELETE applies it to each row of *table-name* and deletes all rows for which the *search-condition* is true.

Each *sub-query* in the *search-condition* is effectively executed for each row of *table-name*. The results are used in the evaluation of the *search-condition* for that row.

If any executed *sub-query* contains a reference to a column of *table-name*, the reference is to the value of that column in the given row. (See Section 3.11.5 on page 76.)

DIAGNOSTICS

The current user must have DELETE privilege on *table-name* and SELECT privilege on every table referenced by *search-condition* ('**42**000').

The **No data** outcome occurs if *search-condition* is specified, but no row satisfies it ('**02**000'). In this case, DELETE deletes 0 rows.

5.4.4 FETCH

NAME

FETCH — Advance the cursor to the next row of a result set and fetch that row.

SYNOPSIS

```
FETCH [FROM] cursor-name
      INTO host-variable-reference [, host-variable-reference]...
```

DESCRIPTION

FETCH advances *cursor-name* to the next row of its result set and assigns column values from that row to the host variables. The number of host variables must match the number of columns in the result set. The cursor advances to the new row even if errors occur during the evaluation of result column values or during the assignment of these values to host variables.

DIAGNOSTICS

The cursor must be open ('**24**000').

The **No data** outcome occurs if the result set is empty, or the cursor is positioned on or after the last row ('**02**000'). No database values are assigned to the host variables.

5.4.5 INSERT

NAME

INSERT — Insert rows into a table.

SYNOPSIS

```
INSERT INTO table-name
      {insert-source | DEFAULT VALUES }
```

where *insert-source* is defined as:

```
[(unqualified-column-name [,unqualified-column-name]...)]
      {query-specification
      | VALUES row-value-constructor}
```

DESCRIPTION

INSERT inserts row values into *table-name*. (If *table-name* is a viewed table, then INSERT inserts into the base table from which *table-name* is derived.)

Depending on the form used, INSERT inserts one row containing one of the following:

- values, for specified columns or for all columns, derived from *query-specification*
- values, for specified columns or for all columns, explicitly specified by a *row-value-constructor* (see Section 3.12 on page 78)
- values for all columns that are the default values for that column in *table-name*.

In the first two cases, the *i*th value specified is assigned to the *i*th column of the table or to the *i*th *unqualified-column-name*.

Omitting the *unqualified-column-name*s is equivalent to specifying all the columns of *table-name*, in ascending order of their position in the table.

If *unqualified-column-name*s are present, each must identify a column of *table-name*, and there can be no duplicates. There must be the same number of *unqualified-column-name*s as the number of values the INSERT statement produces. If INSERT omits some columns of *table-name*, then these columns of the new row are set to their default values. If *table-name* is a viewed table, this is also the case for any column of its base table not included in the viewed table.

The table *table-name* must be updatable and not referenced by a FROM clause of the *query-specification* or of any *sub-query* contained in the *query-specification*.

DIAGNOSTICS

The number of columns specified or implied must match the degree of the derived table (in the *query-specification* form) or the number of *expression*s (in the VALUES form) ('**21**S01').

The current user must have INSERT privilege on *table-name* and SELECT privilege on every table in the FROM clause of *query-specification* ('**42**000').

The **No data** outcome occurs if INSERT included a *query-specification* that produced no rows ('**02**000').

The row to be inserted must not violate any view criteria (see Section 5.3.9 on page 111) ('**44**000').

5.4.6 OPEN

NAME

OPEN — Open a cursor.

SYNOPSIS

OPEN *cursor-name*

DESCRIPTION

OPEN executes the cursor specification for *cursor-name*, identifying a multi-set of rows, which becomes the result set for the cursor. Then OPEN opens *cursor-name* and positions it before the first row in the result set.

Any host variables in the cursor specification affect the cursor specification based on their values at the time OPEN is executed. If their values then change, it does not affect the result set.

DIAGNOSTICS

The cursor must not already be open ('**24**000').

The current user must have SELECT privilege on every table referenced by the cursor specification for *cursor-name* ('**42**000').

The process of opening a cursor may produce expression exceptions, as described in Section 3.9 on page 57. These exceptions may be reported by the OPEN statement, or may be deferred and reported by a subsequent FETCH statement.

5.4.7 SELECT INTO

NAME

SELECT INTO — Assign the values from the specified table to host variables.

SYNOPSIS

```
SELECT [ALL | DISTINCT] select-list
      INTO host-variable-reference [, host-variable-reference]...
      FROM table-reference [, table-reference]...
      [WHERE search-condition]
```

DESCRIPTION

The SELECT INTO statement specifies a one-row table and assigns its values to host variables. The number of host variables must match the number of columns in the result set.

The definitions of the SELECT, FROM and WHERE clauses and the rules regarding multiple use of the keyword DISTINCT are the same as those for a *query-specification* (see Section 3.11.1 on page 71).

If the resulting table contains a single row, values in the row of this table are assigned to their corresponding *host-variable-reference*s: the *i*th *host-variable-reference* is assigned the value of the *i*th column of the result set.

DIAGNOSTICS

The current user must have SELECT privilege on all *table-reference*s ('**42**000').

The SELECT INTO statement must not produce a result with more than one row ('**21**000'). It is undefined whether database values are assigned to the host variables.

The **No data** outcome occurs if SELECT INTO produces a result with 0 rows ('**02**000'). No database values are assigned to the host variables.

5.4.8 Positioned UPDATE

NAME

Positioned UPDATE — Update a row of a table.

SYNOPSIS

```
UPDATE table-name
      SET unqualified-column-name = {expression | NULL | DEFAULT}
      [, unqualified-column-name = {expression | NULL | DEFAULT}]...
      WHERE CURRENT OF cursor-name
```

DESCRIPTION

UPDATE updates one row of *table-name*. The updated row is the row from which the current row of the active set of *cursor-name* is derived. If *table-name* is a viewed table, UPDATE updates the corresponding row of the base table from which it is derived.

Each *unqualified-column-name* identifies a column to be updated (an **update column**) and must identify a column of *table-name* (the **update table**). The same column must not be identified more than once.

The cursor must be updatable. The columns identified by the cursor's implicit or explicit FOR UPDATE clause must include the update columns.

The update table must be the (single) table referenced by the FROM clause of the *query-specification* that defines the result set of the cursor. The scope of the *table-name* is the entire positioned UPDATE statement.

The syntax to the right of the equal sign represents the update value of the corresponding update column. That is, for each update column, the value of that column in the row to be updated is replaced by the value of the corresponding *expression*, or by null value if NULL is specified, or by the column's default value if DEFAULT is specified.

If *expression* is specified, it must not include a *set-function-reference* and any column it references must be a column of the update table. The value specified by such a referenced column is the value of the identified column in the row to be updated before applying any updates.

The cursor position is unchanged even if errors occur during the evaluation of update values or the assignment of update values to update columns.

DIAGNOSTICS

The current user must have UPDATE privilege on each update column in *table-name* and SELECT privilege on *table-name* if any of its columns is referenced by an update value ('**42**000').

The cursor must be open and positioned on a row in its result set ('**24**000').

The row is not updated if the resulting row would violate any view criteria (see Section 5.3.9 on page 111) ('**44**000').

5.4.9 Searched UPDATE

NAME

Searched UPDATE — Update values of columns in a table.

SYNOPSIS

```
UPDATE table-name
      SET unqualified-column-name = {expression | NULL | DEFAULT}
      [, unqualified-column-name = {expression | NULL | DEFAULT}]...
      [WHERE search-condition]
```

DESCRIPTION

UPDATE updates zero or more rows of *table-name*. If *table-name* is a viewed table, UPDATE applies the updates to the corresponding rows of the base table from which it is derived.

Each *unqualified-column-name* identifies a column to be updated (an **update column**) and must identify a column of *table-name* (the **update table**). The same column must not be identified more than once.

The update table must be updatable and must not be referenced (directly or indirectly) by the FROM clause of any *sub-query* contained in the *search-condition*.

The scope of the *table-name* is the entire searched UPDATE statement.

The syntax to the right of the equal sign represents the update value of the corresponding update column. That is, for each update column, the value of that column in the row to be updated is replaced by the value of the corresponding *expression*, or by null value if NULL is specified, or by the column's default value if DEFAULT is specified.

If *expression* is specified, it must not include a *set-function-reference* and any column it references must be a column of the update table. The value specified by such a referenced column is the value of the identified column in the row to be updated before any values in that row are updated.

If *search-condition* is not specified, all rows of the update table are updated.

If *search-condition* is specified, it is applied to each row of the update table and all rows for which the result of the *search-condition* is true are updated.

Each *sub-query* in the *search-condition* is effectively executed for each row of the update table and the results used in the evaluation of the *search-condition* for that row.

If any executed *sub-query* contains a reference to a column of the update table, the reference is to the value of that column in the given row. (See Section 3.11.5 on page 76.)

DIAGNOSTICS

The current user must have UPDATE privilege on each update column in *table-name*, SELECT privilege on *table-name* if any of its columns is referenced by an update value, and SELECT privilege on every table referenced by *search-condition* ('**42**000').

The **No data** outcome occurs if no row satisfies the *search-condition* ('**02**000').

The row is not updated if the resulting row would violate any view criteria (see Section 5.3.9 on page 111) ('**44**000').

5.5 Dynamic SQL Statements

Dynamic SQL statements provide a way to execute SQL statements whose specifics may not be known at compile-time. For example, the application assembles the text of an SQL statement as a character string, uses the PREPARE statement to prepare it for execution, and then specifies the prepared statement in one or more EXECUTE statements. Dynamic SQL enables interactive applications, such as fourth-generation languages, where the exact SQL statement text or parameter values may be known only after computation or user interaction.

Dynamic SQL can use cursors, but the application must use a special form of the DECLARE CURSOR statement (see Section 4.4.2 on page 89) to declare a cursor for subsequent association with a prepared cursor specification.

Three statements already discussed can also be used on dynamic cursors:

- the CLOSE statement (see Section 5.4.1 on page 118)
- the positioned DELETE statement (see Section 5.4.2 on page 118)
- the positioned UPDATE statement (see Section 5.4.8 on page 121).

Their syntax and use is unchanged in dynamic SQL except that *cursor-name* identifies a dynamic cursor. When used in dynamic SQL, they are known respectively as the dynamic CLOSE, dynamic positioned DELETE, and dynamic positioned UPDATE.

5.5.1 SQL Descriptor Areas

An SQL descriptor area is a storage area for descriptive information pertaining to dynamic SQL statements. Embedded SQL statements allocate and maintain these areas. An ALLOCATE DESCRIPTOR statement allocates an SQL descriptor area. A subsequent DEALLOCATE DESCRIPTOR statement deallocates an SQL descriptor area. The following dynamic SQL statements use the SQL descriptor area:

- The DESCRIBE statement requires an SQL descriptor area in order to hold information about the resulting columns in a *cursor-specification*.
- A dynamic FETCH statement can use an SQL descriptor area to hold values of a row of a *cursor-specification*.

In these two cases, the SQL descriptor area provides output from the database system. The application uses GET DESCRIPTOR to move information from the SQL descriptor area into embedded host variables.

- An EXECUTE or a dynamic OPEN statement can use an SQL descriptor area to associate values with the parameters in the prepared statement.

In these cases, the SQL descriptor area provides input to the database system. The application uses SET DESCRIPTOR to move information from embedded host variables into the SQL descriptor area.

In all cases except the DESCRIBE statement, use of the SQL descriptor area is optional, and occurs only if the application uses that variant of the USING clause (see Section 5.5.2 on page 129).

Fields of the SQL Descriptor Area

An SQL descriptor area consists of a COUNT field and zero or more item descriptor areas.

The COUNT field specifies how many item descriptor areas contain data. In general, whichever component (application or database system) sets the SQL descriptor area must also set the COUNT field to show how many item descriptor areas are significant.

When the application allocates an SQL descriptor, it can specify how many descriptor areas to reserve room for.

Fields of the Item Descriptor Areas

Depending on the situation, each item descriptor area may describe a selected column, an input value, or an output value. In every case but DESCRIBE, the item descriptor area contains a value, and associated fields describe the value's attributes, such as its data type. In DESCRIBE, the item descriptor area describes a resulting column of a *cursor-specification* or a dynamic parameter. The item descriptor area does not contain data, but the associated fields specify a data type and type attributes.

The GET DESCRIPTOR and SET DESCRIPTOR statements can refer to the individual fields of an item descriptor area by name:

TYPE An exact numeric value with scale 0 that specifies the SQL data type (see Section 4.1.3 on page 82). For X/Open-defined named data types, the value of TYPE is one of the following:

Code	Named Data Type
1	CHARACTER(*n*)
2	NUMERIC
3	DECIMAL
4	INTEGER
5	SMALLINT
6	FLOAT
7	REAL
8	DOUBLE PRECISION
9	DATETIME
10	INTERVAL
12	CHARACTER VARYING

Other positive numbers are reserved for future definition. Implementors should use negative values for implementation-defined data types.

For implementation-defined values of TYPE, the valid values for, and meanings of, all other fields of the item descriptor area are implementation-defined.

LENGTH An exact numeric value with scale 0, whose value is either the maximum or actual character length of a character string data type depending on the circumstances. Its value always excludes any null byte that ends the character string.

PRECISION An exact numeric value with scale 0. For a numeric data type, its value is an applicable precision. For the data types INTERVAL SECOND, INTERVAL DAY TO SECOND, INTERVAL HOUR TO SECOND, INTERVAL MINUTE TO SECOND, TIME or TIMESTAMP, its value is an applicable precision of the fractional seconds component.

SCALE An exact numeric value with scale 0, whose value is an applicable scale for a numeric data type.

DATETIME_INTERVAL_CODE

An exact numeric value with scale 0 that specifies the date / time subtype. If TYPE is 9 (DATETIME), then valid values for DATETIME_INTERVAL_CODE are as follows:

Code	**Named Date/Time Subtype**
1	DATE
2	TIME
3	TIMESTAMP

If TYPE is 10 (INTERVAL), then valid values for DATETIME_INTERVAL_CODE are as follows:

Code	**Named Date/Time Subtype**
1	YEAR
2	MONTH
3	DAY
4	HOUR
5	MINUTE
6	SECOND
7	YEAR TO MONTH
8	DAY TO HOUR
9	DAY TO MINUTE
10	DAY TO SECOND
11	HOUR TO MINUTE
12	HOUR TO SECOND
13	MINUTE TO SECOND

DATETIME_INTERVAL_PRECISION

An exact numeric value with scale 0 that specifies the leading field precision for certain interval data types (for which TYPE is 10). This is the YEAR field for the data types INTERVAL YEAR and INTERVAL YEAR TO MONTH; or the DAY field for the data types INTERVAL DAY, INTERVAL DAY TO HOUR, INTERVAL DAY TO MINUTE and INTERVAL DAY TO SECOND.

NULLABLE An exact numeric value with scale 0, used by the DESCRIBE statement to indicate whether or not a resulting column can contain the null value.

INDICATOR An exact numeric with scale 0, used as an indicator variable for an input value (OPEN and EXECUTE) or an output value (FETCH).

DATA If the value of the INDICATOR field is 0, this field contains an input value or an output value whose data type is specified by the TYPE field and whose attributes are specified by the fields applicable for that data type.

NAME A character string field in which the DESCRIBE statement returns the name of the column. When retrieving this field, the application should provide sufficient characters to hold a user-defined name of the maximum allowed length.

UNNAMED An exact numeric value with scale 0. The DESCRIBE statement sets it to 0 if the value returned in NAME is an actual name, or to 1 if NAME contains an implementation-generated column name that X/Open does not define (for example, where the subject column represents the UNION of two columns with different names).

RETURNED_LENGTH
: An exact numeric value with scale 0, used by the FETCH statement to return the actual character length of a (non-null) VARCHAR output value, or of a (non-null) VARCHAR result column value if the data type of the output value is not VARCHAR.

Field Usage in DESCRIBE

The value of the TYPE field determines which other fields of the item descriptor area are used by the DESCRIBE statement, as shown in the following table. The **Out** legend means the DESCRIBE statement sets the field. A blank entry means that the value of the field is undefined as a result of the DESCRIBE statement.

	TYPE				
	Character string	**Exact numeric**	**Approx. numeric**	**Date/ time**	**Interval**
PRECISION		Out	Out	Out	Out
SCALE		Out			
LENGTH	Out [1]			Out	
NULLABLE	Out	Out	Out	Out	Out
INDICATOR					
DATA					
NAME	Out	Out	Out	Out	Out
UNNAMED	Out	Out	Out	Out	Out
RETURNED_LENGTH					
DATETIME_INTER-VAL_CODE				Out	Out
DATETIME_INTER-VAL_PRECISION					Out

[1] The maximum length is returned.

Field Usage in OPEN, EXECUTE and FETCH

The value of the TYPE field determines which other fields of the item descriptor area are used by the OPEN, EXECUTE and FETCH statements, as shown in the following table. The **In** legend means the application must set the field before calling OPEN, EXECUTE, or FETCH. The **Out** legend means the FETCH statement sets the field. The **In-Out** legend means the application sets the field before calling OPEN or EXECUTE, and the FETCH statement sets the field.

	TYPE					
	Character string	**INTEGER, SMALLINT REAL, DBL. PREC.**	**NUMERIC, DECIMAL**	**FLOAT**	**Date/ time**	**Interval**
PRECISION			In	In	In	In
SCALE			In			
LENGTH	In [1]				In	
NULLABLE						
INDICATOR	In-Out	In-Out	In-Out	In-Out	In-Out	In-Out
DATA	In-Out	In-Out	In-Out	In-Out	In-Out	In-Out
NAME						
UNNAMED						
RETURNED_LENGTH	Out					
DATETIME_INTER-VAL_CODE					In-Out	In
DATETIME_INTER-VAL_PRECISION						In

[1] Maximum length for FETCH, actual length for OPEN / EXECUTE.

At the time GET DESCRIPTOR reads the DATA field, or SET DESCRIPTOR writes it, the value of TYPE and the applicable values of LENGTH, PRECISION and SCALE, must be valid (for example, PRECISION may not be negative) and consistent. The data type of the host variable receiving or supplying the DATA field must match the TYPE field and the applicable values of LENGTH, PRECISION, and SCALE according to the following table:

TYPE field	**Required type of host variable**
CHARACTER	CHARACTER(*n*), where *n* is the value of LENGTH
DATETIME	CHARACTER(*n*), where *n* is the value of LENGTH
DECIMAL or NUMERIC	DECIMAL(*p*, *s*) or NUMERIC(*p*, *s*) where *p* is the value of PRECISION and *s* is the value of SCALE
DOUBLE PRECISION	DOUBLE PRECISION
INTEGER	INTEGER
INTERVAL	SHORT, INTEGER, or FLOAT according to type (see the table in Section 4.1.3 on page 82)
REAL	REAL
SMALLINT	SMALLINT

5.5.2 USING and INTO Clauses

NAME

USING and INTO — Describe the resulting columns, input or output values for a dynamic SQL statement and provide a storage area for the input or output values.

SYNOPSIS

A *using-clause* has the form:

```
using-arguments | using-descriptor
```

where *using-arguments* is defined as:

```
{USING | INTO} host-variable-reference
      [, host-variable-reference]...
```

and *using-descriptor* is defined as:

```
{USING | INTO} SQL DESCRIPTOR descriptor-name
```

and *descriptor-name* is defined as:

```
character-string-literal | embedded-variable-name
```

DESCRIPTION

The syntactic element *using-clause* describes clauses beginning with either USING or INTO. When this element appears in a dynamic FETCH statement, it must use the keyword INTO. Elsewhere, it must use the keyword USING.

If an *embedded-variable-name* is specified, it must reference a character string variable.

For COBOL, *descriptor-name* must be a *character-string-literal*.

The assignment rules in Section 3.5 on page 51 also apply to the implied assignment of the value of any host variable or DATA field in an SQL descriptor area to the corresponding *dynamic-parameter* in the dynamic SQL statement, and *vice versa*, to the implied assignment of any column value of the current row of a cursor to the corresponding host variable or DATA field in an SQL descriptor area.

DIAGNOSTICS

If an item descriptor area describes an input value, the effective data type must be assignment-compatible with the assumed data type of the corresponding dynamic parameter; if it describes a target field, the effective data type must be assignment-compatible with the data type of the corresponding result column ('**07**006').

If *using-descriptor* is specified, an SQL descriptor area must have been allocated with the same *descriptor-name* using ALLOCATE DESCRIPTOR and not subsequently deallocated using DEALLOCATE DESCRIPTOR ('**33**000').

5.5.3 ALLOCATE DESCRIPTOR

NAME

ALLOCATE DESCRIPTOR — Allocate an SQL descriptor area.

SYNOPSIS

```
ALLOCATE DESCRIPTOR descriptor-name [WITH MAX occurrences]
```

where *occurrences* is defined as:

```
unsigned-integer | embedded-variable-name
```

DESCRIPTION

The ALLOCATE DESCRIPTOR statement allocates an SQL descriptor area identified by *descriptor-name*.

The SQL descriptor area will have at least as many item descriptor areas as the application requests with *occurrences*. All values of the SQL descriptor area are initially undefined. If the application omits *occurrences*, an implementation-defined number of item descriptor areas is allocated. See Section 7.1 on page 175.

The data type of *occurrences* must be exact numeric with scale 0. (For COBOL, *occurrences* must be an *unsigned-integer*.)

For the scope of a *descriptor-name*, see Section 4.7 on page 95.

DIAGNOSTICS

No other SQL descriptor area identified by the same *descriptor-name* must have been allocated and not subsequently deallocated ('**33**000').

The value of *occurrences* must be in the range from 1 through an implementation-defined maximum ('**07**009').

The value of *descriptor-name*, with leading and trailing spaces removed, must be a valid user-defined name ('**33**000').

5.5.4 DEALLOCATE DESCRIPTOR

NAME

DEALLOCATE DESCRIPTOR — Deallocate an SQL descriptor area.

SYNOPSIS

```
DEALLOCATE DESCRIPTOR descriptor-name
```

DESCRIPTION

The DEALLOCATE DESCRIPTOR statement deallocates an SQL descriptor area that was previously allocated for the specified *descriptor-name*.

DIAGNOSTICS

An SQL descriptor area named *descriptor-name* must have been allocated ('**33**000').

5.5.5 DESCRIBE

NAME

DESCRIBE — Get information about a prepared statement.

SYNOPSIS

```
DESCRIBE [INPUT | OUTPUT] statement-name
      USING SQL DESCRIPTOR descriptor-name
```

The syntax USING SQL DESCRIPTOR *descriptor-name* is one of the forms of the USING/INTO clause discussed in Section 5.5.2 on page 129, and is subject to the rules described there.

DESCRIPTION

The DESCRIBE statement obtains two types of information about a statement identified by *statement-name*:

OP
- The DESCRIBE INPUT form obtains information on the dynamic parameters corresponding to markers in the statement. For DESCRIBE INPUT, an ‘‘item’’ in the following description is a dynamic parameter. When DESCRIBE INPUT is applied to a prepared statement that has no dynamic parameters, the number of items is zero. DESCRIBE INPUT can apply to a cursor specification, and the cursor need not be open.
- The DESCRIBE OUTPUT form applies only to a cursor specification. The cursor need not be open. It obtains information on the columns of the derived table that would result from executing the cursor specification. For DESCRIBE OUTPUT, an ‘‘item’’ in the following description is a column. When DESCRIBE OUTPUT is applied to a prepared statement that is not a cursor specification, the number of items is zero.

If neither INPUT nor OUTPUT is specified, OUTPUT is assumed.

DESCRIBE stores a description of the items in the SQL descriptor area specified by *using-descriptor*. It sets the COUNT field to the number of items and sets the *i*th item descriptor area to describe the *i*th item. (If the number of items is zero, DESCRIBE sets the COUNT field of the SQL descriptor area to zero and assigns no values to item descriptor areas.)

DESCRIBE sets each item descriptor area based on the corresponding item from the prepared statement, as follows:

NAME, UNNAMED

For DESCRIBE OUTPUT where the column has a name, NAME is set to the column name and UNNAMED is set to 0. In all other cases, NAME is set to a value based on undefined criteria and UNNAMED is set to 1.

TYPE and Attributes

TYPE is set to indicate the item’s data type; and other fields that apply to that data type are set to indicate the item’s data attributes (see the table in Section 5.5.1 on page 124).

NULLABLE

For DESCRIBE OUTPUT, NULLABLE is set to 0 or 1 to indicate whether the column is constrained by NOT NULL. For DESCRIBE INPUT, NULLABLE is always set to 1 because dynamic parameters are never constrained by NOT NULL.

DIAGNOSTICS

There must be a prepared statement associated with *statement-name* ('**26**000'). (Moreover, the textual sequence of PREPARE and DESCRIBE is constrained on some implementations;

see Section 7.7 on page 184).

The **Success with warning** outcome occurs if there are more items to describe than allocated item descriptor areas ('**01**005'). COUNT is set to the total number of items to describe, but no values are assigned to item descriptor areas.

5.5.6 EXECUTE

NAME

EXECUTE — Associate input values with a prepared statement and execute it.

SYNOPSIS

```
EXECUTE statement-name [using-clause]
```

DESCRIPTION

An EXECUTE statement associates input values with a prepared statement and executes the statement.

If the prepared statement associated with *statement-name* contains *dynamic-parameters*, a *using-clause* describes their values (see Section 5.5.2 on page 129).

If the application uses the *using-arguments* form of *using-clause*, the *i*th *host-variable-reference* supplies the value for the *i*th *dynamic-parameter* that appears in the prepared statement.

If the application uses the *using-descriptor* form, the application must set the COUNT field of the specified *descriptor-name* to the number of input values (perhaps 0). The *i*th item descriptor area contains the input value and its description for the *i*th *dynamic-parameter* that appears in the prepared statement. The application sets the TYPE field based on the input value's data type; sets the LENGTH, PRECISION and SCALE fields, as applicable based on the input value's data type, to indicate the input value's data attributes (see Section 5.5.1 on page 124); and sets the DATA field to the input value.

The application uses the INDICATOR field as it would use a host indicator variable. The application sets INDICATOR to −1 if the data value is the null value, or to 0 if the data value is non-null.

DIAGNOSTICS

There must be a prepared statement associated with *statement-name* ('**26**000'), and it must not be a *cursor-specification* ('**07**003'). (Moreover, the textual sequence of PREPARE and EXECUTE is constrained on some implementations; see Section 7.7 on page 184).

If the prepared statement associated with the *statement-name* contains *dynamic-parameters*, the EXECUTE statement must have a *using-clause* ('**07**004').

If the application uses the *using-arguments* form of *using-clause*, the prepared statement must have one or more *dynamic-parameters*, and their number must equal the number of *host-variable-references* ('**07**001').

If the application uses the *using-descriptor* form of *using-clause*, it must set COUNT equal to the number of dynamic parameters in the prepared statement, and must set the fields of the item descriptor areas to valid and consistent values as applicable ('**07**001'). COUNT must not be negative or exceed the allocated number of item descriptor areas ('**07**008').

If EXECUTE does not produce any of the above diagnostics, it may produce other diagnostics based on the result of the prepared statement it executes.

5.5.7 EXECUTE IMMEDIATE

NAME

EXECUTE IMMEDIATE — Execute an SQL statement.

SYNOPSIS

```
EXECUTE IMMEDIATE embedded-variable-name
```

DESCRIPTION

The EXECUTE IMMEDIATE statement executes the SQL statement contained in *embedded-variable-name*. The effect of EXECUTE IMMEDIATE is comparable to using PREPARE on *embedded-variable-name* and then executing it with EXECUTE.

The SQL statement in *embedded-variable-name* can be any type of statement that could be PREPAREd, except cursor specifications. (See Section 5.5.11 on page 136).

DIAGNOSTICS

The *embedded-variable-name* must reference a character string variable that contains only the text of an SQL statement that could legally be prepared ('**42**000').

This SQL statement must not be a cursor specification and must not contain any *dynamic-parameter*s ('**42**000').

If EXECUTE IMMEDIATE does not produce any of the above diagnostics, it may produce other diagnostics based on the result of the statement it executes.

5.5.8 Dynamic FETCH

NAME

Dynamic FETCH — Fetch a row for a cursor.

SYNOPSIS

```
FETCH [FROM] cursor-name using-clause
```

DESCRIPTION

Dynamic FETCH advances *cursor-name* to the next row of its active set and assigns column values from that row to host variables or to fields in an SQL descriptor area, depending on the form of *using-clause* that is specified (see Section 5.5.2 on page 129). The cursor advances to the new row even if errors occur during the evaluation of result column values or during the assignment of these values to the targets specified by *using-clause*.

To use the *using-descriptor* form of *using-clause*, the application must first do the following:

- Set the COUNT field of the specified *descriptor-name* to the number of output values that are described.
- In each item descriptor area:
  - — Set the TYPE field based on the output value's data type.
  - — Set the LENGTH, PRECISION and SCALE fields, as applicable based on the output value's data type, to indicate the output value's data attributes (see Section 5.5.1 on page 124).

  The *i*th item descriptor area contains the description of the *i*th output value.

The FETCH statement sets the DATA field to the respective data value; sets the INDICATOR field to −1 for the null value, or to 0 for any non-null value; and sets the RETURNED_LENGTH field as described in Section 5.5.1 on page 124.

DIAGNOSTICS

The cursor must be open ('**24**000').

If the application uses the *using-arguments* form of *using-clause*, the number of *host-variable-references* must equal the number of columns in the result set ('**07**002'). If the application uses the *using-descriptor* form of *using-clause*, it must set COUNT to the number of columns in the result set and must set the fields of the item descriptor areas to valid and consistent values as applicable ('**07**002'). For these errors, the cursor position is unchanged.

The value of COUNT must not be negative or exceed the allocated number of item descriptor areas ('**07**008').

The **No data** outcome occurs if (1) the result set is empty, (2) the cursor was positioned on the last row, or (3) the cursor was positioned after the last row ('**02**000'). No assignment occurs to host variables or to fields of item descriptor areas.

5.5.9 GET DESCRIPTOR

NAME

GET DESCRIPTOR — Get information from an SQL descriptor area.

SYNOPSIS

```
GET DESCRIPTOR descriptor-name get-descriptor-information
```

where *get-descriptor-information* is defined as:

```
embedded-variable-name-1 = COUNT
    | VALUE item-number get-item-info [, get-item-info]...
```

and *get-item-info* is defined as:

```
embedded-variable-name-2 = field-name
```

and *item-number* is defined as:

```
unsigned-integer | embedded-variable-name-3
```

and *field-name* is defined as:

```
TYPE | LENGTH | PRECISION | SCALE
    | DATETIME_INTERVAL_CODE | DATETIME_INTERVAL_PRECISION
    | NULLABLE | INDICATOR | DATA | NAME | UNNAMED | RETURNED_LENGTH
```

DESCRIPTION

A GET DESCRIPTOR statement retrieves values from the SQL descriptor area *descriptor-name*. To determine the number of active item descriptor areas, the application uses the form *embedded-variable-name-1* = COUNT. The application uses the VALUE form to retrieve fields of the item descriptor area specified by *item-number*.

The data type of *embedded-variable-name-2* must be equal to the data type of the associated *field-name* (see Section 5.5.1 on page 124).

The value of DATA is undefined if INDICATOR indicates the null value. If the value of DATA is undefined, the value of the embedded variable associated with DATA is undefined after execution of the statement.

If present, *embedded-variable-name-1* and *item-number* must be exact numeric with scale 0.

DIAGNOSTICS

The *descriptor-name* must identify a currently allocated SQL descriptor area ('**33**000').

item-number must be in the range from 1 through the number of item descriptor areas the application allocated ('**07**009'). If *item-number* exceeds the value of COUNT, values are retrieved but the **No data** outcome occurs ('**02**000').

If DATA is specified in a GET DESCRIPTOR statement and INDICATOR indicates a null value, then INDICATOR must also be specified in that statement ('**22**002').

Any host variable to which the DATA field is assigned must be appropriate for the data type, as specified by the item descriptor's current TYPE, and by each of the DATETIME_INTERVAL_CODE, DATETIME_INTERVAL_PRECISION, LENGTH, PRECISION and SCALE fields that apply to that type ('**22**005').

5.5.10 Dynamic OPEN

NAME

Dynamic OPEN — Associate input values and open a cursor.

SYNOPSIS

`OPEN` *`cursor-name`* `[`*`using-clause`*`]`

DESCRIPTION

The description of the OPEN statement in Section 5.4.6 on page 120 applies to the dynamic OPEN statement as well.

If the prepared statement associated with *cursor-name* contains *dynamic-parameters*, a *using-clause* describes their values (see Section 5.5.2 on page 129).

If the application uses the *using-arguments* form of *using-clause*, the *i*th *host-variable-reference* supplies the value for the *i*th *dynamic-parameter* that appears in the prepared statement.

If the application uses the *using-descriptor* form, the application must set the COUNT field of the specified *descriptor-name* to the number of input values (perhaps 0). The *i*th item descriptor area contains the input value and its description for the *i*th *dynamic-parameter* that appears in the prepared statement. The application sets the TYPE field based on the input value's data type; sets the LENGTH, PRECISION and SCALE fields, as applicable based on the input value's data type, to indicate the input value's data attributes (see Section 5.5.1 on page 124); and sets the DATA field to the input value.

The application uses the INDICATOR field as it would use a host indicator variable. The application sets INDICATOR to −1 if the data value is the null value, or to 0 if the data value is non-null.

DIAGNOSTICS

There must be a prepared statement associated with *cursor-name* ('**26**000').

If the prepared statement associated with the *cursor-name* contains *dynamic-parameters*, the OPEN statement must have a *using-clause* ('**07**004').

If the application uses the *using-arguments* form of *using-clause*, the prepared statement must have one or more *dynamic-parameters*, and their number must equal the number of *host-variable-references* ('**07**001').

If the application uses the *using-descriptor* form of *using-clause*, it must set COUNT equal to the number of dynamic parameters in the prepared statement, and must set the fields of the item descriptor areas to valid and consistent values as applicable ('**07**001'). COUNT must not be negative or exceed the allocated number of item descriptor areas ('**07**008').

5.5.11 PREPARE

NAME

PREPARE — Prepare a statement for execution.

SYNOPSIS

```
PREPARE statement-name FROM embedded-variable-name
```

DESCRIPTION

The PREPARE statement creates an executable SQL statement from the character string in *embedded-variable-name*.

The following statements and specifications are allowed as contents of the *embedded-variable-name*: ALTER, CREATE, *cursor-specification*, searched DELETE, dynamic positioned DELETE, DROP, GRANT, INSERT, REVOKE, searched UPDATE, and dynamic positioned UPDATE. (It is implementation-defined which, if any, other statements can be PREPAREd.)

For the scope of a *statement-name*, see Section 4.7 on page 95.

Whenever a *dynamic-parameter* appears in an expression, it has an assumed data type according to the following rules:

Location of *dynamic-parameter*	**Assumed Data Type**
An operand of a binary arithmetic or comparison operator.	Same as the other operand.
BETWEEN: First operand.	Same as the second operand.
BETWEEN: 2nd or 3rd operand.	Same as the first operand.
IN: *expression*	Same as the first value, or as the result column of the *sub-query*.
IN: *value*	Same as the *expression*.
expression in quantified predicate.	Same as the result column of the *sub-query*.
INSERT: *expression*	Same as the insert column.
LIKE: *pattern-value*	VARCHAR(254)
LIKE: *escape-character*	VARCHAR(254)
UPDATE: update value.	Same as the update column.

Table 5-2 Assumed Data Type of Dynamic Parameters Based on Context

Using these assumptions, the data type of the surrounding *expression* is determined as described in Section 3.3 on page 47. If the assumed data type is based on another operand, the other operand cannot also be a *dynamic-parameter* because the resulting data type cannot be inferred at the time of the PREPARE statement. Thus *dynamic-parameter*s cannot occur (1) as both operands of a binary operator, (2) as both the first and second operands of BETWEEN, (3) as both the first and third operands of BETWEEN, or (4) as both the *expression* and the first *value* of IN. A *dynamic-parameter* likewise cannot appear alone as the operand of unary + or – or the argument of a set function.

A successful execution of PREPARE destroys any existing prepared statement that was PREPAREd using the same *statement-name*.

If the prepared statement is a dynamic positioned DELETE or dynamic positioned UPDATE statement, the associated cursor must be open and the prepared statement remains in existence at least until that cursor instance is closed. Otherwise, as long as *statement-name* is not reused, the prepared statement remains in existence at least until the end of the current transaction.

If the prepared statement is a *cursor-specification* and the *statement-name* is associated with a dynamic cursor, then an association is made between the prepared statement and the cursor and is preserved until the prepared statement is destroyed.

DIAGNOSTICS

The *statement-name* must not identify an existing prepared statement that is the *cursor-specification* of an open cursor ('**24**000').

If the contents of *embedded-variable-name* are a dynamic positioned DELETE or a dynamic positioned UPDATE statement, the cursor referenced by the statement being prepared must be open ('**24**000').

If the *statement-name* is associated with a dynamic cursor, the prepared statement must be a *cursor-specification* ('**07**005').

The *embedded-variable-name* must reference a character string variable that holds a valid SQL statement (see **DESCRIPTION** for a list of allowed statements) ('**42**000'). This means that it cannot contain an *sql-prefix*, *sql-terminator* or host language comment. Moreover, this variable cannot contain any of the following ('**42**000'):

- an *embedded-variable-name*
- a *dynamic-parameter* in a context that prevents the data type of the surrounding expression from being inferred (see above), or anywhere in a *select-list*
- an SQL comment.

5.5.12 SET DESCRIPTOR

NAME

SET DESCRIPTOR — Assign values to an SQL descriptor area.

SYNOPSIS

```
SET DESCRIPTOR descriptor-name set-descriptor-information
```

where *set-descriptor-information* is defined as:

```
COUNT = {embedded-variable-name-1 | unsigned-integer}
      | VALUE item-number set-item-info [, set-item-info]...
```

and *set-item-info* is defined as:

```
field-name = {embedded-variable-name-2 | literal}
```

and *item-number* is defined as:

```
unsigned integer | embedded-variable-name-3
```

and *field-name* is defined as:

```
TYPE | LENGTH | PRECISION | SCALE
      | DATETIME_INTERVAL_CODE | DATETIME_INTERVAL_PRECISION
      | INDICATOR | DATA
```

DESCRIPTION

A SET DESCRIPTOR statement assigns values to the SQL descriptor area *descriptor-name*. To specify the number of item descriptor areas, the application uses the COUNT form. The application uses the VALUE form to assign values to fields of the item descriptor area specified by *item-number*.

The data type of *embedded-variable-name-2* or *literal* must be the data type of the specified *field-name* (see Section 5.5.1 on page 124).

If present, *embedded-variable-name-1* and *item-number* must be exact numeric with scale 0.

Automatic Setting of Fields

When a statement specifies a value for any field of an item descriptor area other than DATA, the resulting value of DATA is undefined. When a statement specifies a value for the TYPE field, some other fields receive default values as indicated below and the remaining fields are set to undefined values.

Application Sets TYPE to:	**Other Fields Implicitly Set**
CHARACTER	LENGTH set to 1
CHARACTER VARYING	LENGTH set to 1
DATETIME	PRECISION set to 0
DECIMAL or NUMERIC	SCALE set to 0; PRECISION set to the implementation-defined default precision for the respective data type
FLOAT	PRECISION set to the implementation-defined default precision for FLOAT
INTERVAL	DATETIME_INTERVAL_PRECISION set to 2

Table 5-3 Implicit Setting of Item Descriptor Area Fields

Moreover, when the TYPE field is DATETIME or INTERVAL and a statement specifies a date/time or interval subcode by setting the value of the DATETIME_INTERVAL_CODE field, some other fields receive default values as indicated below:

- When TYPE is DATETIME and a statement sets DATETIME_INTERVAL_CODE to indicate TIME, then PRECISION is set to 0. For TIMESTAMP, PRECISION is set to 6.
- When TYPE is INTERVAL and a statement sets DATETIME_INTERVAL_CODE to indicate an INTERVAL data type that includes a SECOND field, PRECISION is set to 6. For other INTERVAL data types, PRECISION is set to 0.

When a statement specifies values for DATA and for other fields, the assignment to DATA occurs last. When a statement specifies values for TYPE and for other fields, the assignment to TYPE occurs first. When a statement specifies field values for date/time or interval data types, the assignments are performed so that no explicit assignment is overridden by the implicit assignments listed above.

DIAGNOSTICS

The *descriptor-name* must identify a currently allocated SQL descriptor area ('**33**000').

item-number must be in the range from 1 through the number of item descriptor areas the application allocated ('**07**009').

Any host variable that assigns a value to the DATA field must have the same data type as the effective data type specified by that item descriptor area's current TYPE and by each of the DATETIME_INTERVAL_CODE, DATETIME_INTERVAL_PRECISION, LENGTH, PRECISION and SCALE fields that apply to that type ('**22**005').

5.6 Transaction Control Statements

A transaction is a sequence of operations on a database that must be atomic. This means that the database system must either effect all of the operations or effect none of them.

A transaction starts (becomes active) when a program that does not already have an active transaction executes an SQL statement that operates on a database. An ‘‘SQL statement that operates on a database’’ does not include:

- the transaction control statements COMMIT, ROLLBACK and SET TRANSACTION discussed later in this section (although, technically, completing a transaction using COMMIT and ROLLBACK operates on a database, it does not start a new transaction)
- the connection statements (see Section 5.7 on page 142); this means an application can change connections without starting a new transaction
- the session statements SET CATALOG, SET NAMES and SET SCHEMA (see Section 5.8 on page 147)
- the GET DIAGNOSTICS statement (see Section 5.10 on page 150).

Applications should complete each transaction explicitly with either the COMMIT or ROLLBACK statement. Ceasing execution without ending the transaction causes either a COMMIT or ROLLBACK according to implementation-defined criteria.

Applications should not mix data definition statements and data manipulation statements within transactions. See Section 7.4 on page 183.

5.6.1 General Diagnostics

X/Open Distributed Transaction Processing (DTP) documents describe methods of coordinating databases so that changes to any recoverable resources are committed or rolled back atomically. For example, the emerging **Transaction Demarcation API** specifies non-SQL functions by which an application declares the start, commitment, or rollback of a global transaction. These global transaction control functions should eventually be used instead of SQL transaction control statements so that SQL work can be coordinated with non-SQL work.

If an implementation supports X/Open DTP, an application must not use both the global transaction control functions and the SQL transaction control statements in a single transaction. Specifically, the COMMIT and ROLLBACK statements fail if executed within a global transaction that the application started by calling *tx_begin*( ) (’**2D**000’).

5.6.2 COMMIT

NAME

COMMIT — Complete the current transaction by applying all of the database changes.

SYNOPSIS

```
COMMIT [WORK]
```

DESCRIPTION

The COMMIT statement completes the current transaction; closes any cursors that the transaction opened; and commits any changes made to the database since the transaction started.

In various implementations, COMMIT can fail for a variety of reasons, with different implications on the state of the database. See Section 7.5 on page 184.

DIAGNOSTICS

If the connection fails during the COMMIT statement, the application may not be able to determine whether commitment occurred before the failure ('**08**007').

5.6.3 ROLLBACK

NAME

ROLLBACK — Complete the current transaction by applying none of the database changes.

SYNOPSIS

```
ROLLBACK [WORK]
```

DESCRIPTION

The ROLLBACK statement completes the current transaction; closes any cursors that the transaction opened; and provides that any database changes made since the transaction started do not take effect.

5.6.4 SET TRANSACTION

NAME

SET TRANSACTION — Set the attributes of the next transaction.

SYNOPSIS

```
SET TRANSACTION transaction-mode[, transaction-mode]...
```

where *transaction-mode* is defined as:

```
DIAGNOSTICS SIZE number
| ISOLATION LEVEL READ UNCOMMITTED
| ISOLATION LEVEL READ COMMITTED
| ISOLATION LEVEL REPEATABLE READ
| ISOLATION LEVEL SERIALIZABLE
| READ ONLY
| READ WRITE
```

DESCRIPTION

The SET TRANSACTION statement specifies the values of the transaction attributes discussed in Section 2.7.1 on page 28. These values apply to the next transaction the application may begin over the current connection. It is implementation-defined whether the specified attributes persist if the application should disconnect and reconnect to the same database.

The keywords SET TRANSACTION are followed by *transaction-mode* clauses. The DIAGNOSTICS SIZE clause specifies the size of the diagnostics area; the READ ONLY and READ WRITE clauses specify the access mode; the others specify the isolation level (see Section 2.7.2 on page 29. Redundant and contradictory designations in a single SET TRANSACTION statement are invalid.

Defaults

If a SET TRANSACTION statement does not specify the access mode, the resulting access mode is READ WRITE, except that a statement that specifies ISOLATION LEVEL READ UNCOMMITTED implies READ ONLY. It is invalid to specify both READ WRITE and READ UNCOMMITTED in the same SET TRANSACTION statement.

If a SET TRANSACTION statement does not specify the isolation level, then ISOLATION LEVEL SERIALIZABLE is assumed.

If a SET TRANSACTION statement does not specify the size of the diagnostics area, then an implementation-defined default size is assumed.

If an application starts a transaction without having executed SET TRANSACTION since the end of any previous transaction, the access mode is READ WRITE, the isolation level is ISOLATION LEVEL SERIALIZABLE, and the size of the diagnostics area is implementation-defined.

DIAGNOSTICS

A transaction must not already be active on the current connection ('**25**000').

In the DIAGNOSTICS SIZE form, *number* must be 1 or greater and must not exceed an implementation-defined maximum value ('**35**000').

APPLICATION USAGE

Since SET TRANSACTION affects only the next transaction, applications should execute SET TRANSACTION before the start of each transaction.

5.7 Connection Statements

Connection statements give the application access to multiple servers.

5.7.1 Current and Dormant Connections

The CONNECT statement connects the client (and effectively, the application bound to that client) to a server; the DISCONNECT statement ends this connection. A client can be connected to several servers at the same time, and can establish several connections to the same server. (The maximum number of concurrent connections is implementation-defined.) However, only one connection is **current** at any time. Other connections are **dormant**.

The connection that the CONNECT statement specifies becomes the current one. If the application establishes multiple connections, it can use the SET CONNECTION statement to select one connection as the current one. In either case, if a previous connection was current, it becomes dormant. The application cannot use a server through a dormant connection; it must first use SET CONNECTION.

Making a connection dormant and then current again is comparable to disconnecting and then reconnecting to the server, except that dormancy typically avoids the need to perform authentication again, and may avoid costs and uses of resources associated with an initial connection.

A current or dormant connection is **active** if any SQL statement that operates on a database has been successfully executed at its associated server using that connection during the current transaction. The application must complete the current transaction before DISCONNECT will succeed.

5.7.2 Default Connection

If the first SQL statement an application executes is not CONNECT, a **default connection** is established to an implementation-defined server. This remains the current connection until the application executes a CONNECT statement. All connection statements let the application specify this connection explicitly using the keyword DEFAULT. DEFAULT is a discrete connection just as named connections are.

5.7.3 State Table

The following state table shows the state of connection *conn1* resulting from the execution of a connection statement referencing it or referencing some other connection *conn2*. A blank entry means the state of *conn1* is unchanged. A SQLSTATE value means that the statement fails because of the error indicated by that SQLSTATE value.

	Nonexistent	**Current**	**Dormant**
`CONNECT TO` *serv* `AS` *conn1*	Current	’**08**002’	’**08**002’
`DISCONNECT` *conn1*	’**08**003’	Nonexistent	Nonexistent
`SET CONNECTION` *conn1*	’**08**003’		Current
`CONNECT TO` *serv* `AS` *conn2*		Dormant	
`DISCONNECT` *conn2*			
`SET CONNECTION` *conn2*		Dormant	

A connection’s initial state in this table is **Nonexistent**.

5.7.4 Connection Context

Each current or dormant connection has a set of information called the **connection context**. This includes:

- the position of all open cursors
- the contents of all SQL descriptor areas
- the name of the current user and all information the server associates with this name
- all state information the server needs in order to reference the application
- other implementation-defined context.

When the current connection becomes dormant, the server saves that connection's context. When a dormant connection becomes current, the server restores the context to its exact state at the time the connection became dormant.

5.7.5 General Diagnostics

It is implementation-defined whether a single transaction (see Section 5.6 on page 139) may encompass more than one server. If an implementation disallows this, then using CONNECT or SET CONNECTION when a transaction is active sets SQLSTATE to '**0A**001'.

Connection Errors

The DISCONNECT statement (see Section 5.7.7 on page 146) describes how an application explicitly disconnects from a server. Disconnection can occur independently of the application, because of communication errors, administrative action, or other events. Four SQLSTATE values indicate a disconnection:

- The value '**08**006' occurs on a SET CONNECTION statement when the selected connection has been independently disconnected and cannot be reestablished.
- The value '**08**007' occurs on a COMMIT statement when the current connection has been independently disconnected.
- The value '**40**003' occurs on a statement, other than COMMIT, that is processed by a server when the current connection has been independently disconnected. The transaction is rolled back.
- The value '**08**003' indicates either the connection selected by a SET CONNECTION or DISCONNECT statement has been explicitly disconnected or the following conditions apply:
  - — The statement is processed by a server.
  - — The current connection has been explicitly disconnected, or independently disconnected but not since the last statement was executed.
  - — The lost current connection was not the implicit default connection.

A DISCONNECT statement specifying an active connection while a transaction is active is rejected ('**25**000'). However, independent disconnection of the current connection (except on a COMMIT) rolls back any active transaction ('**40**003').

If the current connection is disconnected, then there is no current connection; subsequent SQL statements report the nonexistent connection ('**08**003') until the application uses a CONNECT or SET CONNECTION statement. If the implicit default connection is independently disconnected, then implicit reconnection occurs, as it did when the application began.

The diagnostics area may contain additional information on the reason for a connection failure. (See also the referenced X/Open **RDA** Specification.)

The following flowchart illustrates the reporting of connection errors and the special considerations that apply to the implicit default connection:

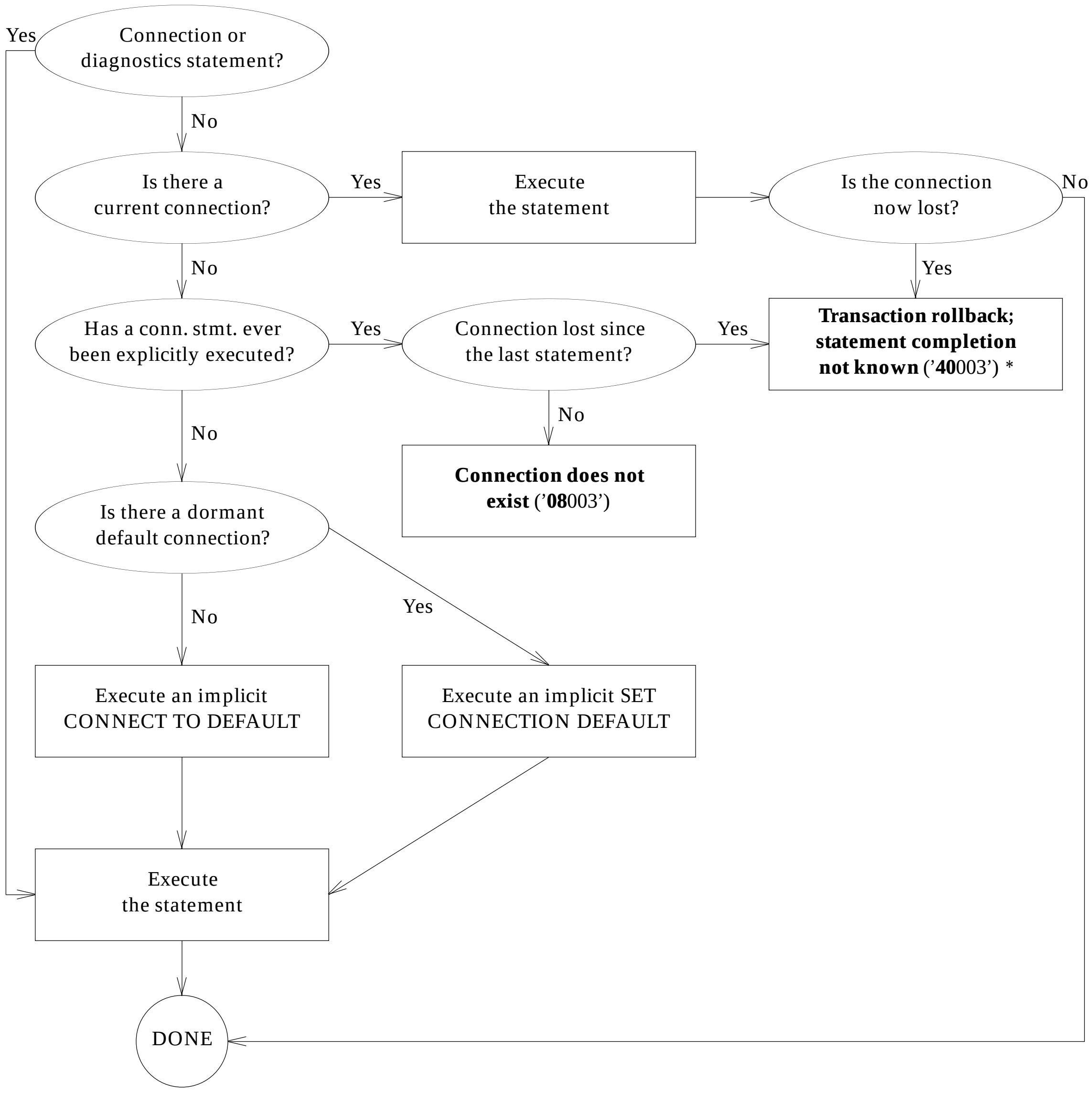
Connection or diagnostics statement?
Yes
No
Is there a current connection?
Yes
Execute the statement
Is the connection now lost?
No
Yes
Has a conn. stmt. ever been explicitly executed?
Yes
Connection lost since the last statement?
Yes
Transaction rollback; statement completion not known ('40003') *
No
No
Connection does not exist ('08003')
Is there a dormant default connection?
No
Yes
Execute an implicit CONNECT TO DEFAULT
Execute an implicit SET CONNECTION DEFAULT
Execute the statement
DONE

Figure 5-1 Reporting of Connection Errors (Flowchart)

* If the statement was COMMIT, then SQLSTATE may instead be '**08**007'. If it is, the application cannot determine whether the server committed the transaction before the connection was lost.

5.7.6 CONNECT

NAME

CONNECT — Connect the client with a server.

SYNOPSIS

```
CONNECT TO
      {server [AS connection]
      [USER user [USING authentication]]
      | DEFAULT}
```

where *server*, *connection* and *user* are defined as:

character-string-literal | *embedded-variable-name*

and *authentication* is defined as:

embedded-variable-name

and all *embedded-variable-name*s reference character-string variables.

DESCRIPTION

CONNECT connects the application to a server. If DEFAULT appears, the connection is to the default server. If *server* appears, the connection is to the server it identifies. The length of *server* must not exceed 128 characters. The specified connection becomes the current connection. If another connection was current, it becomes dormant.

The optional *connection* can be used to reference the connection to *server* in any subsequent SET CONNECTION or DISCONNECT statements. If the application does not specify *connection*, then *server* is the implicit *connection*.

IMPLEMENTATION-DEFINED ISSUES

The mapping from *server* or DEFAULT to an actual server is implementation-defined.

The specified server uses the contents of the optional USER clause, and may apply other criteria, at the time of the CONNECT statement to determine whether to accept or reject the connection. (If the USER clause is omitted, there is an implicit *user* whose value is implementation-defined.) If the server accepts the connection, *user* becomes the value of the current authorisation identifier.

DIAGNOSTICS

A CONNECT statement can fail if it cannot establish a connection for implementation-defined reasons (’**08**001’), or if the server rejects the connection (’**08**004’).

If DEFAULT is specified, the default connection must not be current or dormant; otherwise, *connection* must not identify a current or dormant connection (’**08**002’).

The value of *connection*, with leading and trailing spaces removed, must follow the syntax rules for user-defined names (’**2E**000’).

The value of *user*, with leading and trailing spaces removed, must follow the syntax rules for user-defined names and must not violate any implementation-defined restrictions on its value (’**28**000’).

5.7.7 DISCONNECT

NAME

DISCONNECT — End a connection between a client and server.

SYNOPSIS

```
DISCONNECT {connection | ALL | CURRENT | DEFAULT}
```

where *connection* is defined as:

```
character-string-literal | embedded-variable-name
```

and *embedded-variable-name* references a character-string variable.

DESCRIPTION

DISCONNECT ends a connection. The form DISCONNECT *connection* ends the specified connection. DISCONNECT CURRENT ends the current connection. DISCONNECT DEFAULT ends the default connection. DISCONNECT ALL ends all the application's current and dormant connections.

A DISCONNECT statement that does not end the current connection makes no change to the context of the current connection.

DIAGNOSTICS

If CURRENT is specified, there must be a current connection; otherwise, all specified connections must be either current or dormant ('**08**003').

An active connection must not be disconnected ('**25**000').

The **Success with Warning** outcome occurs if errors occur performing the disconnection ('**01**002').

5.7.8 SET CONNECTION

NAME

SET CONNECTION — Make a specified connection the current one.

SYNOPSIS

```
SET CONNECTION {connection | DEFAULT}
```

where *connection* is defined as:

```
character-string-literal | embedded-variable-name
```

and *embedded-variable-name* references a character-string variable.

DESCRIPTION

The SET CONNECTION statement makes the specified connection the current one. If another connection was previously current, it becomes dormant.

DIAGNOSTICS

The specified connection must be current or dormant ('**08**003').

A SET CONNECTION statement can fail if the specified connection cannot be made current ('**08**006').

5.8 Session Statements

Session statements set information that is associated with the session in progress over an active connection to a server. See Section 2.4.1 on page 19 for a discussion of the concept. See Section 3.1.3 on page 35 for a discussion of the syntax of object naming.

5.8.1 SET CATALOG

NAME

SET CATALOG — Set the default catalog name for unqualified object references.

SYNOPSIS

```
SET CATALOG catalog-name
```

where *catalog-name* is defined as:

```
character-string-literal | embedded-variable-name
```

DESCRIPTION

The SET CATALOG statement sets the default catalog name (for references to objects that do not specify a catalog name) to the value of *catalog-name*, with any leading and trailing spaces removed.

Execution of SET CATALOG does not check that *catalog-name* actually exists. This check occurs when a subsequent dynamic SQL statement is prepared that references an object that does not specify a catalog. The default catalog name that is set by the SET CATALOG statement does not affect static SQL statements and affects only dynamic SQL statements that are prepared after the execution of the SET CATALOG statement.

DIAGNOSTICS

The value of *schema-name*, with leading and trailing spaces removed, must be a *user-defined-name* ('**3D**000').

5.8.2 SET NAMES

NAME

SET NAMES — Set the default character set.

SYNOPSIS

```
SET NAMES character-set-name
```

where *character-set-name* is defined as:

```
character-string-literal | embedded-variable-name
```

DESCRIPTION

The SET NAMES statement specifies the default character set for user-defined names and character-string literals to the value of *character-set-name*, with any leading and trailing spaces removed.

Execution of SET NAMES does not check that *character-set-name* actually exists. This check occurs when a subsequent dynamic SQL statement is prepared that contains a *user-defined name* or *character-string-literal* that does not explicitly specify a character set. The default character set that is set by the SET NAMES statement affects only dynamic SQL statements that are prepared after the execution of the SET NAMES statement.

DIAGNOSTICS

The value of *character-set-name*, with leading and trailing spaces removed, must be a *user-defined-name* ('**2C**000').

5.8.3 SET SCHEMA

NAME

SET SCHEMA — Set the default schema name for unqualified object references.

SYNOPSIS

```
SET SCHEMA schema-name
```

where *schema-name* is defined as:

```
character-string-literal | embedded-variable-name
```

DESCRIPTION

The SET SCHEMA statement sets the default schema name, and may set the default catalog name, for references to unqualified objects (that is, for objects that do not specify a catalog and schema name).

If *schema-name* contains a '.' character, then the characters preceding '.' are the default catalog name and the characters following '.' are the default schema name. If *schema-name* does not contain a '.' character, then *schema-name* becomes the default schema name and the default catalog name does not change.

The implementation ignores any leading and trailing spaces in *schema-name*.

Execution of SET SCHEMA does not check that any specified schema and catalog actually exist. This check occurs when a subsequent dynamic SQL statement is prepared that references an unqualified object. The default schema name that is set by the SET SCHEMA statement affects only dynamic SQL statements that are prepared after the execution of the SET SCHEMA statement.

DIAGNOSTICS

The value of *schema-name*, with leading and trailing spaces removed, must be either a *user-defined-name*, or two *user-defined-name*s separated by a single '.' character ('**3F**000').

5.9 SET SESSION AUTHORIZATION

NAME

SET SESSION AUTHORIZATION -Set the session authorization identifier

SYNOPSIS

```
SET SESSION AUTHORIZATION session-authorization
```

where *session-authorization* is defined as:

```
character-string-literal | embedded-variable-name
```

DESCRIPTION

The SET SESSION AUTHORIZATION statement specifies a new value to be used for the authorization identifier. This identifier is used to validate the privileges of subsequent SQL statements that are running under invoker's rights.

DIAGNOSTICS

A transaction must not be active on the current connection ('**25**000').

The value of *session-authorization* , with leading and trailing spaces removed, must follow the syntax rules for user-defined names and must not violate any implementation-defined restrictions on its value ('**28**000').

The current authorization identifier must satisfy any implementation-defined tests for authorisation to execute the SET SESSION AUTHORIZATION statement ('**28**000').

5.10 Diagnostic Statement

NAME

GET DIAGNOSTICS — Get information from the diagnostics area.

SYNOPSIS

```
GET DIAGNOSTICS {statement-information | exception-information}
```

where *statement-information* is defined as:

```
statement-information-item [, statement-information-item]...
```

and *statement-information-item* is defined as:

```
embedded-variable-name-1 = field-name-1
```

and *exception-information* is defined as:

```
EXCEPTION exception-number exception-information-item
      [, exception-information-item]...
```

and *exception-information-item* is defined as:

```
embedded-variable-name-2 = field-name-2
```

and *field-name-1* is defined as:

```
NUMBER | MORE | DYNAMIC_FUNCTION | DYNAMIC_FUNCTION_CODE
      | ROW_COUNT
```

and *field-name-2* is one of the following:

```
    CATALOG_NAME         CONSTRAINT_NAME         RETURNED_SQLSTATE
    CLASS_ORIGIN         CONSTRAINT_SCHEMA       SCHEMA_NAME
    COLUMN_NAME          CURSOR_NAME             SERVER_NAME
    CONDITION_NUMBER     MESSAGE_LENGTH          SUBCLASS_ORIGIN
    CONNECTION_NAME      MESSAGE_OCTET_LENGTH    TABLE_NAME
    CONSTRAINT_CATALOG   MESSAGE_TEXT
```

and *exception-number* is defined as:

```
unsigned-integer | embedded-variable-name-3
```

DESCRIPTION

A GET DIAGNOSTICS statement retrieves selected status information from the diagnostics area. A given GET DIAGNOSTICS statement retrieves either count and overflow information, or information on a specific exception.

The GET DIAGNOSTICS statement never changes the contents of the diagnostics area, although it does set SQLSTATE.

Retrieving Count and Overflow

This form of GET DIAGNOSTICS deposits into the specified *embedded-variable-name-1* the value of the diagnostics area field named by *field-name-1*. The data type of *embedded-variable-name-1* must be that of the requested field, as described below.

NUMBER	An INTEGER that is the number of exceptions that the most recently executed SQL statement placed into the diagnostics area.
MORE	A character string of length 1, whose value is 'N' if the most recently executed SQL statement stored in the diagnostics area all the exceptions it

detected, or 'Y' if it detected more exceptions than it stored in the diagnostics area.

DYNAMIC_FUNCTION
A character string of variable length, whose value denotes the statement type of the most recently executed SQL statement, from the table below.

DYNAMIC_FUNCTION_CODE
An integer whose value denotes the statement type of the most recently executed SQL statement, from the table below. Positive values were chosen to align with emerging standards work. Negative values denote statement types that are X/Open extensions to the International Standard.

SQL Statement Executed	**Value of DYNAMIC_FUNCTION**	**Value of DYNAMIC_ FUNCTION_CODE**
ALTER TABLE	ALTER TABLE	4
CREATE INDEX	CREATE INDEX	-1
CREATE TABLE	CREATE TABLE	77
CREATE VIEW	CREATE VIEW	84
cursor-specification	SELECT CURSOR	-3
searched DELETE	DELETE WHERE	19
dynamic positioned DELETE	DYNAMIC DELETE CURSOR	38
DROP INDEX	DROP INDEX	-2
DROP TABLE	DROP TABLE	32
DROP VIEW	DROP VIEW	36
GRANT	GRANT	48
INSERT	INSERT	50
REVOKE	REVOKE	59
searched UPDATE	UPDATE WHERE	82
dynamic positioned UPDATE	DYNAMIC UPDATE CURSOR	81

ROW_COUNT An exact numeric value with scale 0 whose value, after certain SQL statements, is the number of rows that statement processed.

— The INSERT statement sets ROW_COUNT to the number of rows inserted into the table.

— The searched UPDATE statement sets ROW_COUNT to the number of rows updated in the table.

— The searched DELETE statement sets ROW_COUNT to the number of rows deleted from the table.

After any other SQL statement, the value of ROW_COUNT is undefined.

The count does not include any rows in other tables that the statement may have affected.

OP X/Open encourages implementations to define the row count for additional SQL statements as follows:

a. For any positioned UPDATE and positioned DELETE statement, the row count is 1.

b. For any other SQL statement, the row count is 0.

A portable application cannot assume these semantics, and should not use the row count unless the SQL statement executed was INSERT, searched UPDATE or searched DELETE.

Retrieving Exception Data

This form of GET DIAGNOSTICS returns one or more fields of an actual exception that the most recent SQL statement raised. The application specifies the exception by number, using either an *unsigned-integer* or *embedded-variable-name-3* (which must be an exact numeric with scale 0). The first exception is number 1.

The GET DIAGNOSTICS statement deposits into the specified *embedded-variable-name-2* the value of the diagnostics area field named by *field-name-1*. The data type of *embedded-variable-name-2* must be that of the requested field, as described below.

CATALOG_NAME
A varying-length character string of maximum length 254. In exceptions that pertain to a table, this is the name of the catalog that contains the table.

CLASS_ORIGIN
A varying-length character string of maximum length 254 that identifies the defining source of the class portion of RETURNED_SQLSTATE. Its value is 'ISO 9075' if the International Standard defines the class. (This is the case for all diagnostics defined in this specification.) Otherwise, its value is implementation-defined and must not be 'ISO 9075'.

COLUMN_NAME
A varying-length character string of maximum length 254. In exceptions that pertain to a specific column of a table, this is the name of the column.

CONDITION_NUMBER
An exact numeric with scale 0 that is the serial number of the diagnostic record; that is, the value supplied as *exception-number*.

CONNECTION_NAME
A varying length character string of maximum length 128 whose value is determined as follows:

- If the most recent statement was a connection statement, it contains the explicit or implicit *connection* referenced by that statement.
- Otherwise, it contains the explicit or implicit *connection* associated with the most recently executed explicit or implicit CONNECT or SET CONNECTION statement.

DEFAULT identifies the default connection.

CONSTRAINT_CATALOG
CONSTRAINT_NAME
CONSTRAINT_SCHEMA
These fields are reserved for future use.

CURSOR_NAME
A varying-length character string of maximum length 254. In exceptions that pertain to a cursor, this is the name of the cursor.

MESSAGE_TEXT
A varying-length character string of maximum length 254 that contains an implementation-defined explanation of this exception (such as an error message). MESSAGE_TEXT may be a zero-length string.

MESSAGE_LENGTH
An exact numeric value with scale 0 whose value is the length of the current MESSAGE_TEXT string in characters. See also MESSAGE_OCTET_LENGTH.

MESSAGE_OCTET_LENGTH
An exact numeric value with scale 0 whose value is the length of the current MESSAGE_TEXT string in octets. See also MESSAGE_LENGTH.

RETURNED_SQLSTATE
A character string of length 5 that holds the SQLSTATE value describing this type of exception (see Appendix B).

SCHEMA_NAME
A varying-length character string of maximum length 254. In exceptions that pertain to a table, this is the name of the schema that contains the table.

SERVER_NAME
A varying-length character string of maximum length 128 that contains the value of *server* associated with CONNECTION_NAME.

DEFAULT identifies the default server.

SUBCLASS_ORIGIN
A field with the same format as CLASS_ORIGIN above, that identifies the defining source of the subclass portion of RETURNED_SQLSTATE. The following are the valid values for diagnostics defined in this specification:

ISO 9075	All diagnostics except as specified below.
ISO / IEC 9579-1	Diagnostics beginning with ’**HZ**0’, ’**HZ**1’ and ’**HZ**2’.
ISO / IEC 9579-2	Diagnostics beginning with ’**HZ**3’.
ISO 8649	Diagnostics beginning with ’**HZ**41’, ’**HZ**42’ and ’**HZ**43’.
ISO / IEC 10026-2	Diagnostics beginning with ’**HZ**45’, ’**HZ**46’, ’**HZ**47’, ’**HZ**48’, ’**HZ**49’ and ’**HZ**4A’.
X / OPEN SQL	All subclasses that begin with ’S’.

If the subclass portion of RETURNED_SQLSTATE is implementation-defined, then SUBCLASS_ORIGIN is an implementation-defined value that does not begin with either ’ISO’ or ’X / OPEN’.

TABLE_NAME
A varying-length character string of maximum length 254. In exceptions that pertain to a table, this is the name of the table.

DIAGNOSTICS
The exception requested by the GET DIAGNOSTICS EXCEPTION form must be one of the exceptions that exists in the diagnostics area; *exception-number* must be in the range from 1 through the value of NUMBER (’**35**000’).

Chapter 6

Information Schema

System views are read-only views from which users can retrieve information about any objects to which they have access. This information also describes the system views themselves.

All system views are part of the schema INFO_SCHEM and are accessible to every database user. However, in some cases, certain users are not entitled to perceive the complete system view. These cases are described for each system view in the **CONSTRAINTS** section.

Normally, a view is based on another (underlying) table or view. However, X/Open does not define the basis of system views; indeed, X/Open expects the underlying data structures to vary among implementations. Instead, the following sections define the system views in terms of a hypothetical CREATE TABLE statement. This illustrates the names and attributes of all mandatory columns in the system view. Implementations map their system information into this form as they see fit.

The information schema INFO_SCHEM may contain additional system views. An application obtains information on the existence and structure of any additional system views by examining the COLUMNS, TABLES and VIEWS system views.

Any system views may contain additional columns. Therefore, applications should not apply forms such as SELECT * or SELECT *table-name*.* to a system view, since the result may vary among implementations.

Although the maximum length of user-defined names in X/Open SQL is 18 characters, fields in the system views that contain names are defined as VARCHAR(128) for compatibility with the International Standard.

Object Names in the System Views

The system views that describe objects (the TABLES, COLUMNS, INDEXES, SCHEMATA, TABLE_PRIVILEGES, COLUMN_PRIVILEGES, USAGE_PRIVILEGES and VIEWS system views) contain columns that specify the catalog and schema in which the object resides.

As described in Section 3.1.3 on page 35, some implementations do not support catalog names. In these cases, columns that specify catalog (whose names end in _CAT) contain a zero-length string. An application can determine whether the implementation supports catalog names by examining the CATALOG_NAME attribute in the SERVER_INFO system view (see *SERVER_INFO* on page 166).

Deprecated Schema

The previous issue of this specification specified a schema named INFORMATION_SCHEMA containing system views with column names that differed from those selected by standards organisations in order to adhere to the 18-character limit. On final adoption, the International Standard selected different names, some of which exceed 18 characters.

DE A schema is available that contains system views with columns named as they were in the previous issue. The name of this schema is OLD_INFO_SCHEMA. For the four system views that were defined in the previous issue, this specification indicates the former names of relevant columns.

OLD_INFO_SCHEMA is deprecated. X/Open intends to remove it from future issues of this specification. Application programs should change schema and column names to match the International Standard.

NAME

CHARACTER_SETS — Identify the character sets and their default collations that are accessible for a given user.

CONCEPTUAL DEFINITION

```
CREATE TABLE CHARACTER_SETS (
    CHARSET_CAT        VARCHAR(128) NOT NULL,
    CHARSET_SCHEM      VARCHAR(128) NOT NULL,
    CHARSET_NAME       VARCHAR(128) NOT NULL,
    FORM_OF_USE        VARCHAR(128) NOT NULL,
    NUM_CHARS          INTEGER NOT NULL,
    DEF_COLLATE_CAT    VARCHAR(254) NOT NULL,
    DEF_COLLATE_SCHEM  VARCHAR(254) NOT NULL,
    DEF_COLLATE_NAME   VARCHAR(254) NOT NULL,
    REMARKS            VARCHAR(254))
```

DESCRIPTION

The CHARACTER_SETS system view contains a row for each character set. The row contains at least these columns:

CHARSET_CAT	The name of the catalog that contains the character set in question.
CHARSET_SCHEM	The name of the schema that contains the character set in question.
CHARSET_NAME	The name of the character set in question.
FORM_OF_USE	A user-defined name that indicates the form-of-use of the character set.
NUM_CHARS	The number of characters in the character set.
DEF_COLLATE_CAT	The name of the catalog that contains the default collation for the character set.
DEF_COLLATE_SCHEM	The name of the schema that contains the default collation for the character set.
DEF_COLLATE_NAME	The name of the default collation for the character set.
REMARKS	May contain descriptive information about the table.

The CHARACTER_SETS system view includes one row describing the character set named SQL_TEXT.

CONSTRAINTS

The current user's privileges, at the time the user gains access to this system view, determine the rows that are visible. The user sees one row describing each character set to which the user has USAGE privilege, or to which PUBLIC has USAGE privilege.

NAME

COLLATIONS — Identify the collations that are accessible for a given user.

CONCEPTUAL DEFINITION

```
CREATE TABLE COLLATIONS (
    COLLATION_CAT    VARCHAR(128) NOT NULL,
    COLLATION_SCHEM  VARCHAR(128) NOT NULL,
    COLLATION_NAME   VARCHAR(128) NOT NULL,
    CHARSET_CAT      VARCHAR(128) NOT NULL,
    CHARSET_SCHEM    VARCHAR(128) NOT NULL,
    CHARSET_NAME     VARCHAR(128) NOT NULL,
    PAD_ATTRIBUTE    VARCHAR(254) NOT NULL,
    REMARKS          VARCHAR(254))
```

DESCRIPTION

The COLLATIONS system view contains a row for each collation. The row contains at least these columns:

COLLATION_CAT	The name of the catalog that contains the collation in question.
COLLATION_SCHEM	The name of the schema that contains the collation in question.
COLLATION_NAME	The name of the collation in question.
CHARSET_CAT	The name of the catalog that contains the character set on which the collation is defined.
CHARSET_SCHEM	The name of the schema that contains the character set on which the collation is defined.
CHARSET_NAME	The name of the character set on which the collation is defined.
PAD_ATTRIBUTE	One of the following values: ’NO PAD’ The collation being described has the **no pad** attribute. ’PAD SPACE’ The collation being described has the **pad space** attribute.
REMARKS	May contain descriptive information about the table.

The COLLATIONS system view includes one row describing the collation named SQL_TEXT. This is defined on the character set named SQL_TEXT and is the default collation for that character set.

CONSTRAINTS

The current user’s privileges, at the time the user gains access to this system view, determine the rows that are visible. The user sees one row describing each collation to which the user has USAGE privilege, or to which PUBLIC has USAGE privilege.

NAME

COLUMN_PRIVILEGES — Identify the privileges on columns of tables in terms of the grantor and grantee.

CONCEPTUAL DEFINITION

```
CREATE TABLE COLUMN_PRIVILEGES (
    GRANTOR         VARCHAR(128) NOT NULL,
    GRANTEE         VARCHAR(128) NOT NULL,
    TABLE_CAT       VARCHAR(128) NOT NULL,
    TABLE_SCHEM     VARCHAR(128) NOT NULL,
    TABLE_NAME      VARCHAR(128) NOT NULL,
    COLUMN_NAME     VARCHAR(128) NOT NULL,
    PRIVILEGE_TYPE  VARCHAR(254) NOT NULL,
    IS_GRANTABLE    VARCHAR(254) NOT NULL,
    REMARKS         VARCHAR(254))
```

DESCRIPTION

The COLUMN_PRIVILEGES system view describes each case where a column privilege was granted.

Within each row, there are the following columns:

GRANTOR	The user name of the user that granted the privilege in question. For rows that describe the implicit privileges that a table's owner has on each column of the table, GRANTOR is '_SYSTEM'.
GRANTEE	The user name of the user to whom the privilege in question was granted. Granting a privilege to PUBLIC results in only one row in the COLUMN_PRIVILEGES view (per privilege granted) and the GRANTEE column contains the value PUBLIC.
TABLE_CAT	The name of the catalog that contains the table in question.
TABLE_SCHEM	The name of the schema that contains the table in question.
TABLE_NAME	The name of the table in question.
COLUMN_NAME	The name of the column in question.
PRIVILEGE_TYPE	The type of column privilege that was granted. Its value is 'INSERT', 'REFERENCES', 'SELECT' or 'UPDATE'. Granting multiple privileges to a given user results in multiple rows in the COLUMN_PRIVILEGES view, one row describing each privilege.
IS_GRANTABLE	This column contains the value 'YES' if the grantor specified WITH GRANT OPTION when granting the privilege, or 'NO' otherwise.
REMARKS	May contain descriptive information about the table.

COLUMN_PRIVILEGES includes one row for each privilege granted to each user or to PUBLIC on each column (whether or not the specified privilege can be granted or revoked on individual columns). An exception is that COLUMN_PRIVILEGES does not contain information on the DELETE privilege, which in no case applies to individual columns.

COLUMN_PRIVILEGES includes rows that represent the implicit privileges that a table's owner has on each column of the table.

CONSTRAINTS

Each user sees only rows from the COLUMN_PRIVILEGES view where one of the following is true:

- The GRANTOR column contains the name of the current user.
- The GRANTEE column contains the name of the current user or PUBLIC.

APPLICATION USAGE

Granting a privilege on an entire table implicitly grants privileges on each column (as well as on future columns), each of which is visible in the COLUMN_PRIVILEGES system view.

NAME

COLUMNS — Identify the columns that are accessible for a given user.

CONCEPTUAL DEFINITION

```
CREATE TABLE COLUMNS (
    TABLE_CAT          VARCHAR(128) NOT NULL,
    TABLE_SCHEM        VARCHAR(128) NOT NULL,
    TABLE_NAME         VARCHAR(128) NOT NULL,
    COLUMN_NAME        VARCHAR(128) NOT NULL,
    ORDINAL_POSITION   INTEGER NOT NULL,
    COLUMN_DEF         VARCHAR(254),
    IS_NULLABLE        VARCHAR(254) NOT NULL,
    DATA_TYPE          VARCHAR(254) NOT NULL,
    CHAR_MAX_LENGTH    INTEGER,
    CHAR_OCTET_LENGTH  INTEGER,
    NUM_PREC           INTEGER,
    NUM_PREC_RADIX     INTEGER,
    NUM_SCALE          INTEGER,
    DATETIME_PREC      INTEGER
    INTERVAL_TYPE      VARCHAR(128),
    INTERVAL_PREC      INTEGER,
    CHAR_SET_CAT       VARCHAR(128),
    CHAR_SET_SCHEM     VARCHAR(128),
    CHAR_SET_NAME      VARCHAR(128),
    COLLATION_CAT      VARCHAR(128),
    COLLATION_SCHEM    VARCHAR(128),
    COLLATION_NAME     VARCHAR(128),
    REMARKS            VARCHAR(254))
```

DESCRIPTION

The COLUMNS system view contains a row for each column accessible to the current user. The row contains at least these columns:

TABLE_CAT	The name of the catalog containing TABLE_NAME.
TABLE_SCHEM	The name of the schema containing TABLE_NAME.
TABLE_NAME	The name of the table or view.
COLUMN_NAME	The name of the column of the specified table or view.
ORDINAL_POSITION	The ordinal position of the column in the table. The first column in the table is number 1.
COLUMN_DEF	The column's default value, using legal syntax for *default-value* in the *column-definition* of the CREATE TABLE or ALTER TABLE statement. If the default value is a character string, then this column is that string enclosed in single quotes. If the default value is a numeric literal, then this column contains the original character representation with no enclosing single quotes. If the default value is a date-time literal, then this column contains the appropriate keyword (DATE, TIME or TIMESTAMP) followed by the *date-value* and/or *time-value* enclosed in single quotes. If the default value is a *pseudo-literal*, then this column contains the keyword, such as CURRENT_DATE, with no enclosing single quotes.

	If NULL was specified as the default value, then this column is the word NULL, not enclosed in quotes. If the default value cannot be represented without truncation, then this column contains TRUNCATED with no enclosing single quotes. If no default value was specified, then its value is null. The value of COLUMN_DEF is suitable for use in generating a new *column-definition*, except when it contains the value TRUNCATED.
IS_NULLABLE	Contains the value ’NO’ if the column is known to be not nullable, according to the rules in the International Standard; and ’YES’ otherwise.
DATA_TYPE	Identifies the type of the column and can contain one of the following values: CHARACTER CHARACTER VARYING DATE DECIMAL DOUBLE PRECISION FLOAT INTEGER INTERVAL NUMERIC REAL SMALLINT TIME TIMESTAMP
CHAR_MAX_LENGTH	Contains the maximum length in characters for a character data type column. For all other data types it is null.
CHAR_OCTET_LENGTH	Contains the maximum length in octets for a character data type column. For all other data types it is null. (For single-octet character sets, this is the same as CHAR_MAX_LENGTH.)
NUM_PREC	If DATA_TYPE is an approximate numeric data type, this column contains the number of bits of mantissa precision of the column. For exact numeric data types, this column contains the total number of decimal digits allowed in the column. Otherwise, its value is null. NUM_PREC_RADIX (see below) indicates the units of measurement.
NUM_PREC_RADIX	If DATA_TYPE is an approximate numeric data type, this column contains the value 2 because NUM_PREC specifies a number of bits. For exact numeric data types, this column contains the value 10 because NUM_PREC specifies a number of decimal digits. Otherwise, its value is null. By combining the precision with the radix, an application can calculate the maximum number that the column can hold.
NUM_SCALE	Defines the total number of significant digits to the right of the decimal point. For the INTEGER and SMALLINT data types, it is 0. For the CHARACTER, CHARACTER VARYING, DATETIME, FLOAT, INTERVAL, REAL and DOUBLE PRECISION data types, its value is null.

DATETIME_PREC	For date/time and interval data types, this column contains the number of digits of precision of the fractional seconds component. For other data types, its value is null.
INTERVAL_TYPE	For interval data types, this column contains a character string with text of an *interval-qualifier* specifying the interval precision (for example, 'HOUR TO MINUTE'). For other data types, its value is null.
INTERVAL_PREC	This column contains the number of significant digits for the leading precision of interval data types. For all other data types, its value is null.
CHAR_SET_CAT	The name of the catalog that contains the character set used by the column in question.
CHAR_SET_SCHEM	The name of the schema that contains the character set used by the column in question.
CHAR_SET_NAME	The name of the character set used by the column in question.
COLLATION_CAT	The name of the catalog that contains the collation used by the column in question.
COLLATION_SCHEM	The name of the schema that contains the collation used by the column in question.
COLLATION_NAME	The name of the collation used by the column in question.
REMARKS	May contain descriptive information about the column.

CONSTRAINTS

The current user's privileges, at the time the user gains access to the COLUMNS system view, determine the rows that are visible.

- If the user has one or more privileges (INSERT, DELETE or SELECT) on the table, or if PUBLIC has such privileges, then the user sees one row describing each column in that table.
- If the user or PUBLIC has REFERENCES or UPDATE privilege on one or more columns, then the user sees one row for each such column.

COLUMN NAMES IN OLD_INFO_SCHEM

DE The COLUMNS system view exists in the schema OLD_INFO_SCHEM for compatibility with the previous issue of this specification. In this schema:

It is implementation-defined whether the following columns are present: TABLE_CAT, ORDINAL_POSITION, COLUMN_DEF, CHAR_OCTET_LENGTH, CHAR_SET_CAT, CHAR_SET_SCHEM, CHAR_SET_NAME, COLLATION_CAT, COLLATION_SCHEM, COLLATION_NAME, DATETIME_PREC, INTERVAL_TYPE.

TABLE_SCHEM is called TABLE_SCHEMA.
NUM_PREC is called NUMERIC_PRECISION.
NUM_PREC_RADIX is called NUMERIC_PREC_RADIX.
NUM_SCALE is called NUMERIC_SCALE.

All other columns are present with the same names as shown above.

APPLICATION USAGE

The COLUMNS system view describes the data type and type attributes of table columns. It includes information available in the SQL descriptor of dynamic SQL, but for historical reasons, the column names do not match the field names of the SQL descriptor.

Information on various data types is available in the following columns:

Interval
The *interval-qualifier* is in INTERVAL_TYPE.
The fractional seconds precision is in DATETIME_PREC.
The leading precision is in INTERVAL_PREC.

Date/time
The *interval-qualifier* is in INTERVAL_TYPE.
The fractional seconds precision is in DATETIME_PREC.

Numeric
The precision is in NUM_PREC.
The radix of the precision is in NUM_PREC_RADIX.
The scale is in NUM_SCALE.

NAME

INDEXES — Identify the indexes that are accessible to a given user.

CONCEPTUAL DEFINITION

```
CREATE TABLE INDEXES (
    TABLE_CAT         VARCHAR(128) NOT NULL,
    TABLE_SCHEM       VARCHAR(128) NOT NULL,
    TABLE_NAME        VARCHAR(128) NOT NULL,
    COLUMN_NAME       VARCHAR(128) NOT NULL,
    INDEX_NAME        VARCHAR(128) NOT NULL,
    ORDINAL_POSITION  INTEGER NOT NULL,
    NON_UNIQUE        VARCHAR(128) NOT NULL,
    ASC_OR_DESC       CHAR(1) NOT NULL,
    REMARKS           VARCHAR(254))
```

DESCRIPTION

The INDEXES system view describes each index accessible to the current user. It contains one row for each index column. The row contains at least these columns:

TABLE_CAT	The name of the catalog containing TABLE_NAME.
TABLE_SCHEM	The name of the schema containing TABLE_NAME.
TABLE_NAME	The name of the base table.
COLUMN_NAME	The name of the column of the specified base table and index.
INDEX_NAME	The unique name of the index.
ORDINAL_POSITION	The ordinal number of the column in the index. The ordering is determined by the order of the columns in the CREATE INDEX statement (see Section 5.3.5 on page 105). The numbering of the columns of the index starts from 1 and increases contiguously.
NON_UNIQUE	Contains the value 'NO' if at most one row is allowed in TABLE_NAME for each combination of values in the specified columns for this index; and 'YES' otherwise.
ASC_OR_DESC	Contains the value 'A' if the order of the referenced column is ascending and 'D' if the order of the referenced column is descending.
REMARKS	May contain descriptive information about the index.

CONSTRAINTS

The current user's privileges, at the time the user gains access to the INDEXES system view, determine the rows that are visible. If the user has SELECT privilege on a given table, or if PUBLIC has SELECT privilege, then the user sees one row describing each index column created on that table.

NAME

SCHEMATA — Describe the schemata of the database.

CONCEPTUAL DEFINITION

```
CREATE TABLE SCHEMATA (
    CAT_NAME            VARCHAR(128) NOT NULL,
    SCHEM_NAME          VARCHAR(128) NOT NULL,
    SCHEM_OWNER         VARCHAR(128) NOT NULL,
    DEF_CHAR_SET_CAT    VARCHAR(128),
    DEF_CHAR_SET_SCHEM  VARCHAR(128),
    DEF_CHAR_SET_NAME   VARCHAR(128),
    REMARKS             VARCHAR(254))
```

DESCRIPTION

The SCHEMATA system view contains one row describing each schema of the database. Within each row, there are the following columns:

CAT_NAME	The name of the catalog that contains the schema in question.
SCHEM_NAME	The name of the schema in question.
SCHEM_OWNER	A user name indicating the owner of the schema in question.
DEF_CHAR_SET_CAT	The name of the catalog that contains the default character set for the schema in question.
DEF_CHAR_SET_SCHEM	The name of the schema that contains the default character set for the schema in question.
DEF_CHAR_SET_NAME	The name of the default character set for the schema in question.
REMARKS	May contain descriptive information about the table.

CONSTRAINTS

None; the SCHEMATA system view is always completely visible to all users.

NAME

SERVER_INFO — Provide attributes of the current database system or server.

CONCEPTUAL DEFINITION

```
CREATE TABLE SERVER_INFO (
    SERVER_ATTRIBUTE  VARCHAR(254) NOT NULL,
    ATTRIBUTE_VALUE   VARCHAR(254))
```

DESCRIPTION

The SERVER_INFO system view describes the server the application is currently connected to. Each row provides information about one attribute. The SERVER_ATTRIBUTE column identifies an attribute of servers; the ATTRIBUTE_VALUE specifies the value of that attribute as it applies to the current server.

All X/Open-compliant database systems must provide applications with certain minimum information about any server. That is, for all the values of SERVER_ATTRIBUTE listed below, the SERVER_INFO system view must contain one row with that value, describing that attribute of the current server.

Implementations may define additional attributes and their corresponding values. For these attributes, the name of the SERVER_ATTRIBUTE, the range of values in ATTRIBUTE_VALUE, and the resulting interpretation are implementation-defined.

Here are the attributes that SERVER_INFO must always specify:

CATALOG_NAME	This attribute is 'YES' if the server supports catalog names, and 'NO' otherwise (see Section 3.1.3 on page 35).
COLLATION_SEQ	This attribute is the assumed ordering of the character set for this server. Acceptable values are 'ISO 8859-1' and 'EBCDIC'.
IDENTIFIER_LENGTH	This attribute is the maximum number of characters for a user-defined name. The value is the character string representation of the decimal value.
INTERVAL_FRACTIONAL_PRECISION	This attribute is the default fractional precision for all objects of data type INTERVAL SECOND, INTERVAL MINUTE TO SECOND, INTERVAL HOUR TO SECOND or INTERVAL DAY TO SECOND.
INTERVAL_LEADING_PRECISION	This attribute is the default leading precision for all intervals (see Section 7.1 on page 175).
ROW_LENGTH	This attribute is the maximum size of a row, using the units defined in Section 7.1 on page 175. The value is the character string representation of the decimal value.
TIME_PRECISION	This attribute is the default fractional precision for all objects whose data type is date/time and whose subtype is TIME (see Section 7.1 on page 175).
TIMESTAMP_PRECISION	This attribute is the default fractional precision for all objects whose data type is date/time and whose subtype is TIMESTAMP (see Section 7.1 on page 175).

TXN_ISOLATION	This attribute is the initial transaction isolation level the server assumes. The value must be one of the following, corresponding to isolation levels defined in the International Standard: READ UNCOMMITTED READ COMMITTED REPEATABLE READ SERIALIZABLE
USERID_LENGTH	This attribute is the maximum number of characters of a user name (or ‘‘authorisation identifier’’). The value is the character string representation of the decimal value.

CONSTRAINTS

None; the SERVER_INFO system view is always completely visible to all users.

COLUMN NAMES IN OLD_INFO_SCHEM

DE The SERVER_INFO system view exists in the schema OLD_INFO_SCHEM for compatibility with the previous issue of this specification. It is defined exactly as shown above. However, this issue defines new server attributes, which are accessible only using the INFO_SCHEM schema name.

NAME

SQL_LANGUAGES — List the SQL standards and SQL dialects that are supported.

CONCEPTUAL DEFINITION

```
CREATE TABLE SQL_LANGUAGES (
    SOURCE             VARCHAR(254) NOT NULL,
    SOURCE_YEAR        VARCHAR(254),
    CONFORMANCE        VARCHAR(254),
    INTEGRITY          VARCHAR(254),
    IMPLEMENTATION     VARCHAR(254),
    BINDING_STYLE      VARCHAR(254),
    PROGRAMMING_LANG   VARCHAR(254))
```

DESCRIPTION

The SQL_LANGUAGES system view contains a row for every conformance claim the SQL product makes (including subsets defined for ISO and vendor-specific versions). Rows defining ISO standard and vendor-specific languages may exist in the same table. Each row has at least these columns and, if it makes an X/Open SQL conformance claim, the columns contain these values:

Column Name	Meaning	X/Open Values
SOURCE	The organisation that defined this SQL version.	’X/OPEN SQL’
SOURCE_YEAR	The year the relevant source document was approved.	’1992’
CONFORMANCE	The conformance level to the relevant document that the implementation claims.	null
INTEGRITY	(Meaning no longer specified.)	’YES’
IMPLEMENTATION	A character string, defined by the vendor, that uniquely identifies the vendor’s SQL product.	null
BINDING_STYLE	A column included to envisage future adoption of direct, module or other binding styles.	’EMBEDDED’ or ’CLI’
PROGRAMMING_LANG	The host language for which the binding style is supported.	’C’ or ’COBOL’

Rows that do not make an X/Open SQL conformance claim may have additional columns, as specified by the respective vendor or international standard.

CONSTRAINTS

None; the SQL_LANGUAGES system view is always completely visible to all users.

COLUMN NAMES IN OLD_INFO_SCHEM

DE The SQL_LANGUAGES system view exists in the schema OLD_INFO_SCHEM for compatibility with the previous issue of this specification. It is defined exactly as shown above.

NAME

TABLE_PRIVILEGES — Identify the privileges on tables in terms of the grantor and grantee.

CONCEPTUAL DEFINITION

```
CREATE TABLE TABLE_PRIVILEGES (
    GRANTOR         VARCHAR(128) NOT NULL,
    GRANTEE         VARCHAR(128) NOT NULL,
    TABLE_CAT       VARCHAR(128) NOT NULL,
    TABLE_SCHEM     VARCHAR(128) NOT NULL,
    TABLE_NAME      VARCHAR(128) NOT NULL,
    PRIVILEGE_TYPE  VARCHAR(254) NOT NULL,
    IS_GRANTABLE    VARCHAR(254) NOT NULL,
    REMARKS         VARCHAR(254))
```

DESCRIPTION

The TABLE_PRIVILEGES system view contains one row describing each case where a privilege was granted on an entire table.

TABLE_PRIVILEGES includes rows that represent the implicit privileges that a table's owner has on the table.

Within each row, there are the following columns:

GRANTOR	The user name of the user that granted the privilege in question. For rows that describe the implicit privileges that a table's owner has on the table, GRANTOR is '_SYSTEM'.
GRANTEE	The user name of the user to whom the privilege in question was granted. Granting a privilege to PUBLIC results in only one row in the TABLE_PRIVILEGES view (per privilege granted) and the GRANTEE column contains the value PUBLIC.
TABLE_CAT	The name of the catalog that contains the table in question.
TABLE_SCHEM	The name of the schema that contains the table in question.
TABLE_NAME	The name of the table in question.
PRIVILEGE_TYPE	The type of privilege that was granted. Its value must be one of the following: DELETE, INSERT, REFERENCES, SELECT, UPDATE. Rows where PRIVILEGE_TYPE is REFERENCES or UPDATE describe only the cases where a user was granted the respective privileges to the entire table. When the grantor's GRANT statement granted REFERENCES or UPDATE privileges to specified columns of a table, there are no such rows in TABLE_PRIVILEGES but there are rows in COLUMN_PRIVILEGES. Granting multiple privileges to a given user results in multiple rows in the TABLE_PRIVILEGES view, one row describing each privilege.
IS_GRANTABLE	This column contains the value 'YES' if the grantor specified WITH GRANT OPTION when granting the privilege, or 'NO' otherwise.

REMARKS	May contain descriptive information about the table.

CONSTRAINTS

Each user sees only rows from the TABLE_PRIVILEGES view where one of the following is true:

- The GRANTOR column contains the name of the current user.
- The GRANTEE column contains the name of the current user or PUBLIC.

APPLICATION USAGE

An entry in TABLE_PRIVILEGES does not assert that the grantee has the privilege on any specific column of the table, because privileges can be granted on a table and revoked on individual columns. An application should refer to the COLUMN_PRIVILEGES table for privileges to specific columns.

NAME

TABLES — Identify the tables accessible to the current user.

CONCEPTUAL DEFINITION

```
CREATE TABLE TABLES (
    TABLE_CAT    VARCHAR(128) NOT NULL,
    TABLE_SCHEM  VARCHAR(128) NOT NULL,
    TABLE_NAME   VARCHAR(128) NOT NULL,
    TABLE_TYPE   VARCHAR(254) NOT NULL,
    REMARKS      VARCHAR(254))
```

DESCRIPTION

The TABLES system view contains exactly one row for each table to which the current user has access (see **Constraints** below). The row contains at least these columns:

TABLE_CAT	The name of the catalog containing TABLE_NAME.
TABLE_SCHEM	The name of the schema containing TABLE_NAME.
TABLE_NAME	The name of the table or view.
TABLE_TYPE	Identifies the type of the table or view. It can have two possible values: 'BASE TABLE' or 'VIEW'.
REMARKS	May contain descriptive information about the table.

CONSTRAINTS

The current user's privileges, at the time the user gains access to the TABLES system view, determine the rows that are visible. If the user has one or more privileges (INSERT, DELETE or SELECT; or REFERENCES or UPDATE on one or more columns) on a given table, or if PUBLIC has such privileges, then the user sees one row describing that table.

COLUMN NAMES IN OLD_INFO_SCHEM

DE The TABLES system view exists in the schema OLD_INFO_SCHEM for compatibility with the previous issue of this specification. In this schema:

It is implementation-defined whether TABLE_CAT is present.
TABLE_SCHEM is called TABLE_SCHEMA.
All other columns are present with the same names as shown above.

NAME

TRANSLATIONS — Identify the translations that are accessible for a given user.

CONCEPTUAL DEFINITION

```
CREATE TABLE TRANSLATIONS (
    TRANSLATION_CAT    VARCHAR(128) NOT NULL,
    TRANSLATION_SCHEM  VARCHAR(128) NOT NULL,
    TRANSLATION_NAME   VARCHAR(128) NOT NULL,
    SRC_CHARSET_CAT    VARCHAR(128) NOT NULL,
    SRC_CHARSET_SCHEM  VARCHAR(128) NOT NULL,
    SRC_CHARSET_NAME   VARCHAR(128) NOT NULL,
    TGT_CHARSET_CAT    VARCHAR(128) NOT NULL,
    TGT_CHARSET_SCHEM  VARCHAR(128) NOT NULL,
    TGT_CHARSET_NAME   VARCHAR(128) NOT NULL,
    REMARKS            VARCHAR(254))
```

DESCRIPTION

The TRANSLATIONS system view contains a row for each translation. There are not necessarily any translations.

Each row identifies a translation and specifies a source and target character sets. The source character set is the character set to which characters belong that are to be translated by the translation in question. The target character set is the character set to which characters belong that are the result of the translation in question.

The row contains at least these columns:

TRANSLATION_CAT	The name of the catalog that contains the translation in question.
TRANSLATION_SCHEM	The name of the schema that contains the translation in question.
TRANSLATION_NAME	The name of the translation in question.
SRC_CHARSET_CAT	The name of the catalog that contains the source character set.
SRC_CHARSET_SCHEM	The name of the schema that contains the source character set.
SRC_CHARSET_NAME	The name of the source character set.
TGT_CHARSET_CAT	The name of the catalog that contains the target character set.
TGT_CHARSET_SCHEM	The name of the schema that contains the target character set.
TGT_CHARSET_NAME	The name of the target character set.
REMARKS	May contain descriptive information about the table.

CONSTRAINTS

The current user's privileges, at the time the user gains access to this system view, determine the rows that are visible. The user sees one row describing each translation to which the user has USAGE privilege, or to which PUBLIC has USAGE privilege.

NAME

USAGE_PRIVILEGES — Identify the privileges on character sets and collations.

CONCEPTUAL DEFINITION

```
CREATE TABLE USAGE_PRIVILEGES (
    GRANTOR         VARCHAR(128) NOT NULL,
    GRANTEE         VARCHAR(128) NOT NULL,
    OBJECT_CAT      VARCHAR(128) NOT NULL,
    OBJECT_SCHEM    VARCHAR(128) NOT NULL,
    OBJECT_NAME     VARCHAR(128) NOT NULL,
    OBJECT_TYPE     VARCHAR(254) NOT NULL,
    PRIVILEGE_TYPE  VARCHAR(254) NOT NULL,
    IS_GRANTABLE    VARCHAR(254) NOT NULL,
    REMARKS         VARCHAR(254))
```

DESCRIPTION

The USAGE_PRIVILEGES system view contains one row describing each case where a privilege was granted on a character set or a collation. Within each row, there are the following columns:

GRANTOR	The user name of the user that granted the privilege in question.
GRANTEE	The user name of the user to whom the privilege in question was granted. Granting a privilege to PUBLIC results in only one row in the USAGE_PRIVILEGES view (per privilege granted) and the GRANTEE column contains the value PUBLIC.
OBJECT_CAT	The name of the catalog that contains the object in question.
OBJECT_SCHEM	The name of the schema that contains the object in question.
OBJECT_NAME	The name of the object in question.
OBJECT_TYPE	The type of the object. Its value is 'CHARACTER SET' or 'COLLATION'.
PRIVILEGE_TYPE	The type of privilege that was granted. Its value must be 'USAGE'.
IS_GRANTABLE	This column contains the value 'YES' if the grantor specified WITH GRANT OPTION when granting the privilege, or 'NO' otherwise.
REMARKS	May contain descriptive information about the table.

CONSTRAINTS

Each user sees only rows from the USAGE_PRIVILEGES view where one of the following is true:

- The GRANTOR column contains the name of the current user.
- The GRANTEE column contains the name of the current user or PUBLIC.

NAME

VIEWS — Identify the viewed tables accessible to the current user.

CONCEPTUAL DEFINITION

```
CREATE TABLE VIEWS (
    TABLE_CAT        VARCHAR(128) NOT NULL,
    TABLE_SCHEM      VARCHAR(128) NOT NULL,
    TABLE_NAME       VARCHAR(128) NOT NULL,
    VIEW_DEFINITION  VARCHAR(implementation-defined),
    CHECK_OPTION     VARCHAR(254),
    IS_UPDATABLE     VARCHAR(254),
    REMARKS          VARCHAR(254))
```

DESCRIPTION

The VIEWS system view contains exactly one row for each viewed table to which the current user has access (see **Constraints** below). The row contains at least these columns:

TABLE_CAT	The name of the catalog containing TABLE_NAME.
TABLE_SCHEM	The name of the schema containing TABLE_NAME.
TABLE_NAME	The name of the table or view.
VIEW_DEFINITION	The definition of the view as it would appear in a CREATE VIEW statement. If the actual definition would not fit in the implementation-defined maximum length of this column, then the implementation sets this column to the null value.
CHECK_OPTION	Contains the value 'CASCADED' if WITH CHECK OPTION was specified in the CREATE VIEW statement that created the table, and the value 'NONE' otherwise.
IS_UPDATABLE	Contains the value 'YES' if the table is updatable and 'NO' otherwise.
REMARKS	May contain descriptive information about the view.

CONSTRAINTS

The current user's privileges, at the time the user gains access to the VIEWS system view, determine the rows that are visible. If the user has one or more privileges (INSERT, DELETE or SELECT; or REFERENCES or UPDATE on one or more columns) on a given view, or if PUBLIC has such privileges, then the user sees one row describing that view.

APPLICATION USAGE

Views are also listed in the TABLES system view.

Chapter 7

Implementation-specific Issues

7.1 Limits

Limits vary among implementations. This section defines the maximum values for certain limits that application developers may safely assume all X/Open-compliant systems support. For example, X/Open is aware of implementations where the limit on the length of a character string is as low as 254 or as high as 4096. X/Open specifies (in 7.1.1 (103) below) a portability limit of 254. This asserts that all X/Open-compliant implementations shall set any such limit at 254 or higher, and asserts that portable applications must conform to a limit of 254.

Implementations are not required to have any of the following limits. X/Open requires only that any such limits be set to at least the value specified by X/Open.

The tables in the following sections contain columns for X/Open, SPIRIT Issue 2 and SPIRIT Issue 3. (A full definition of the SPIRIT SQL profiles is provided in Appendix D and Appendix E, respectively.) Limits that affect only the SPIRIT specifications appear in Section 7.1.5 through Section 7.1.7.

The guaranteed minimum values are specified for limits on ‘‘attributes’’, a term used in this section as a generic reference to various aspects of an SQL implementation. The legend N/A (not applicable) means that the respective SQL specification does not guarantee a minimum value of any limit on the given attribute.

7.1.1 Supplementary Definitions

The International Standard specifies that a variety of parameters are implementation-defined. To facilitate portability, X/Open specifies values for some of these parameters:

		Guaranteed Minimum Value		
Ref.	**Attribute**	**X/Open**	**SPIRIT 2**	**SPIRIT 3**
101	Length of a user-defined name (SPIRIT: identifier)	18	18	18
102	Length of a national character identifier	18	8	8
103	Length of a character string	254 [1]	254	254
104	Maximum length of a variable-length character string	254 [1]	4000	32000
105	Length of a NATIONAL CHARACTER string	254	127	127
106	Maximum length of a NATIONAL CHARACTER VARYING string	N/A	2000	16000
107	In the ALLOCATE DESCRIPTOR statement, the maximum value and the default value for *occurrences* (in SPIRIT, this is a derived limit; see Section 7.1.6 on page 179)	100	see text	see text
108	Precision of any DECIMAL or NUMERIC data type, in decimal digits	15 [2]	15	15

		Guaranteed Minimum Value		
Ref.	**Attribute**	**X/Open**	**SPIRIT 2**	**SPIRIT 3**
109	Mantissa precision of any FLOAT data type, in bits	47 [2]	47	47
110	Exponent precision of any FLOAT data type, in bits	N / A	7 [3]	7 [3]
111	Precision of a SMALLINT number	5 digits	15 bits	15 bits
112	Precision of an INTEGER number	10 digits	31 bits	31 bits
113	Mantissa precision of a REAL number, in bits	21	21	23 [4]
114	Exponent precision of a REAL number, in bits	N / A	7 [3]	7 [3]
115	Mantissa precision of a DOUBLE PRECISION number, in bits	47	47	47
116	Exponent precision of a DOUBLE PRECISION number, in bits	N / A	7 [3]	7 [3]
117	Precision of SQLCODE	9 digits	31 bits	31 bits
	Fractional precision, in decimal digits, of the seconds component of:			
118	values of type TIME	6	N / A	0
119	values of type TIMESTAMP	6	N / A	6
120	values of type INTERVAL	6	N / A	6
121	Leading precision for INTERVAL data types, in decimal digits	7	N / A	7
122	Fractional precision for INTERVAL data types that have a SECOND field	6	N / A	6
123	Length of an item of type BIT	N / A	N / A	8000
124	Maximum length of an item of type BIT VARYING	N / A	N / A	32000

Notes:

[1] Character strings that serve as server names or connection names in connection statements (see Section 5.7 on page 142) are limited to 128 characters.

[2] X/Open leaves implementation-defined the default precision for these data types, as does the International Standard. X/Open advises portable applications to always code a precision explicitly for these data types.

[3] The range of positive values that an implementation must support for REAL is 1.0×10^{-37} to $1.0 \times 10^{+38}$. The range of negative values for each is -1.0×10^{-37} to $-1.0 \times 10^{+38}$. The range of positive values that an implementation must support for FLOAT and DOUBLE PRECISION is 1.0×10^{-78} to $1.0 \times 10^{+75}$. The range of negative values for each is -1.0×10^{-78} to $-1.0 \times 10^{+75}$.

The value 1.0×10^{-37} cited above deviates from that specified by the X/Open definition of the C language. X/Open is working with SPIRIT to resolve this.

[4] The difference in this value between SPIRIT 2 and SPIRIT 3 derives from the publication of FIPS 127-2.

The SERVER_INFO system view provides information on the limits applicable to the current server (see *SERVER_INFO* on page 166).

7.1.2 Additional Limits

This section specifies limits that the International Standard does not contemplate.

Ref.	Attribute	Guaranteed Minimum Value		
		X/Open	**SPIRIT 2**	**SPIRIT 3**
201	Number of tables in a catalog	N/A	2047	10000
202	Number of columns in a table (for SPIRIT, other limits are derived from this limit; see Section 7.1.6 on page 179)	100	255	255
203	Number of columns constituting an index	6	N/A	N/A
204	Number of tables a statement references [5]	10	15	15
205	Number of tables a FROM clause references [5]	N/A	15	15
206	Number of cursors simultaneously open	10	N/A	N/A
207	Number of columns that a single INSERT statement can insert	20	N/A	N/A
208	Number of columns that a single UPDATE statement can update	20	N/A	N/A
209	Number of sub-queries in an SQL statement	N/A	14	14
210	Levels of nesting of sub-queries, not counting the outermost level	9	9	9
211	Predicates in a WHERE or HAVING clause	N/A	255	750
212	Number of columns in a UNIQUE constraint	N/A	16	16
213	Number of columns in a GROUP BY column list	N/A	16	16
214	Number of sort items in an ORDER BY clause	N/A	16	16
215	Number of *referencing-column*s in a FOREIGN KEY	N/A	16	16
216	Number of columns in a named-column JOIN	N/A	16	16

Note:

[5] X/Open specifies ‘‘direct or indirect’’ references. SPIRIT specifies that the number of table references is the sum of: the number of views and base tables named in the statement, the number of underlying views and tables (see Subclause 4.9, "Tables", of the International Standard) for each derived table or cursor, and the number of <correlation name>s (either given in the SQL statement or contained in some view named in the SQL statement) not directly associated with a named table or view.

7.1.3 Storage Capacity

Various implementations limit the size and complexity of rows, and of other groupings of columns, based on their internal storage method. The International Standard does not specify such limits. As a common basis of measurement, X/Open assumes a storage model in which:

- The existence of a column requires 2 units of storage.
- Each character column requires a number of storage units equal to its maximum length.
- Each exact numeric column requires the following storage units:
  - **X/Open**: A number equal to its decimal precision.
  - **SPIRIT**: A number one greater than its decimal precision.
- Each approximate numeric column requires the following storage units:
  - **X/Open**: A number equal to its precision.
  - **SPIRIT**: A number one greater than one-fourth of its binary precision.
- Each date/time column requires a number of storage units equal to the number of characters in its visible representation. This is defined in Table 3-4 on page 43.
- Each interval column requires the following storage units:
  - **X/Open**: A number equal to the number of characters in its visible representation. This is defined in **Length of an Interval** on page 46.
  - **SPIRIT**: 20 storage units. In addition, if the interval has a non-zero fractional seconds precision, then add that precision plus 1.

An implementation may use other storage models, but it shall permit at least the following complexity as measured by the X/Open storage model:

Ref.	Attribute	Guaranteed Minimum Value		
		X/Open	**SPIRIT 2**	**SPIRIT 3**
301	Total length of a row	2000	4000	32000
302	Total length of an index key	120	N/A	N/A
303	Length of columns specified in a GROUP BY clause	120	250	250
304	Length of columns specified in an ORDER BY clause	120	250	250
305	Length of columns specified in a FOREIGN KEY column list	N/A	250	250
306	Length of columns specified in a JOIN column list	N/A	250	250
307	Length of columns in a UNIQUE constraint	N/A	250	250

Implementations with limited storage capacities may limit the total size or complexity of the database in ways that X/Open cannot characterise.

7.1.4 Statement Complexity

Certain implementation capabilities are guaranteed regarding the length of an SQL statement. SPIRIT defines this length to be the result of applying the OCTET_LENGTH function (see Section 3.9.3 on page 59) to the SQL statement with the SQL statement considered to be an instance of a CHARACTER VARYING data type.

Any limits on the following attributes imposed by compliant implementations shall be at least as follows:

		Guaranteed Minimum Value		
Ref.	**Attribute**	**X/Open**	**SPIRIT 2**	**SPIRIT 3**
401	Length of an SQL <schema definition>	4000	32000	32000
402	Length of an <SQL data statement>	4000	32000	32000
403	Length of an <SQL statement variable>	4000	32000	32000

7.1.5 Embedded Aspects (SPIRIT Only)

Any limits on the following attributes imposed by SPIRIT-compliant implementations shall be at least as follows:

		Guaranteed Minimum Value		
Ref.	**Attribute**	**X/Open**	**SPIRIT 2**	**SPIRIT 3**
501	Number of host variables in an SQL declare section	N / A	1024	1024
502	Number of executable SQL statements in a program (compilation unit)	N / A	512	1000
503	Number of WHENEVER statements in a program (compilation unit)	N / A	128	128
504	Number of cursors in a program (compilation unit)	N / A	255	255

7.1.6 Derived Limits (SPIRIT Only)

Section 7.1 on page 177 guarantees a minimum value for any limit on the number of columns in a table. SPIRIT guarantees the same minimum value as the limit of each of the following:

- the number of columns in a view
- the number of values in an INSERT statement
- the number of SET clauses in an UPDATE statement
- the number of items in a SELECT list
- the maximum value of *occurrences* in an ALLOCATE DESCRIPTOR statement
- the number of item descriptor areas allocated to an SQL descriptor area if *occurrences* is omitted from the ALLOCATE DESCRIPTOR statement.

In addition, any implementation-defined limit on the number of host variables in an SQL statement shall be at least twice the limit of the number of columns in a table.

7.1.7 Language-specific Limits (SPIRIT Only)

In C, any limit on the length of a host identifier is at least 6 if the identifier is an **extern**, and at least 18 otherwise.

In COBOL, any limit on the length of a host identifier is at least 18.

In Fortran, any limit on the length of a host identifier is at least 6.

7.2 Vendor-specific SQL

Implementations often provide SQL functions beyond those documented herein. Many useful functions are not yet standardised.

An **escape clause** is a syntactic mechanism for vendor-specific SQL extensions to be implemented in the framework of standardised SQL. Using escape clauses, an application can request a vendor-specific function in a way that does not keep it from compiling or executing in an environment that does not support the function.

However, any application whose correct operation depends on vendor-specific SQL functions is necessarily restricted in its portability, since an X/Open-compliant SQL implementation need not provide any vendor-specific functions.

7.2.1 Vendor Escape Clause

FUNCTION

Delimit and identify uses of vendor-specific extensions to X/Open SQL.

SYNOPSIS

An *vendor-escape-clause* is defined as:

```
--(* dialect, extended-SQL-text *)--
```

where *dialect* is defined as:

```
ISO-dialect | vendor-dialect
```

and *ISO-dialect* is defined as:

```
YEAR(ISO-year), CONFORMANCE(ISO-conformance)
```

and *vendor-dialect* is defined as:

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

DESCRIPTION

The escape clause introduces SQL syntax not defined in this X/Open specification. This non-standard syntax is the *extended-SQL-text*. The escape clause includes a description of the SQL dialect that the *extended-SQL-text* uses, to allow coherent interpretation even if several dialects should define the same syntax differently.

ISO-based Dialects

The *ISO-dialect* form introduces SQL syntax that a specified dialect that the International Standard defines and that X/Open may not yet support. The *ISO-year* element is the publication year of the standard. Currently, the application must specify YEAR(1992). The *ISO-conformance* element is the conformance level. The application must also specify CONFORMANCE(1) for Entry level conformance, CONFORMANCE(2) for Intermediate level conformance, or CONFORMANCE(3) for Full level conformance.

Vendor Dialects

The *vendor-dialect* form introduces vendor-specific SQL syntax. The *vendor-name* element is a vendor identification that is consistent across all that vendor's SQL implementations. The *product-name* element identifies the SQL version in an implementation-defined way.

Processing

In either type of clause, the *extended-SQL-text* contains all or part of a valid SQL statement in the SQL dialect that the escape clause specifies. The text must be such that the processing specified below produces valid SQL syntax in the applicable SQL dialect. The text must not contain an *sql-prefix*, an *sql-terminator*, a *dynamic-parameter*, or an *embedded-variable-name*.

Processing of an SQL statement that contains a *vendor-escape-clause* is done as follows:

1. If the database system supports *dialect*, then it conceptually replaces the entire *vendor-escape-clause* with the text provided in *extended-SQL-text* of the escape clause. (The result, in context, must be valid SQL syntax in that dialect.)
2. If the database system does not support *dialect*, then it conceptually deletes the entire *vendor-escape-clause*. (The result, in context, must be valid X/Open SQL syntax.)
3. This edited SQL statement is processed in the usual way.[30]

Escape clauses are allowed only in SQL statement text processed by PREPARE and EXECUTE IMMEDIATE.

30. If the entire prepared SQL statement text is a vendor escape clause, then its execution will fail ('**42**000') in implementations that do not support the specified dialect, since null SQL statements are not valid.

7.3 Restrictions on Names

a. Most implementations have reserved words in addition to those listed in this specification (see Section 3.1.6 on page 37). To ensure portability, applications should avoid using identifiers that might be reserved. For example, include underscore characters or digits in the identifiers.

b. To avoid name conflicts between system-defined and user-defined objects, applications should not use procedure, function or variable names starting with *sql* or *SQL* or *SYS*.

c. Some implementations have keyword-oriented parsers. To ensure portability, the names of embedded host variables should be distinct from SQL keywords (including the additional reserved words referred to in a) above).

d. In some implementations, *user-name* can be longer than 18 characters. A program retrieving certain pseudo-literals into a host variable (see **Authorisation Pseudo-literals** on page 50) may have to allow an implementation-defined amount of additional space.

e. The user-defined names that are valid *user-name*s are implementation-defined.

7.4 Data Definition Statements in Transactions

Several products currently implement transaction semantics differently:

- In some cases, each data definition statement is a separate transaction that is immediately committed.
- In some cases, executing a data definition statement also commits all previously-executed data manipulation statements.
- In some implementations, data definition statements are fully controlled by transaction control statements, as X/Open specifies. But even in this case, transactions containing several statements, including one data definition statement, can produce interactions that X/Open does not define.

To portably produce the same effects, applications must obey these restrictions:

- The non-dynamic or dynamic execution of a data manipulation statement should not occur within the same transaction as the non-dynamic or dynamic execution of a data definition statement.
- Every data definition statement should immediately be followed by a COMMIT statement.
- Applications should not assume that a ROLLBACK statement reverses the effects of any data definition statement.
- If the program is assumed to be restartable after a crash, it must analyse its previous progress and must be prepared to clean up any intermediate state of data definition operations.

In effect, this means that there are currently only two portable classes of transactions:

- data manipulation transactions, which follow the defined rules for transactions and may either be committed or aborted
- data definition transactions, which consist of a single statement only, followed by a COMMIT statement.

If an implementation does not allow mixing of data definition and data manipulation statements in transactions, any statement executed that violates this rule sets SQLSTATE to '**25**000'.

7.5 Commitment of Transactions

Some distributed implementations currently may not correctly notify the application of the outcome of COMMIT in cases where communication errors occur. In these cases:

- The transaction may still be active.
- The transaction may have been committed successfully.
- The transaction may have been partially or totally rolled back.

The application should carefully check the status of the database before doing any further work.

7.6 Error Treatment

SQLCODE

Positive SQLCODE values other than +100 are handled in various ways; some implementations treat them as errors and trap them via SQLERROR and some treat them as warnings and trap them via SQLWARNING. Applications should therefore use the WHENEVER statement to determine the result class of an exception condition rather than test SQLCODE explicitly.

Indicator Variables

Section 3.8 on page 56 says that the value −1 in an indicator variable means that the accompanying host variable is null, and says that X/Open reserves other negative values for future definition.

Some implementations currently use negative values other than −1 in an indicator variable to indicate a null value.

Applications must use the value −1 in the indicator variable to convey to the implementation that the associated host variable is null. However, on output, applications must accept any negative number in the indicator variable and must not make inferences based on the particular negative number the implementation returns.

7.7 Textual Sequencing

There are two notable cases in which a specific sequence of execution is necessary:

- The PREPARE statement must precede the EXECUTE statement that executes the prepared statement.
- The PREPARE statement must precede any DESCRIBE statement that references the prepared statement.

On some implementations, some of the computation is performed by a preprocessor at compile time. The sequencing rules above would then apply to the textual sequence of the relevant statements in the source program. The International Standard does not specify textual precedence rules, but some implementations impose rules in the above cases on the order of execution, textual order, or both.

Portable applications should be written so that the statement pairs listed above both appear in the proper sequence and are executed in the proper sequence.

In three other cases, sequence rules always apply to the textual order in the program:

- The DECLARE CURSOR statement must precede any statement that references the defined cursor.
- The declaration of host variables and indicator variables must textually precede any references to the variables.
- The WHENEVER statement must textually precede any executable SQL statements that it is to affect.

It is implementation-defined whether multiple dynamic DECLARE CURSOR statements can reference the same *statement-name*.

7.8 SELECT

The definition of a *query-specification* (see Section 3.11.1 on page 71, Step 6) allows the use of two forms containing * to specify a sequence of *expression*s. However, it is implementation-defined whether the sequence is established at compile or run time. Therefore, the result of evaluating these *query-specification*s (for example, the number of columns retrieved by a SELECT statement) can differ among implementations. Portable applications should not use these * forms, except in CREATE VIEW, in which * is expanded when the view definition is originally executed.

Appendix A

Syntax Summary

This summary uses the notation defined in Section 3.1.1 on page 31. In addition, the symbol `::=` indicates that the syntactic element on its left is defined in the terms described on its right. Footnotes appear when they are more helpful than a mathematical description. The main text contains additional syntactic and semantic rules that are essential to the definitions.

In the following syntax summary, spaces are used for clarity. The actual rules for when separators must appear in an embedded SQL program are given in Section 3.1.2 on page 32.

A.1 Common Elements

```
approximate-numeric-literal ::= mantissaEexponent

    mantissa ::= exact-numeric-literal

    exponent ::= [+|-]unsigned-integer

approximate-numeric-type ::= FLOAT[(precision)] | DOUBLE PRECISION | REAL

base-table-name ::= [object-qualifier.]unqualified-base-table-name

between-predicate ::= expression [NOT] BETWEEN expression AND expression

cast-expression ::= CAST({expression | NULL}) AS data-type)

character ::= Any character in the implementor’s character set except the newline indication.

character-set-name ::= [object-qualifier.]unqualified-character-set-name

character-string-literal ::= '{character}...'

character-string-type ::=
    CHARACTER[(length)]
    | CHAR[(length)]
    | CHARACTER VARYING(length)
    | VARCHAR(length)

    length ::= unsigned-integer

collation-name ::= [object-qualifier.]unqualified-collation-name

column-definition ::=
    unqualified-column-name data-type
    [DEFAULT default-value]
    [column-constraint-definition
        [, column-constraint-definition]...]
    [COLLATE collation-name]

    column-constraint-definition ::=
        CHECK (search-condition)
        | NOT NULL
        | PRIMARY KEY
        | REFERENCES base-table-name-2 [(unqualified-column-name)]
        | UNIQUE

    default-value ::= literal | NULL

column-name ::=
    [{table-name | correlation-name}.]unqualified-column-name

comparison-operator ::= < | > | <= | >= | = | <>
```

```
comparison-predicate ::=
    expression comparison-operator { expression | (sub-query)}

conversion-name ::= [object-qualifier.]unqualified-conversion-name

correlation-name ::= user-defined-name

cursor-name ::= user-defined-name

data-type ::=
    character-string-type [CHARACTER SET character-set-name]
    | national-character-string-type
    | exact-numeric-type
    | approximate-numeric-type
    | date-time-type
    | interval-type

date-time-literal ::= •
    DATE 'date-value'
    | TIME 'time-value'
    | TIMESTAMP 'date-value space time-value'

    date-value ::=
        year-value minus month-value minus day-value

    time-value ::=
        hour-value colon minute-value colon second-value

    year-value ::= unsigned-integer
    month-value ::= unsigned-integer
    day-value ::= unsigned-integer
    hour-value ::= unsigned-integer
    minute-value ::= unsigned-integer
    second-value ::= unsigned-integer[.unsigned-integer]

date-time-type ::=
    DATE
    | TIME[(precision)]
    | TIMESTAMP[(precision)]

digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

dynamic-parameter ::= ? [interval-qualifier]

exact-numeric-literal ::=
    [+|-]{unsigned-integer[.unsigned-integer]
          | unsigned-integer.
          | .unsigned-integer}

exact-numeric-type ::=
    DECIMAL[(precision[, scale])]
    | DEC[(precision[, scale])]
    | INTEGER
    | INT
    | SMALLINT
    | NUMERIC[(precision[, scale])]

exists-predicate ::= EXISTS(sub-query)
```

• In this section, the syntactic elements *colon*, *minus* and *space* each denote a single character as determined by the character set in use.

```
expression ::=
    string-expression [COLLATE collation-name]
    | arithmetic-expression

    string-expression ::=
        primary | string-expression concat primary

        concat ::=  The concatenation operator, a double vertical bar

    arithmetic-expression ::=
        term | arithmetic-expression {+|-} term

    term ::= factor | term {*|/} factor

    factor ::= [+|-]primary

    primary ::=
        cast-expression
        | column-name
        | dynamic-parameter
        | host-variable-reference
        | literal
        | scalar-function-reference
        | set-function-reference
        | (expression)

host-variable-reference ::=
    embedded-variable-name[indicator-variable]

    embedded-variable-name ::= :host-identifier
    indicator-variable ::= :host-identifier

host-identifier ‡

index-name ::= [object-qualifier.]unqualified-index-name

in-predicate ::=
    expression [NOT] IN {(value {, value}...) | (sub-query)}

    value ::=
        host-variable-reference | literal | dynamic-parameter

keyword †

like-predicate ::=
    column-name [NOT] LIKE pattern-value [ESCAPE escape-character]

    pattern-value ::=
        character-string-literal
        | dynamic-parameter
        | host-variable-reference
        | CURRENT_USER
        | SESSION_USER
        | SYSTEM_USER
        | USER
```

‡ For COBOL, any valid data name; for C, any valid variable name. See also Section 4.1.2 on page 80.

† For the complete list of keywords, see Section 3.1.6 on page 37.

```
    escape-character ::=
        character-string-literal
        | dynamic-parameter
        | host-variable-reference
        | CURRENT_USER
        | SESSION_USER
        | SYSTEM_USER
        | USER

interval-literal ::= INTERVAL [+ | -]
    {year-value minus month-value
    | year-value
    | month-value
    | day-value space hour-value colon minute-value colon second-value
    | day-value space hour-value colon minute-value
    | day-value space hour-value
    | day-value
    | hour-value colon minute-value colon second-value
    | hour-value colon minute-value
    | hour-value
    | minute-value colon second-value
    | minute-value
    | second-value}
    interval-qualifier
```

The component syntactic elements above are defined as for *date-time-literal*.

```
interval-type ::= INTERVAL interval-qualifier

    interval-qualifier ::=
        {YEAR[(leading-precision)] TO MONTH
        | YEAR[(leading-precision)]
        | MONTH[(leading-precision)]
        | DAY[(leading-precision)] TO SECOND[(fractional-precision)]
        | DAY[(leading-precision)] TO MINUTE
        | DAY[(leading-precision)] TO HOUR
        | DAY[(leading-precision)]
        | HOUR[(leading-precision)] TO SECOND[(fractional-precision)]
        | HOUR[(leading-precision)] TO MINUTE
        | HOUR[(leading-precision)]
        | MINUTE[(leading-precision)] TO SECOND[(fractional-precision)]
        | MINUTE[(leading-precision)]
        | SECOND[(leading-precision [,fractional-precision)] ]

    fractional-precision ::= unsigned-integer
    leading-precision ::= unsigned-integer

literal ::=
    character-string-literal
    | date-time-literal
    | interval-literal
    | national-character-string-literal
    | numeric-literal
    | pseudo-literal

national-character-string-literal ::= N'{character}...'
```

```
national-character-string-type ::=
    NATIONAL CHAR[(length)]
    | NATIONAL CHARACTER[(length)]
    | NATIONAL CHARACTER VARYING(length)
    | NATIONAL CHAR VARYING(length)
    | NCHAR[(length)]
    | NCHAR VARYING(length)

    length ::= unsigned-integer

null-predicate ::= column-name IS [NOT] NULL

numeric-literal ::=
    exact-numeric-literal
    | approximate-numeric-literal

object-qualifier ::=
    catalog-name . schema-name
    | schema-name•

    catalog-name ::= user-defined-name
    schema-name ::= user-defined-name

overlaps-predicate ::=
    row-value-constructor-1 OVERLAPS row-value-constructor-2

precision ::= unsigned-integer

predicate ::=
    between-predicate
    | comparison-predicate
    | exists-predicate
    | in-predicate
    | like-predicate
    | null-predicate
    | overlaps-predicate
    | quantified-predicate

privilege-object ::=
    [TABLE] table-name
    | CHARACTER SET character-set-name
    | COLLATION collation-name
    | TRANSLATION translation-name

pseudo-literal ::=
    CURRENT_DATE
    | CURRENT_TIME[(precision)]
    | CURRENT_TIMESTAMP[(precision)]
    | CURRENT_USER
    | SESSION_USER
    | SYSTEM_USER
    | USER

quantified-predicate ::=
    expression comparison-operator {ALL | ANY | SOME} (sub-query)
```

• Other, implementation-defined object qualification schemes are also allowed.

```
    query-expression ::=
        query-expression UNION [ALL] query-expression
        | (query-expression)
        | query-specification

query-specification ::=
    SELECT [ALL | DISTINCT] select-list
    FROM table-reference [, table-reference]...
    [WHERE search-condition]
    [GROUP BY column-clause [, column-clause]...]
    [HAVING search-condition]

    column-clause ::= column-name [COLLATE collation-name]

restricted-name ::= namechar [namechar | digit | _ ]...
```

    namechar ::= Any ‘‘letter’’ in the character set — see **User-defined Names** on page 33

```
row-value-constructor ::=
    (column-spec[, column-spec]...)

    column-spec ::= expression
    | CURRENT_USER
    | DEFAULT
    | NULL
    | SESSION_USER
    | SYSTEM_USER
    | USER

scalar-function-reference ::=
    CHAR_LENGTH(string-expression)
    | CHARACTER_LENGTH(string-expression)
    | CONVERSION(string-expression USING conversion-name)
    | LOWER(string-expression)
    | OCTET_LENGTH(string-expression)
    | POSITION(search-string IN source-string)
    | SUBSTRING(source-string FROM start-position [FOR string-length])
    | TRANSLATION(string-expression USING translation-name)
    | TRIM([[LEADING | TRAILING | BOTH] [trim-character] FROM]
        source-string)
    | UPPER(string-expression)

    source-string ::= string-expression

    search-string ::= string-expression

    start-position ::= arithmetic-expression

    string-length ::= arithmetic-expression

    trim-character ::= string-expression

scale ::= unsigned-integer

search-condition ::=
    boolean-term [OR search-condition]

    boolean-term ::= boolean-factor [AND boolean-term]

    boolean-factor ::= [NOT] boolean-primary

    boolean-primary ::= predicate | (search-condition)

select-list ::= * | select-sublist [, select-sublist]...
```

```
    select-sublist ::=
        expression [[AS] unqualified-column-name]
        | {table-name | correlation-name}.*

separator ::= The blank character, the newline indication, or an SQL comment.

set-function-reference ::= COUNT(*) | distinct-function | all-function

    distinct-function ::=
        {AVG | COUNT | MAX | MIN | SUM} (DISTINCT column-name)

    all-function ::= {AVG | MAX | MIN | SUM} ([ALL] expression)

vendor-escape-clause ::=

    --(* dialect, extended-SQL-text *)--

    dialect ::= ISO-dialect | vendor-dialect

    ISO-dialect ::= YEAR(ISO-year), CONFORMANCE(ISO-conformance)

    vendor-dialect ::= VENDOR(vendor-name), PRODUCT(product-name)

statement-name ::= user-defined-name

sub-query ::=
    SELECT [ALL | DISTINCT] select-list
    FROM table-reference [, table-reference]...
    [WHERE search-condition]
    [GROUP BY column-name [, column-name]...]
    [HAVING search-condition]

table-name ::= base-table-name | viewed-table-name

table-reference ::=
    table-name [[AS] correlation-name]
    | joined-table

    joined-table ::=
    qualified-join | (joined-table)

    qualified-join ::= table-reference-1
        [NATURAL]
        [INNER | LEFT [OUTER] | RIGHT [OUTER]]
        JOIN table-reference-2
        {ON search-condition | USING (column-name[, column-name...])}

token ::= delimiter-token | non-delimiter-token

    delimiter-token ::=
        character-string-literal
        | , | ( | ) | < | > | . | :
        | = | * | + | - | / | <> | >= | <= | ? | concat

    non-delimiter-token ::=
        keyword | numeric-literal | user-defined-name | host-identifier

translation-name ::= [object-qualifier.]unqualified-translation-name

unqualified-base-table-name ::= user-defined-name

unqualified-character-set-name ::= restricted-name

unqualified-collation-name ::= user-defined-name

unqualified-column-name ::= user-defined-name

unqualified-conversion-name ::= user-defined-name

unqualified-index-name ::= user-defined-name

unqualified-translation-name ::= user-defined-name
```

```
unqualified-viewed-table-name ::= user-defined-name

unsigned-integer ::= {digit}...

user-defined-name ::= [ _character-set-name] restricted-name

user-name ::= user-defined-name

viewed-table-name ::= [object-qualifier.]unqualified-viewed-table-name
```

A.2 Embedded Aspects

```
association-management-statement  ::=
    connect-statement
    | disconnect-statement
    | set-connection-statement

data-definition-statement  ::=
    alter-table-statement
    | create-character-set-statement
    | create-collation-statement
    | create-index-statement
    | create-schema-statement
    | create-table-statement
    | create-translation-statement
    | create-view-statement
    | drop-character-set-statement
    | drop-collation-statement
    | drop-index-statement
    | drop-schema-statement
    | drop-table-statement
    | drop-translation-statement
    | drop-view-statement
    | grant-statement
    | revoke-statement

data-manipulation-statement  ::=
    close-statement
    | delete-statement-positioned
    | delete-statement-searched
    | fetch-statement
    | insert-statement
    | open-statement
    | select-statement
    | update-statement-positioned
    | update-statement-searched

declare-authorisation  ::=
    DECLARE AUTHORIZATION character-string-literal  [FOR STATIC ONLY]

declare-cursor  ::= DECLARE cursor-name CURSOR FOR cursor-specification

    cursor-specification  ::=
        query-expression
        [ORDER BY sort-specification[, sort-specification]. ..]
        [FOR {READ ONLY
            | UPDATE [OF unqualified-column-name[, unqualified-column-name]. ..]}]

    sort-specification  ::=
        {unqualified-column-name  | unsigned-integer}  [ASC | DESC]
```

```
dynamic-declare-cursor ::=
    DECLARE cursor-name CURSOR FOR statement-name

dynamic-sql-statement ::=
    allocate-descriptor-statement
    | deallocate-descriptor-statement
    | describe-statement
    | dynamic-close-statement
    | dynamic-delete-statement-positioned
    | dynamic-fetch-statement
    | dynamic-open-statement
    | dynamic-update-statement-positioned
    | execute-statement
    | execute-immediate-statement
    | get-descriptor-statement
    | prepare-statement
    | set-descriptor-statement

embedded-exception-declaration ::=
    WHENEVER {SQLERROR | SQLWARNING | NOT FOUND}
    {CONTINUE | {GOTO | GO TO} host-label}

embedded-sql-construct ::=
    embedded-sql-declarative | embedded-sql-statement

embedded-sql-declarative ::=
    embedded-sql-declare-section

embedded-sql-declare-section ::=
    sql-prefix BEGIN DECLARE SECTION sql-terminator
    [host-variable-definition]...
    sql-prefix END DECLARE SECTION sql-terminator

embedded-sql-statement ::=
    sql-prefix
    {declare-authorisation
    | declare-cursor
    | dynamic-declare-cursor
    | embedded-exception-declaration
    | executable-sql-statement}
    sql-terminator

executable-sql-statement ::=
    data-definition-statement
    | data-manipulation-statement
    | transaction-control-statement
    | dynamic-sql-statement
    | association-management-statement
    | get-diagnostics-statement
    | session-statement

host-label †

host-variable-definition ‡

sql-prefix ::= EXEC SQL
```

† For COBOL, a section name or an unqualified paragraph name. For C, a label defined in the same program block or in a block of a higher scope. (See also Section 4.6 on page 94.)

‡ Any valid variable definition following the rules in Section 4.2 on page 84.

```
sql-terminator •

transaction-control-statement ::=
    commit-statement
    | rollback-statement
    | set-transaction-statement
```

• For COBOL, `END-EXEC`. For C, a semicolon (‘‘;’’).

A.3 Executable SQL Statements

```
allocate-descriptor-statement ::=
    ALLOCATE DESCRIPTOR descriptor-name [WITH MAX occurrences]

    occurrences ::= unsigned-integer | embedded-variable-name

alter-table-statement ::=
    ALTER TABLE base-table-name
    ADD [COLUMN] column-definition
    | DROP [COLUMN] unqualified-column-name {CASCADE | RESTRICT}

close-statement ::= CLOSE cursor-name

commit-statement ::= COMMIT [WORK]

connect-statement ::=
    CONNECT TO server [AS connection]
    [USER user [USING authentication]]

    server ::=
        character-string-literal | embedded-variable-name | DEFAULT

    connection ::= character-string-literal | embedded-variable-name

    user ::= character-string-literal | embedded-variable-name

    authentication ::= embedded-variable-name

create-character-set-statement ::=
    CREATE CHARACTER SET new-char-set [AS]
    GET [character-set-name
    [COLLATE collation-name
    | COLLATION FROM collation-source]

create-collation-statement ::=
    CREATE COLLATION new-collation FOR character-set-name
    FROM collation-source
    [PAD SPACE | NO PAD]

    collation-source ::= collation-name
        | DEFAULT
        | DESC(collation-name)
        | EXTERNAL(character-string-literal)
        | TRANSLATION translation-name [THEN COLLATION collation-name]

create-index-statement ::=
    CREATE [UNIQUE] INDEX index-name ON base-table-name
    (unqualified-column-name [ASC|DESC][, unqualified-column-name [ASC|DESC]]...)

create-schema-statement ::=
    CREATE SCHEMA [object-qualifier] [AUTHORIZATION auth-id]
    [DEFAULT CHARACTER SET character-set-name]
    [schema-element [schema-element]...]

    schema-element ::=
        create-index-statement
        | create-table-statement
        | create-view-statement
        | grant-statement
```

```
create-table-statement ::=
    CREATE TABLE base-table-name-1 (column-element [, column-element]. ..)

    column-element ::= column-definition | table-constraint-definition

    table-constraint-definition ::=
        UNIQUE (unqualified-column-name [, unqualified-column-name]. ..)
        | PRIMARY KEY (unqualified-column-name[, unqualified-column-name]. ..)
        | CHECK (search-condition)
        | FOREIGN KEY referencing-columns
            REFERENCES base-table-name-2 referenced-columns
            [ON DELETE {NO ACTION | CASCADE | SET NULL | SET DEFAULT}]

create-translation-statement ::=
    CREATE TRANSLATION new-translation
    FOR source-char-set TO target-char-set FROM
        { IDENTITY
        | translation-name
        | EXTERNAL(character-string-literal)

create-view-statement ::=
    CREATE VIEW viewed-table-name
    [(unqualified-column-name [, unqualified-column-name]. ..)]
    AS query-expression

deallocate-descriptor-statement ::=
    DEALLOCATE DESCRIPTOR descriptor-name

delete-statement-positioned ::=
    DELETE FROM table-name WHERE CURRENT OF cursor-name

delete-statement-searched ::=
    DELETE FROM table-name [WHERE search-condition]

describe-statement ::=
    DESCRIBE [INPUT | OUTPUT] statement-name
    USING SQL DESCRIPTOR descriptor-name

descriptor-name ::= character-string-literal | embedded-variable-name

disconnect-statement ::= DISCONNECT [connection | ALL | CURRENT ]

drop-character-set-statement ::= DROP CHARACTER SET character-set-name

drop-collation-statement ::= DROP COLLATION collation-name

drop-index-statement ::= DROP INDEX index-name

drop-schema-statement ::= object-qualifier {CASCADE | RESTRICT}

drop-table-statement ::= DROP TABLE base-table-name {CASCADE | RESTRICT}

drop-translation ::= DROP TRANSLATION translation-name

drop-view-statement ::= DROP VIEW viewed-table-name {CASCADE | RESTRICT}

dynamic-close-statement ::= close-statement

dynamic-delete-statement-positioned ::= delete-statement-positioned

dynamic-fetch-statement ::= FETCH [FROM] cursor-name using-clause

dynamic-open-statement ::= OPEN cursor-name [using-clause ]

dynamic-update-statement-positioned ::= update-statement-positioned

execute-statement ::= EXECUTE statement-name [using-clause]

execute-immediate-statement ::= EXECUTE IMMEDIATE embedded-variable-name
```

```
fetch-statement ::=
    FETCH [FROM] cursor-name INTO host-variable-reference
    [, host-variable-reference]. ..

field-name ::=
    TYPE | LENGTH | PRECISION | SCALE
    | DATETIME_INTERVAL_CODE | DATETIME_INTERVAL_PRECISION
    | INDICATOR | DATA

get-descriptor-statement ::=
    GET DESCRIPTOR descriptor-name get-descriptor-information

    get-descriptor-information ::=
        embedded-variable-name-1 = COUNT
        | VALUE item-number get-item-info  [, get-item-info]. ..

    get-item-info ::=
        embedded-variable-name-2 =
        field-name | NAME | NULLABLE | UNNAMED | RETURNED_LENGTH

get-diagnostics-statement ::=
    GET DIAGNOSTICS {statement-information | exception-information}

    statement-information ::=
        statement-information-item [, statement-information-item]. ..

        statement-information-item ::=
            embedded-variable-name-1= field-name-1

    exception-information ::=
        EXCEPTION exception-number
        exception-information-item [, exception-information-item]. ..

        exception-information-item ::=
            embedded-variable-name-2 = field-name-2

        exception-number ::=
            unsigned-integer | embedded-variable-name-3

        field-name-1 ::= NUMBER | MORE | DYNAMIC_FUNCTION
            | DYNAMIC_FUNCTION_CODE | ROW_COUNT

        field-name-2†

grant-statement ::=
    GRANT {ALL PRIVILEGES | privilege [, privilege]...}
    ON privilege-object
    TO {PUBLIC | user-name [, user-name]...}
    [WITH GRANT OPTION]
```

† Specifies a field name associated with a specific diagnostic. For the complete list of valid values, see Section 5.10 on page 150.

```
insert-statement ::=
    INSERT INTO table-name
    {insert-source | DEFAULT VALUES }

    insert-source ::=
        [(unqualified-column-name [,unqualified-column-name]. ..)]
        {query-specification
        | VALUES row-value-constructor}

item-number ::= unsigned-integer | embedded-variable-name

open-statement ::= OPEN cursor-name

prepare-statement ::=
    PREPARE statement-name FROM embedded-variable-name

privilege ::=
    DELETE
    | INSERT
    | REFERENCES [(unqualified-column-name [, unqualified-column-name]. ..)]
    | SELECT
    | UPDATE [(unqualified-column-name [, unqualified-column-name]. ..)]
    | USAGE

revoke-statement ::=
    REVOKE {ALL PRIVILEGES | privilege [, privilege]...}
    ON privilege-object
    FROM {PUBLIC | user-name [, user-name]...}
    {CASCADE | RESTRICT}

rollback-statement ::= ROLLBACK [WORK]

select-statement ::=
    SELECT [ALL | DISTINCT] select-list
    INTO host-variable-reference [, host-variable-reference]. ..
    FROM table-reference [, table-reference]...
    [WHERE search-condition]

session-statement ::=
    SET [CATALOG | SCHEMA | SESSION AUTHORIZATION]
        {character-string-literal | embedded-variable-name}
    | SET NAMES character-set-name

set-connection-statement ::= SET CONNECTION server

set-descriptor-statement ::=
    SET DESCRIPTOR descriptor-name set-descriptor-information

    set-descriptor-information ::=
        COUNT = embedded-variable-name-1 | unsigned-integer
        | VALUE item-number set-item-info [, set-item-info]...

    set-item-info ::=
        field-name = embedded-variable-name-2 | literal

set-transaction-statement ::=
    SET TRANSACTION transaction-mode[, transaction-mode]...

    transaction-mode ::=
        DIAGNOSTICS SIZE number
        | ISOLATION LEVEL READ UNCOMMITTED
        | ISOLATION LEVEL READ COMMITTED
        | ISOLATION LEVEL REPEATABLE READ
        | ISOLATION LEVEL SERIALIZABLE
        | READ ONLY
        | READ WRITE
```

```
update-statement-positioned ::=
    UPDATE table-name
    SET unqualified-column-name = {expression | NULL | DEFAULT VALUE}
    [, unqualified-column-name = {expression | NULL | DEFAULT VALUE}]...
    WHERE CURRENT OF cursor-name

update-statement-searched ::=
    UPDATE table-name
    SET unqualified-column-name = {expression | NULL | DEFAULT VALUE}
    [, unqualified-column-name = {expression | NULL | DEFAULT VALUE}]...
    [WHERE search-condition]

using-clause ::= using-arguments | using-descriptor

    using-arguments ::=
        {USING | INTO} host-variable-reference
        [, host-variable-reference]. ..

    using-descriptor ::=
        {USING | INTO} SQL DESCRIPTOR descriptor-name
```

Appendix B

SQLSTATE Values

SQLSTATE is a 5-character string that an SQL statement returns to indicate status. The following tables define all the SQLSTATE values that apply to X/Open SQL. Where these tables give more than one SQLSTATE value that may pertain to a statement, the text of this specification indicates which value is the primary error.

The **Meaning** column is not normative or visible to the application. Its contents are generally based on the International Standard but sometimes have been adapted to use X/Open terminology. A change in wording should not be construed as defining a different diagnostic from the International Standard.

SQLSTATE	Meaning	Section
’**00**000	**Success**	**4.5.1**
’**01**000’	**Success with warning**	**4.5.1**
’**01**002’	— Disconnect error.	**5.7.7**
’**01**003’	— Null value eliminated in set function.	**3.9.4**
’**01**004’	— String data, right truncation.	**3.5, 3.8**
’**01**005’	— Insufficient item descriptor areas.	**5.5.5**
’**01**006’	— Privilege not revoked.	**5.3.18**
’**01**007’	— Privilege not granted.	**5.3.17**
’**01**00B’	— Default value too long for system view.	**5.3.2,7**
’**02**000’	**No data**	**4.5.1, 5.4, 5.5**
’**07**000’	**Dynamic SQL error**	
’**07**001’	— *using-clause* does not match dynamic parameters.	**5.5.6,10**
’**07**002’	— *using-clause* does not match target specifications.	**5.5.8**
’**07**003’	— Cursor specification cannot be executed.	**5.5.6**
’**07**004’	— *using-clause* is required for dynamic parameters.	**5.5.6,10**
’**07**005’	— Prepared statement is not a cursor specification.	**5.5.11**
’**07**006’	— Restricted data type attribute violation.	**5.5.2**
’**07**008’	— Invalid descriptor count.	**5.5.6,8,10**
’**07**009’	— Invalid descriptor index.	**5.5.3,9,12**
’**08**000’	**Connection exception**	
’**08**001’	— Client unable to establish connection	**5.7.6**
’**08**002’	— Connection name in use.	**5.7.3,6**
’**08**003’	— Connection does not exist.	**5.2.6, 5.7.3,5,7,8**
’**08**004’	— Server rejected the connection.	**5.7.6**
’**08**006’	— Connection failure.	**5.7.5,8**
’**08**007’	— Transaction resolution unknown.	**5.6.2, 5.7.5**
’**0A**000’	**Feature not supported**	
’**0A**001’	— Multiple-server transaction.	**5.7.5**
’**21**000’	**Cardinality violation**	**3.10.1, 3.11.4, 5.4.7**
’**21**S01’	— Insert value list does not match column list.*	**5.2.1, 5.4.5**
’**21**S02’	— Degree of derived table does not match column list.*	**5.2.1, 5.3.9**
’**22**000’	**Data exception**	
’**22**001’	— String data, right truncation.	**3.5, 3.9.3,6**
’**22**002’	— Null value, no indicator parameter.	**3.7, 5.5.9**
’**22**003’	— Numeric value out of range.	**3.5, 3.9.1,4,6**
’**22**005’	— Error in assignment.	**5.5.9,12**
’**22**006’	— Invalid interval format.	**3.9.6**
’**22**007’	— Invalid date/time format.	**3.4, 3.5, 3.9.6**
’**22**008’	— Date/time field overflow.	**3.5, 3.9.2**
’**22**011’	— Substring error	**3.9.3**
’**22**012’	— Division by zero.	**3.9.1,2**
’**22**015’	— Interval field overflow.	**3.5, 3.9.2,6**
’**22**018’	— Invalid character value for CAST	**3.9.6**
’**22**019’	— Invalid escape character.	**3.10.5**
’**22**021’	— Translation result not in target repertoire	**3.9.3**
’**22**024’	— Unterminated string.	**3.5**
’**22**025’	— Invalid escape sequence.	**3.10.5**
’**22**027’	— Trim error.	**3.9.3**

SQLSTATE	Meaning	Section
'**23**000'	**Integrity constraint violation**	**5.2.4, 5.3.2,7**
'**24**000'	**Invalid cursor state**	**5**
'**25**000'	**Invalid transaction state**	**5.2.5, 5.6.4, 5.7.5,7, 5.9, 7.4**
'**26**000'	**Invalid SQL statement identifier**	**5.5.5,6,10**
'**28**000'	**Invalid authorisation specification**	**5.7.6, 5.9**
'**2C**000'	**Invalid character set name**	**5.8.2**
'**2D**000'	**Invalid transaction termination**	**5.6.1**
'**2E**000'	**Invalid connection name**	**5.7.6**
'**33**000'	**Invalid SQL descriptor name**	**5.5.2,3,4,9,12**
'**35**000'	**Invalid exception number**	**5.6.4, 5.9.1**
'**3D**000'	**Invalid catalog name.**	**5.8.1**
'**3F**000'	**Invalid schema name.**	**5.8.3**
'**40**000'	**Transaction rollback**	
'**40**003'	— Statement completion unknown	**5.7.5**
'**42**000'	**Syntax error or access violation** (see below).	**3.1.1, 3.9.3, 5, 5.2.1, 7.2.1**
'**42**S01'	— Base table or viewed table already exists.*	**5.3.7,9**
'**42**S02'	— Base table not found.*	**5.2.1, 5.3.5,14,16**
'**42**S11'	— Index already exists.*	**5.3.5**
'**42**S12'	— Index not found.*	**5.3.12**
'**42**S21'	— Column already exists.*	**5.3.2**
'**42**S22'	— Column not found.*	**5.3.5,7,17**
'**42**S31'	— Schema already exists.*	**5.3.6**
'**42**S32'	— Schema not found.*	**5.2.1, 5.3.13**
'**42**S42'	— Catalog not found.*	**5.2.1**
'**42**S51'	— Character set already exists.*	**5.3.3**
'**42**S52'	— Character set not found.*	**5.2.1, 5.3.10**
'**42**S61'	— Collation already exists.*	**5.3.4**
'**42**S62'	— Collation not found.*	**5.2.1, 5.3.11**
'**42**S72'	— Conversion not found.*	**5.2.1**
'**42**S81'	— Translation already exists.*	**5.3.8**
'**42**S82'	— Translation not found.*	**5.2.1, 5.3.15**
'**44**000'	**WITH CHECK OPTION violation**	**5.3.9, 5.4.5,8,9**

* This code does not appear in SQLSTATE but may appear in the diagnostics area. Implementations may raise '**42**000' instead of this code if, for example, the application is not entitled to obtain this schema information. See Section 5.2.1 on page 99.

The following codes originate at a remote SQL server and are based on RDA service errors. See the referenced X/Open **RDA** Specification.

SQLSTATE	Meaning
'**HZ**000'	**RDA error**
'**HZ**010'	— AccessControlViolation
'**HZ**020'	— BadRepetitionCount
'**HZ**030'	— CommandHandleUnknown
'**HZ**040'	— ControlAuthenticationFailure
'**HZ**050'	— DataResourceHandleNotSpecified
'**HZ**060'	— DataResourceHandleUnknown
'**HZ**070'	— DataResourceNameNotSpecified
'**HZ**080'	— DataResourceNotAvailable
'**HZ**081'	(transient error)
'**HZ**090'	— DataResourceAlreadyOpen
'**HZ**100'	— DataResourceUnknown
'**HZ**110'	— DialogueIDUnknown
'**HZ**120'	— DuplicateCommandHandle
'**HZ**130'	— DuplicateDataResourceHandle
'**HZ**140'	— DuplicateDialogueID
'**HZ**150'	— DuplicateOperationID
'**HZ**160'	— InvalidSequence
'**HZ**161'	(dialogue already active)
'**HZ**162'	(dialogue initialising)
'**HZ**163'	(dialogue not active)
'**HZ**164'	(dialogue terminating)
'**HZ**165'	(transaction not open)
'**HZ**166'	(transaction open)
'**HZ**167'	(transaction terminating)
'**HZ**170'	— NoDataResourceAvailable
'**HZ**180'	— OperationAborted
'**HZ**181'	(transient error)
'**HZ**190'	— OperationCancelled
'**HZ**200'	— ServiceNotNegotiated
'**HZ**210'	— TransactionRolledBack
'**HZ**220'	— UserAuthenticationFailure
'**HZ**230'	— ControlServicesNotAllowed
	RDA SQL Specialisation Condition
'**HZ**300'	— HostIdentifierError
'**HZ**310'	— InvalidSQLConformanceLevel
'**HZ**320'	— RDATransactionNotOpen
'**HZ**325'	— RDATransactionOpen
'**HZ**330'	— SQLAccessControlViolation
'**HZ**340'	— SQLDatabaseResourceAlreadyOpenError
'**HZ**350'	— SQLDBLArgumentCountMismatch
'**HZ**360'	— SQLDBLArgumentTypeMismatch
'**HZ**365'	— SQLNoCharSet
'**HZ**370'	— SQLDBLTransactionStatementNotAllowed

SQLSTATE	**Meaning**
’**HZ**380’	— SQLUsageModeViolation
	Association Control Service Element Condition
’**HZ**410’	— Abort failure, service provider
’**HZ**411’	— Abort failure, service user
’**HZ**420’	— Association failure, permanent
’**HZ**421’	— Association failure, transient
’**HZ**430’	— Release failure
	Distributed Transaction Processing Condition
’**HZ**450’	— Begin dialogue rejected, provider
’**HZ**451’	— Begin dialogue rejected, user
’**HZ**460’	— Heuristic hazard
’**HZ**461’	— Heuristic mix
’**HZ**470’	— PAbort rollback false
’**HZ**471’	— PAbort rollback true
’**HZ**480’	— Rollback
’**HZ**490’	— UAbort rollback false
’**HZ**491’	— UAbort rollback true
’**HZ**4A0’	— UError

Appendix C

ISO Database Language SQL

This X/Open SQL specification is based on the referenced International Standard. X/Open intends that implementations be able to comply with both this document and with the International Standard. Any discrepancies between the two documents are inadvertent and are to be resolved in favour of the International Standard.

Compliance with Respect to the International Standard

An X/Open-compliant implementation provides all the Entry level facilities of the International Standard. It also provides features, from the Intermediate and Full levels of the International Standard, listed in Section C.1, and the other features listed in Section C.2 on page 212. It must support bindings to an X/Open-compliant host language (see the start of Chapter 4). It must also support all deprecated features (see **Deprecated features** on page 2).

If an X/Open-compliant SQL embedded source program uses deprecated features, it might not continue to be portable to implementations that comply with future X/Open specifications. If the source program uses features specified as optional, implementation-defined or undefined, it will typically not be portable even to some existing X/Open-compliant implementations.

C.1 Included Features from Intermediate and Full SQL

The following tables list the features that X/Open SQL includes from the Intermediate and Full conformance levels of the International Standard.

The features are presented in the order of their appearance in FIPS 127-2. (FIPS item numbers of the form T*n* are from the definition of the Transitional level; those of the form I*n* are from the Intermediate level; and those of the form F*n* are from the Full level.)

In the right-hand column, the SPIRIT feature number is provided. SPIRIT SQL is defined in appendices of this specification. For summaries of features with SPIRIT numbers from 1 up to and including 26, see the **Feature List** on page 215. For summaries of features with SPIRIT numbers from 27 up to and including 43, see the list in Section E.1 on page 233. For summaries of features with SPIRIT numbers from 44 up to and including 46, see the list in Section E.4 on page 239.

	FIPS Transitional Feature	**SPIRIT**
Number	**Description**	**Number**
T1	Dynamic SQL	1
T2	Basic information schema	2
T3	Basic schema manipulation	3, 37
T4	Joined table	14, 40
T5	DATETIME data type	27
T6	VARCHAR data type	4
T7	TRIM function	28
T8	UNION in views	29
T9	Implicit numeric casting	5
T10	Implicit character casting	6
T11	Transaction isolation	17, 30
T12	Get diagnostics	7
T13	Grouped operations	16
T14	Qualified * in select list	8
T15	Lowercase identifiers	9
T16	PRIMARY KEY enhancement	31
T17	Multiple schemas per user	15
T18	Multiple module support	10
T19	Referential delete actions	21
T20	CAST functions	32
T21	INSERT expressions	33
T22	Explicit defaults	34
T23	Privilege tables	35
T24	Keyword relaxations	36

Number	FIPS Intermediate Feature Description	SPIRIT Number
I31	Schema definition statement	41
I42	National character	11
I46	Named character sets	44
I54	Full Dynamic SQL (part of) — support of DESCRIBE INPUT from clause 17 of the International Standard	—
I58	Full character functions	42

OP (I54)

Number	FIPS Full Feature Description	SPIRIT Number
F65	Catalog name qualifiers	38
F69	Collation and translation	45
F76	Session management	12, 39
F77	Connection management	13

OP (F65)

Number	Feature Not In FIPS Description	SPIRIT Number
—	Additional internationalization	46

C.2 Extensions to the International Standard

An extension is a stand-alone facility that is in X/Open SQL but not in the International Standard at any conformance level. The X/Open specification of these features does not inhibit an implementation from conforming to the International Standard.

X/Open includes the following extensions:

- the requirement that implementations permit multiple servers to be included in a single transaction
- the CREATE [UNIQUE] INDEX statement
- the DROP INDEX statement
- WHENEVER SQLWARNING
- embedded COBOL host variables, with the following usages:
  - COMP, corresponding to DECIMAL and NUMERIC (and to INTEGER and SMALLINT, except in the case of SQLCODE)
  - COMP-3, corresponding to DECIMAL, NUMERIC, INTEGER and SMALLINT
  - no usage (just DISPLAY), corresponding to INTEGER and SMALLINT
- the INDEXES and SERVER_INFO system view
- the REMARKS column on the COLUMN_PRIVILEGES, COLUMNS, INDEXES, SCHEMATA, TABLE_PRIVILEGES, TABLES, USAGE_PRIVILEGES and VIEWS system views (on implementations that do not provide remarks, this column is null)
- the vendor escape clause
- the DECLARE AUTHORIZATION statement.

Appendix D

SPIRIT SQL, Issue 2

This Appendix is technically identical to Issue 2 of the SPIRIT SQL specification. (SPIRIT Issue 3 mandates additional features from the implementation. See Appendix E.)

A detailed comparison between SPIRIT SQL and X/Open embedded SQL appears in Section D.7 on page 231.

D.1 Introduction to SPIRIT

The Service Providers' Integrated Requirements for Information Technology (SPIRIT) is constituted as a team within the Network Management Forum.

SPIRIT is a collaborative effort of international telecommunications Service Providers (SP), Information Technology (IT) suppliers, and Independent Software Vendors (ISVs). Its intent is to construct a common set of software specifications for a general purpose computing platform, driven from the requirements of the telecommunications industry.

This platform defines software that supports a wide variety of application types. The platform facilitates application portability, interoperability, and modularity. Agreement among Service Providers, IT suppliers, and ISVs on such a platform is required to meet Service Providers' needs for integrated systems and technology independence in a multivendor software environment.

The objective of these SPIRIT documents is to provide a core set of specifications for use in each company's purchasing of software components for general purpose computing platforms. The SPIRIT specifications are based predominantly on widely-accepted industry standards. The SPIRIT specifications are expected to be used by participating companies as a basis for their own software procurement starting within 1 to 2 years of publication.

The intent of SPIRIT is to adopt and adapt specifications from other sources, including both standards bodies and industry consortia. Adaptation is performed only as necessary to:

- ensure consistency among specifications from diverse sources
- harmonise the use of certain specifications within specific usage scenarios
- remove ambiguities and/or inconsistencies within chosen specifications
- limit features and options for reasons of availability within specific time frames
- limit features and options for technical and/or business reasons.

SPIRIT profiles are created to help in meeting the SPIRIT goals of portability, interoperability, and modularity. SPIRIT profiles can be characterised as having no invention, only selection. The features of SPIRIT profiles are selected based on user requirements.

Objectives

The primary objective of the SPIRIT SQL profile is to provide a clear definition of an SQL language that is available for procurement in the SPIRIT Issue 2 timeframe. This objective includes a requirement that service providers be able to readily write meaningful applications that are portable, without any recoding, among conforming implementations of this profile. Other objectives include alignment with X/Open SQL specifications and *de jure* standards for SQL, as well as character internationalisation support required by applications written by service providers in North America, Europe, and Japan. Vendors are always free to provide facilities beyond those required by this document, including minimum limits on the size of various items, but applications should not use any facilities not required by this document in order to maximize portability.

This document comprises a profile of the International Standard using the format of the referenced NIST FIPS for SQL. The structure of this document is primarily identification of features in the International Standard, but it also identifies features in X/Open SQL. Finally, some internationalisation features — a subset of internationalisation features of the International Standard, with minor extensions to the host language bindings — are derived from the MIA SQL specification. (These features are specified in this appendix because the MIA specification is not widely available.) These internationalisation features now depend on the SPIRIT Character Set Profile defined in Volume 1 (Core Specifications), Part 1 (Overview) of SPIRIT Issue 2.0.

Applicability

This profile will be used for procurements by SPIRIT service providers to specify characteristics of SQL database management systems beginning in Summer, 1995.

Relation to the International Standard

SPIRIT SQL adopts provisions of the International Standard as described below:

SPIRIT SQL requires conformance to Entry SQL and to additional aspects of the language as specified below. Conformance is further constrained by the limits specified in Section 7.1 on page 175 and the definition of various items in Section D.5 on page 228 that the International Standard leaves implementation-defined. SPIRIT Issue 2 approximates the X/Open SQL specification.

All present and future Technical Corrigenda to the International Standard that affect specific items in SPIRIT SQL are implicitly included as part of these profiles, as they are implicitly part of the International Standard. Conformance to these corrections is required in a reasonable timeframe following their publication.

D.2 Conformance Requirements

Conformance to SPIRIT Issue 2 SQL requires the implementation of all the capabilities of Entry level of the International Standard. Conformance also requires the implementation of additional capabilities beyond Entry level, which are listed below.

Citations of the International Standard

All terms delimited by <angle brackets> refer to syntactic productions specified in the International Standard. All cross-references to clauses and subclauses signify the International Standard.

FIPS 127-2 Citations

Most of the features listed below are accompanied by a cross-reference to a numbered feature of FIPS 127-2. The cross-reference is in one of the following forms:

[FIPS T*n*] Transitional SQL level of FIPS 127-2
[FIPS I*n*] Intermediate SQL level of FIPS 127-2
[FIPS F*n*] Full SQL level of FIPS 127-2

The value of *n* is the identifying number of the respective feature in FIPS 127-2.

FIPS 127-2 contains a list of features from Entry level of the International Standard. (These features are numbered in the form E*n*.) Since SPIRIT conformance requires implementation of all features of Entry level, none of the E*n* features are specified separately in the following list.

X/Open Citations

Other entries are accompanied by a notation of the form "[No FIPS component; X/Open SQL]". This notation means that the entry is a feature of X/Open SQL specified in this document.

All cross-references to chapters and sections signify this document.

MIA Citations

Other entries are accompanied by a notation of the form "[No FIPS component; MIA SQL]", meaning that MIA SQL is the source of the feature.

Feature List

1. **Dynamic SQL. [FIPS T1]**
   a. All provisions of Clause 17, "Dynamic SQL", with restrictions identified in the Leveling Rules for Intermediate SQL.
   b. Removal of all Entry SQL Leveling Rules in Clause 17.
   c. Removal of the Entry SQL restriction requiring that a <module contents> not be a <dynamic declare cursor>, as specified in Leveling Rule 2b of Subclause 12.1, "<module>".
   d. Removal of Leveling Rule 2a of Subclause 6.2 that prohibits reference to a dynamic parameter in a <general value specification>.
   e. A <preparable statement> shall include all <preparable SQL data statement>s that are otherwise supported for Entry SQL, as well as any other <preparable statement> for which non-preparable support is claimed by that implementation.

2. **Basic information schema. [FIPS T2]**

   Requires existence of an accessible INFORMATION_SCHEMA consisting of the following views defined in Subclause 21.2, "Information Schema": TABLES, VIEWS, and COLUMNS, all with any restrictions identified in the Leveling Rules for Intermediate SQL.

3. **Basic schema manipulation. [FIPS T3, part 1]**

   a. Support for the following schema definition and schema manipulation statements as <SQL statement>s in an explicit or implicit <procedure>:

      i. Subclauses 11.3 through 11.9, "<table definition>"

      ii. Subclause 11.10, "<alter table statement>", containing Subclause 11.11, "<add column definition>"

      iii. Subclause 11.18, "<drop table statement>"

      iv. Subclause 11.19, "<view definition>"

      v. Subclause 11.20, "<drop view statement>"

      vi. Subclause 11.36, "<grant statement>"

      vii. Subclause 11.37, "<revoke statement>"; all with any other restrictions identified in the Leveling Rules for Entry SQL, and all with any enhancements derived from other features claimed to be supported by the implementation.

   b. Removal of Leveling Rules 2a in Subclauses 11.11, 11.15, 11.18, 11.20, and 11.37.

4. **VARCHAR data type. [FIPS T6]**

   a. Support for CHARACTER VARYING, and its syntactic shorthands VARCHAR and CHAR VARYING, as defined in Subclause 6.1, "<data type>".

   b. Support for the following character operations:

      i. <length expression> defined in Subclause 6.6, "<numeric value function>"

      ii. <character substring function> defined in Subclause 6.7, "<string value function>"

      iii. <concatenation> and <character value expression> defined in Subclause 6.13, "<string value expression>".

   c. Support for comparison of fixed and variable length character strings as defined in Subclause 8.2, "<comparison predicate>".

   d. Support for CHARACTER VARYING in any <embedded SQL host program> supported by the implementation; all with any restrictions identified in the Leveling Rules for Intermediate SQL.

   e. Removal of the Entry SQL requirement that at least one <character representation> be present in a <character string literal>, as specified in Leveling Rule 2c of Subclause 5.3, "<literal>".

   f. Removal of Leveling Rule 2a of Subclause 6.1, "<data type>".

   g. Removal of the Entry SQL requirement that a <numeric value function> not be a <length expression>, as specified in Leveling Rule 2a of Subclause 6.6, "<numeric value function>".

   h. Removal of the Entry SQL restriction against using the SUBSTRING function, as specified in Leveling Rule 2a of Subclause 6.7, "<string value function>".

i. Removal of Leveling Rule 2a of Subclause 6.13, "<string value expression>", that prohibits concatenation in Entry SQL.

j. Removal of Leveling Rule 2a of Subclause 12.3, "<procedure>", that prohibits specification of CHARACTER VARYING in an Entry SQL <procedure>.

k. Removal of Leveling Rule 2a of Subclause 19.4, "<embedded SQL C program>".

5. **Implicit numeric casting. [FIPS T9]**

   Removal of all Entry SQL restrictions for operations involving the assignment of approximate numeric values to exact numeric types, including:

   a. Leveling Rule 2c of Subclause 13.3, "<fetch statement>"
   b. Leveling Rule 2a of Subclause 13.5, "<select statement: single row>"
   c. Leveling Rule 2b of Subclause 13.8, "<insert statement>"
   d. Leveling Rule 2a of Subclause 13.9, "<update statement: positioned>"
   e. Leveling Rule 2a of Subclause 13.10, "<update statement: searched>".

6. **Implicit character casting. [FIPS T10]**

   Removal of all Entry SQL restrictions for operations involving the assignment of character string values to character string types, including:

   a. Leveling Rule 2c of Subclause 13.8, "<insert statement>"
   b. Leveling Rule 2b of Subclause 13.9, "<update statement: positioned>"
   c. Leveling Rule 2b of Subclause 13.10, "<update statement: searched>".

7. **Get diagnostics. [FIPS T12]**

   a. Support for the <get diagnostics statement>, specified in Subclause 18.1, as an <SQL statement> in an explicit or implicit <procedure>, with any restrictions identified in the Leveling Rules for Intermediate SQL.
   b. Removal of Leveling Rule 2a of Subclause 18.1.

8. **Qualified * in select list. [FIPS T14]**

   Removal of the Entry SQL restriction for specifying <qualifier>.* in a <select sublist>, as specified in Leveling Rule 2b of Subclause 7.9, "<query specification>".

9. **Lowercase identifiers. [FIPS T15]**

   Removal of the Entry SQL restriction requiring that identifiers not contain any lower-case letters, as specified in Leveling Rule 2b of Subclause 5.2, "<token> and <separator>"; with restrictions identified in the Leveling Rules for Intermediate SQL.

10. **Multiple module support. [FIPS T18]**

    Removal of the Entry SQL restriction that a <module> be associated with an SQL-agent during its execution, and that an SQL-agent be associated with at most one <module>, as specified in Leveling Rule 2a of Subclause 12.1, "<module>".[31]

31. This sentence uses the term ‘‘<module>’’ because FIPS 127-2 uses that term; however, for the purposes of the SPIRIT SQL profile, which does not include the notion of <module>s, the appropriate analogous concept is ‘‘<embedded SQL statement>s contained in an <embedded SQL host program>’’.)

With the removal of this restriction, authors can compile <embedded SQL host program>s separately and rely on the implementation to link them properly at execution time. Portable applications should adhere to the following conventions:

a. Avoid linking modules having cursors with the same <cursor name>.

b. Avoid linking modules that prepare statements with the same <SQL statement name>.

c. Avoid linking modules that allocate descriptors with the same <descriptor name>.

d. Treat the scope of an <embedded exception declaration> as a single compilation unit.

e. Reference an <embedded variable name> only within the compilation unit that declares it.

11. **National character. [FIPS I42]**

a. Support for NATIONAL CHARACTER, and its syntactic shorthands NATIONAL CHAR and NCHAR, as defined in Subclause 6.1, "<data type>".

b. Support for <national character string literal> as defined in Subclause 5.3, "<literal>".

c. Removal of the Entry SQL restriction requiring that a <delimiter token> shall not be a <national character string literal>, as specified in Leveling Rule 2a of Subclause 5.3, "<literal>".

d. Removal of <national character string type> from Leveling Rule 2c of Subclause 6.1, "<data type>".

e. Support for NATIONAL CHARACTER VARYING, and its syntactic shorthands NATIONAL CHAR VARYING and NCHAR VARYING, as specified in Subclause 6.1, "<data type>".

12. **Session management. [FIPS F76, part 1]**

a. Partial support for "Session management", as specified in Subclause 16.2, "<set schema statement>".

b. Removal of all Intermediate SQL Leveling Rules in Subclauses 16.2.

13. **Connection management. [FIPS F77]**

a. Support for all provisions of Clause 15, "Connection management", including: CONNECT, SET CONNECTION, and DISCONNECT. Those <simple value specification>s that are valid <connection target>s and <user name>s are implementation-defined, so long as Syntax Rule 1 of Subclause 15.1, "<connect statement>", is satisfied. The communication protocols used to implement the connection management statements are implementation-defined.

b. Removal of all Intermediate SQL Leveling Rules in Subclauses 15.1, 15.2, and 15.3.

c. Removal of the Intermediate SQL restriction against reference to <connection name>, as specified in Leveling Rule 1b of Subclause 5.4, "Names and identifiers".

14. **Joined table. [FIPS T4, part 1]**

a. All provisions for INNER JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN, <join condition>, and <named columns join> from Subclause 7.5, "<joined table>", with restrictions identified in the Leveling Rules for Intermediate SQL.

b. Removal of Entry SQL Leveling Rule 2a of Subclause 6.3, "<table reference>", that prohibits <joined table> in a table reference.

c. Removal of Entry SQL Leveling Rule 2c of Subclause 7.10, "<query expression>", that prohibits <joined table> in a <query expression>.

This feature does not include support for CROSS JOIN, UNION JOIN, or FULL OUTER JOIN. Support for NATURAL JOIN is included in SPIRIT Issue 3.

15. **Multiple schemas per user. [FIPS T17]**

a. Support for separation of <schema name> and <authorization identifier> in a <schema definition>. The <schema definition> itself need be processed only as required for Entry SQL (that is, it need not be supported as an SQL statement in an explicit or implicit <procedure>), and a <schema element> need only be as required by Entry SQL. See below (feature [FIPS I31], "Schema definition statement") for more restrictive syntactic requirements.

b. Implementation of Subclause 21.2.4, "SCHEMATA view", in the INFORMATION_SCHEMA.

16. **Grouped operations. [FIPS T13]**

Removal of all Entry SQL restrictions for operations involving grouped views and other grouping operations, including:

a. Leveling Rule 2a of Subclause 7.3, "<table expression>"

b. Leveling Rule 2a of Subclause 7.4, "<from clause>"

c. Leveling Rule 2c of Subclause 7.9, "<query specification>"

d. Leveling Rule 2a of Subclause 7.11, "<scalar subquery>, <row subquery> and <table subquery>"

e. Leveling Rule 2b of Subclause 13.5, "<select statement: single row>".

17. **Transaction isolation. [FIPS T11, part 2]**

Removal of the Entry SQL restriction against inclusion of an <updatability clause> in a <declare cursor>, as specified by Leveling Rule 2b of Subclause 13.1, "<declare cursor>".

18. **SQLCA status area. [No FIPS component; formerly in X/Open SQL]**

The SQL Communications Area (SQLCA) allows an application to test the outcome of the most recently executed SQL statement. SQLCA is specified in Section D.3 on page 225.

19. **Table indexes. [No FIPS component; X/Open SQL]**

The CREATE INDEX and DROP INDEX statements permit an application to define and destroy indexes on base tables. The feature is not part of the International Standard because the International Standard does not address this subject. It is included here because it is part of X/Open SQL and is widely implemented by vendors. The CREATE INDEX statement is specified in Section 5.3.5 on page 105 and the DROP INDEX statement is specified in Section 5.3.12 on page 113.

20. **SQLWARNING. [No FIPS component; X/Open SQL]**

The WHENEVER declaration in embedded SQL controls program behaviour based on statement results (success, error, and so forth); the SQLWARNING option gives additional flexibility for application programs. The International Standard does not include this feature. It is included here because it is part of X/Open SQL and is expected to be part of the next generation of the SQL standard. SQLWARNING is defined in Section 4.6 on page 94.

21. **Referential delete actions. [FIPS T19]**

    a. Remove Leveling Rule 2a of Subclause 11.8, "<referential constraint definition>", thereby providing support for a <referential triggered action> that contains a <delete rule>, with restrictions identified in the Leveling Rules for Intermediate SQL.

    b. Remove Leveling Rule 2b of Subclause 11.4, "<column definition>", thereby allowing an ON DELETE trigger.

22. **Additional internationalisation: Character Set Issues. [No FIPS component; MIA SQL]**

    a. Replace Subclause 11.1, "<schema definition>", Syntax Rule 5 with:

       "If <schema character set specification> is not specified, then a <schema character set specification> containing a <character set specification> *CSP* is implicit. *CSP* shall identify a character set provided by the SPIRIT Character Set Profile. The mechanism by which such implicit specification is determined is implementation-defined."

    b. Replace Subclause 12.2, "<module name clause>", Syntax Rule 4 with:

       "If <module character set specification> is not specified, then a <module character set specification> containing a <character set specification> *CSP* is implicit. *CSP* shall identify a character set provided by the SPIRIT Character Set Profile. The mechanism by which such implicit specification is determined is implementation-defined."

    c. Replace Subclause 19.1, "<embedded SQL host program>", Syntax Rule 6 with:

       "If <embedded character set declaration> is not specified, then an <embedded character set declaration> containing a <character set specification> *CSP* is implicit. *CSP* shall identify a character set provided by the SPIRIT Character Set Profile and that contains at least every character that is in <SQL language character>."

    Although the preceding items are written as though explicit named character sets are required, it is the intent in SPIRIT Issue 2 that the same effect be accomplished by using the implementation-defined national character set instead, without requiring the use of the name of that character set.

23. **Additional internationalisation: C Bindings. [No FIPS component; MIA SQL]**

    a. Replace Subclause 19.4, "<embedded SQL C program>", Format for <C variable definition> with:

```
<C variable definition> ::=
    [ <C storage class> ]
    {   [ <C class modifier> ] <C variable specification>
      | <C VARCHAR structure> }
```

    b. Insert into Subclause 19.4, "<embedded SQL C program>", Format, the following BNF production:

```
<C VARCHAR structure> ::=
    struct <left brace>
      short <length identifier> <semicolon>
      char <character field identifier>
               <C array specification> <semicolon>
    <right brace> <C host identifier>
```

c. Replace Subclause 19.4, "<embedded SQL C program>", Syntax Rule 9 with:

"In a <C character variable> or a <C VARCHAR variable>, if a <character set specification> is specified, then the equivalent SQL data type is CHARACTER or CHARACTER VARYING whose character repertoire is the same as the character repertoire specified by the <character set specification>. If <character set specification> is not specified, then a <character set specification> *CSP* is implicit; *CSP* shall identify a character set in the SPIRIT Character Set Profile. The mechanism by which a specific character set is chosen is implementation-defined."

d. Insert into Subclause 19.4, "<embedded SQL C program>", the following new Syntax Rule:

"Each <C host identifier> specified in a <C VARCHAR structure> describes a variable-length character string and its length. The field identified by the <character field identifier> in the host variable describes the variable- length character string and the field identified by the <length identifier> in the host variable describes the length of the variable-length character string. The equivalent SQL data type is CHARACTER VARYING whose maximum length is the <length> of the <C array specification> and whose character set is implicitly specified by a <character set specification> *CSP*. <length> shall be greater than 1. *CSP* shall identify a character set provided by the SPIRIT Character Set Profile. The mechanism by which a specific character set is chosen is implementation-defined."

24. **Additional internationalisation: COBOL Bindings. [No FIPS component; MIA SQL]**

a. Replace Subclause 19.5, "<embedded SQL COBOL program>", Format for <COBOL variable definition> with:

```
<COBOL variable definition> ::=
      { { 01 | 77 } <COBOL host identifier>
          <COBOL type specification>
          [ <character representation>... ] <period> }
    | <COBOL variable length type specification>
```

b. Replace Subclause 19.5, "<embedded SQL COBOL program>", Format for <COBOL type specification> with:

```
<COBOL type specification> ::=
      <COBOL character type>
    | <COBOL bit type>
    | <COBOL numeric type>
    | <COBOL integer type>
    | <COBOL fixed length national character type>
```

c. Insert into Subclause 19.5, "<embedded SQL COBOL program>", Format, the following BNF productions:

```
<COBOL fixed length national character type> ::=
    PIC [ IS ] N <left paren> <length> <right paren>

<COBOL variable length type specification> ::=
      <COBOL variable length character type>
    | <COBOL variable length national character type>
```

```
<COBOL variable length character type> ::=
    01 <data name-1>
       49 <data name-2>
          PIC [ IS ]
            S9 <left paren> 4 <right paren>
            [ USAGE [ IS ] ] <integer declaration>
       49 <data name-3>
          PIC [ IS ] X <left paren> <length> <right paren>

<COBOL variable length national character type> ::=
    01 <data name-1>
       49 <data name-2>
          PIC [ IS ]
            S9 <left paren> 4 <right paren>
            [ USAGE [ IS ] ] <integer declaration>
       49 <data name-3>
          PIC [ IS ] N <left paren> <length> <right paren>

<data name-1> ::= <COBOL host identifier>

<data name-2> ::= <COBOL host identifier>

<data name-3> ::= <COBOL host identifier>
```

d. Replace Subclause 19.5, "<embedded SQL COBOL program>", Format for <COBOL numeric type> with:

```
<COBOL numeric type> ::=
      { PIC | PICTURE } [ IS ]
             S <COBOL nines specification>
             [ USAGE [ IS ] ]
             { DISPLAY SIGN LEADING SEPARATE | PACKED-DECIMAL }
```

e. Replace Subclause 19.5, "<embedded SQL COBOL program>", Syntax Rule 7 with:

"A <COBOL character type> describes a character string variable whose equivalent SQL data type is CHARACTER with the same length and character set specified by the <character set specification>. If <character set specification> is not specified, then a <character set specification> *CSP* is implicit. *CSP* shall identify a character set provided by the SPIRIT Character Set Profile. The mechanism by which a specific character set is chosen is implementation-defined."

f. Replace Subclause 19.5, "<embedded SQL COBOL program>", Syntax Rule 9 with:

"A <COBOL numeric type> describes an exact numeric variable. The equivalent SQL data type for DISPLAY SIGN LEADING SEPARATE is NUMERIC and for PACKED-DECIMAL is DECIMAL, of the same precision and scale."

g. Insert into Subclause 19.5, "<embedded SQL COBOL program>", the following new Syntax Rule:

> "A <COBOL fixed length national character type> describes a character string variable whose equivalent SQL data type is CHARACTER with the same length and whose character set is implicitly specified by a <character set specification> *CSP*. *CSP* shall identify a character set provided by the SPIRIT Character Set Profile. The mechanism by which a specific character set is chosen is implementation-defined."

h. Insert into Subclause 19.5, "<embedded SQL COBOL program>", the following new Syntax Rule:

> "Each <COBOL host identifier> specified in a <COBOL variable length character type> describes a variable-length character string and its length. The field <data name-2> in the host variable describes the length of the variable-length character string. The field <data name-3> describes the variable-length character string. The maximum length is specified by <length>. The equivalent SQL data type is CHARACTER VARYING whose maximum length is <length> and whose character set is implicitly specified by a <character set specification> *CSP*. <length> shall be greater than 1. *CSP* shall identify a character set provided by the SPIRIT Character Set Profile. The mechanism by which a specific character set is chosen is implementation-defined."

i. Insert into Subclause 19.5, "<embedded SQL COBOL program>", the following new Syntax Rule:

> "Each <COBOL host identifier> specified in a <COBOL variable length national character type> describes a variable-length national character string and its length. The field <data name-2> in the host variable describes the length of the variable-length character string. The field <data name-3> describes the variable-length character string. The maximum length is specified by <length>. The equivalent SQL data type is NATIONAL CHARACTER VARYING whose maximum length is <length>. <length> shall be greater than 1."

25. **Additional internationalisation: Fortran Bindings. [No FIPS component; new to SPIRIT, based on MIA SQL]**

Replace Subclause 19.6, "<embedded SQL Fortran program>", Syntax Rule 7 with:

> "CHARACTER describes a character string variable whose equivalent SQL data type is CHARACTER with the same length and character set specified by the <character set specification>. If <character set specification> is not specified, then a <character set specification> *CSP* is implicit. *CSP* shall identify a character set provided by the SPIRIT Character Set Profile. The mechanism by which a specific character set is chosen is implementation-defined."

26. **Additional Information Schema views. [related to FIPS I35, part 2; also X/Open]**

    The SERVER_INFO view described in *SERVER_INFO* on page 166 provides attributes of a database system or server. Each row provides information about one attribute of the database or server.

    The SQL_LANGUAGES view described in *SQL_LANGUAGES* on page 168 provides one row for each dialect of SQL that is supported by the server. (This is closely related to Subclause 21.2.26, "SQL_LANGUAGES view", in the International Standard.)

    For SPIRIT SQL, the contents of the row or rows are specified in Section D.8 on page 231.

D.3 INCLUDE SQLCA

The SQL Communication Area (SQLCA) is a deprecated feature from the previous issue of this X/Open specification. SQLCA is not present in the current definition of X/Open SQL, but is specified by SPIRIT (see item 18 in Section D.2 on page 215) because it is required by MIA and because a large number of existing applications use it.

The definition of SQLCA below is identical to the definition in the previous issue of this X/Open specification. X/Open deprecated SQLCA in that issue; therefore, the description below contains recommendations by which applications can avoid use of SQLCA.

FUNCTION

Include the SQLCA declaration in the compilation unit.

SYNOPSIS

```
INCLUDE SQLCA
```

DESCRIPTION

Using the deprecated INCLUDE SQLCA declaration in the compilation unit causes the inclusion of source statements that define the SQL Communication Area.

Special Rules for COBOL

- The INCLUDE SQLCA declarative must be placed in the Working Storage or Linkage section of the Data Division.
- The SQLCA is defined by a record description (that is, a non-elementary, 01-level data item). The name of the record is SQLCA and all components have upper-case names.
- If the INCLUDE SQLCA declarative is placed in the Linkage Section, the Procedure Division header must contain a USING phrase that includes a parameter named SQLCA.
- For additional placement rules, see Section 4.7 on page 95.

Special Rules for C

- The INCLUDE SQLCA declarative must be placed outside all functions of the C program and must textually precede all SQL statements.
- The SQLCA is defined by a structure. The name of the structure is *sqlca* and all components have lower-case names.

D.3.1 Contents of the SQLCA

The SQLCA has at least the following fields:

SQLCODE The SQLCODE field adheres to the description of the SQLCODE status variable in Section 4.5.4 on page 92, except that it need only be capable of holding a SMALLINT, and the C and COBOL constraints in Section 4.5.4, do not apply.

SQLERRD This is an array of six 4-byte integers that provide additional information about the execution of the statement. X/Open only specifies the third byte (**SQLERRD**(3) in COBOL, or **sqlerrd**[2] in C). This byte contains the same value as the ROW_COUNT field in the diagnostics area (see Section 5.10 on page 150).

SQLWARN This is a sequence of at least eight single-character variables that give specific information when the **Success with warning** outcome occurs. Each variable's value is either 'W' to denote a warning or blank to denote no warning. X/Open only specifies the use of the first four:

SQLWARN0	SQLWARN0 indicates whether any warning occurred. If blank, no warning occurred; all other SQLWARN*n* are also blank. If set to 'W', at least one of the other SQLWARN*n* variables is also 'W'.
SQLWARN1	If 'W', at least one character-string value was truncated when it was stored into a host variable ('**01**004').
SQLWARN2	If 'W', at least one null value was eliminated from the argument set of a function ('**01**003').
SQLWARN3	If 'W', either (a) the number of host variables in a FETCH or SELECT statement is unequal to the number of columns in the table the statement operates on ('**42**000'), or (b) the number of host variables in a dynamic FETCH statement is unequal to the number of columns in the table the statement operates on ('**07**002'), or (c) the value of COUNT in the SQL descriptor area that is referenced in a dynamic FETCH statement is less than that number of columns ('**07**002').

Programs should check SQLSTATE for the values given above instead of checking the SQLWARN*n* variables.

D.4 Sizing for Database Constructs

SPIRIT SQL specifies minimum guaranteed values for certain limits, so as to mandate greater implementation capabilities than does X/Open embedded SQL. Moreover, SPIRIT specifies minimum guaranteed limits for features that neither X/Open nor the International Standard specify.

The tables in Section 7.1 on page 175 indicate the minimum guaranteed values for these limits in both X/Open embedded SQL and in SPIRIT SQL.

D.5 Resolution of Implementation-defined Items

The International Standard leaves a number of items implementation-defined (meaning that any conforming implementation must specify and document the values or behaviours for those items). SPIRIT, as a profile of the International Standard, defines many of those implementation-defined items and leaves implementation-defined only those that are irrelevant for SPIRIT (or for SPIRIT Issue 2) or for which a more specific definition would serve no use in writing portable applications.

The implementation-defined items are numbered below using the numbers in Annex B of the International Standard. In all cases in which a numbered item does not appear in the list below, it means SPIRIT has not provided a definition and the feature remains implementation-defined.

Certain items are relevant only to SPIRIT Issue 3 and are defined in Section E.3 on page 238.

2. Truncation is performed when trailing digits are removed from a numeric value.

6. Truncation is performed when least significant values are lost while converting between data types.

19. SPIRIT reserves the right to require support of statements that are not defined in the International Standard; an example of such a statement is X/Open's CREATE INDEX.

20. SPIRIT reserves the right to reserve name space segments for all SQL name spaces, although it does not currently make any such reservation; X/Open advises against the use of names starting with 'sql', 'SQL', or 'SYS'.

36. SPIRIT specifies a list of the national character set(s) that must be provided for each country in which SPIRIT will be used for procurement purposes.

44. SPIRIT specifies a list of the national character set(s) that must be provided for each country in which SPIRIT will be used for procurement purposes.

52. SPIRIT specifies the minimum acceptable value for the maximum limits on each data type; see Section 7.1 on page 175.

53. SPIRIT specifies the minimum acceptable value for the maximum limit on the length of each sort of string data type; see Section 7.1 on page 175.

56. SPIRIT specifies the minimum acceptable value for the maximum precision and scale for exact numeric data types; see Section 7.1 on page 175.

57. SPIRIT specifies the minimum acceptable value for the maximum precision for the FLOAT data type; see Section 7.1 on page 175.

58. SPIRIT specifies the minimum acceptable range for the exponents of the FLOAT, REAL, and DOUBLE PRECISION approximate numeric data types; see Section 7.1 on page 175.

63. The data type of indicator variables is the programming-language equivalent of the SQL INTEGER type.[32]

64. The data type of the result of the COUNT function is INTEGER.

65. The data type of the result of SUM is INTEGER when applied to either INTEGER or SMALLINT; and DECIMAL or NUMERIC with maximum precision when applied to

32. The International Standard does not address the case in which a string longer than 32,767 characters is truncated on retrieval. A proposal for the next edition of the International Standard is to set the indicator variable to 1 in this case.

DECIMAL or NUMERIC, respectively.

66. The data type of the result of AVG is INTEGER when applied to either INTEGER or SMALLINT; and DECIMAL or NUMERIC with maximum precision when applied to DECIMAL or NUMERIC, respectively.

67. The data type of the result of both AVG and SUM is DOUBLE PRECISION when applied to any approximate numeric operand.

68-70.
Not relevant to SPIRIT Issue 2. Defined for SPIRIT Issue 3 in Section E.3 on page 238.

72. Truncation is performed if necessary when casting to exact or approximate numeric data types.

76. Truncation is performed if necessary when determining the result of a division operation.

77. Not relevant to SPIRIT Issue 2. Defined for SPIRIT Issue 3 in Section E.3 on page 238.

78. Truncation is performed if necessary when retrieving an approximate numeric value into an exact numeric target.

79. Truncation is performed if necessary when storing an approximate numeric value into an exact numeric target.

82. The maximum value of an interval leading field precision in an interval qualifier must not be less than 3.

84. Not relevant to SPIRIT Issue 2. Defined for SPIRIT Issue 3 in Section E.3 on page 238.

92. The implementation must provide a mechanism by which the application can specify the default character set to be used if such a default is not specified for a module. The implementation itself remains implementation-defined.

97. The null character that defines the end of a character string in the C language is a character that has the maximum number of octets supported for the character set and for which the value of every octet in the character's encoding is zero.

98. The size of a character (char) in the C language is eight bits.

99. The precision of a COBOL variable corresponding to the SQLCODE status parameter is 31 bits; in COBOL, this is specified as `PIC S9(9) USAGE BINARY`.

101. The precision of a COBOL variable corresponding to SMALLINT is 15 bits; implementations shall interpret COBOL variable declarations of `PIC S9(`*n*`) USAGE BINARY`, where *n* is 4 or less, to correspond to the SMALLINT data type.

     The precision of a COBOL variable corresponding to INTEGER is 31 bits; implementations shall interpret COBOL variable declarations of `PIC S9(`*n*`) USAGE BINARY`, where *n* is between 5 and 9, to correspond to the INTEGER data type.

104. The precision of a COBOL variable corresponding to the SQLCODE status parameter is `FIXED BINARY (31)`.

106. Null values are considered greater than all non-null values in determining the order of rows in a table associated with a cursor having an ORDER BY clause.

109. Not relevant to SPIRIT Issue 2. Defined for SPIRIT Issue 3 in Section E.3 on page 238.

115. SPIRIT specifies a minimum value for the default number of dynamic descriptor areas; see Section 7.1 on page 175.

116. SPIRIT specifies a minimum value for the number of item descriptor areas in a dynamic descriptor area; see Section 7.1 on page 175.

128. The precision of a COBOL variable corresponding to SMALLINT is 15 bits; implementations shall interpret COBOL variable declarations of `PIC S9(`*n*`) USAGE BINARY`, where *n* is 4 or less, to correspond to the SMALLINT data type. The precision of a COBOL variable corresponding to INTEGER is 31 bits; implementations shall interpret COBOL variable declarations of `PIC S9(`*n*`) USAGE BINARY`, where *n* is between 5 and 9, to correspond to the INTEGER data type.

147. This item is not specified by SPIRIT, but X/Open has reserved some of the implementation-defined space related to this item; see Appendix B on page 203.

D.6 Interpretation of the International Standard

In the following case where the International Standard is unclear or incomplete, SPIRIT SQL adds this interpretation as a mandatory requirement on implementations:

Character string literals in embedded SQL COBOL programs may be continued from one line to another. The continuation lines are indicated in the COBOL manner by placing a hyphen in the Indicator Area and by using an apostrophe (<quote>) as the first non-blank character in Area B of the COBOL continuation line.

D.7 Differences from X/Open SQL

X/Open SQL has the following feature, which SPIRIT SQL has not adopted:

- Natural join, as described in Section 3.11.2 on page 73.

D.8 Conformance Claim in SQL_LANGUAGES

SQL_LANGUAGES on page 168 presents the SQL_LANGUAGES system view. The table there defines the format of each row of the view and gives valid values for rows that constitute an X/Open conformance claim.

The following table gives valid values of a row that constitutes a claim of conformance to SPIRIT SQL Issue 2:

Column Name	SPIRIT Issue 2 Values
SOURCE	’SPIRIT SQL’
SOURCE_YEAR	’1995’
CONFORMANCE	null
INTEGRITY	’YES’
IMPLEMENTATION	null
BINDING_STYLE	’EMBEDDED’
PROGRAMMING_LANG	’C’, ’COBOL’ or ’FORTRAN’

The only difference from the valid values for a row that claims conformance to SPIRIT SQL Issue 3 (see Section E.6 on page 242) is the value of SOURCE_YEAR.

SPIRIT SQL, Issue 3

Appendix D is technically equivalent to SPIRIT SQL Issue 2.

The present appendix, when read in conjunction with the information in Appendix D, contains the technical content of SPIRIT SQL Issue 3.

Section D.1 on page 213 contains an introduction to the Service Providers Integrated Requirements for Information Technology (SPIRIT) and its objectives. That introduction also applies to Appendix E with the following exception:

Applicability

This profile will be used for procurements by SPIRIT service providers to specify characteristics of SQL database management systems beginning in Summer, 1996. (A separate effective date is specified for enhanced internationalisation features in Section E.4 on page 239.)

E.1 Conformance Requirements

SPIRIT Issue 3 is a superset of SPIRIT Issue 2 (which is specified in Appendix D). Conformance to SPIRIT Issue 3 requires the implementation of all the features specified for Issue 2, and of all the features listed below:

27. **DATETIME data types. [FIPS T5]**

    a. Support for DATE, TIME, TIMESTAMP, and INTERVAL data types as defined in Subclause 6.1, "<data type>", with the exception of support for time zones and time zone management.

    b. Support for <datetime literal> and <interval literal> as defined in Subclause 5.3, "<literal>".

    c. Support for the following datetime operations:

       i. <datetime field> in an <extract expression> as defined in Subclause 6.6

       ii. "<numeric value function>", <datetime value function> defined in Subclause 6.8

       iii. <datetime value expression> defined in Subclause 6.14

       iv. <interval value expression> defined in Subclause 6.15

       v. <overlaps predicate> defined in Subclause 8.11 (Note: with support for <row value constructor>s here).

    d. Support for datetime comparison defined in Subclause 8.2, "<comparison predicate>".

    e. Support for <interval qualifier> as specified in Subclause 10.1.

    f. Support for <datetime value function> in a <default clause> as specified in Subclause 11.5, "<default clause>"; all with restrictions identified in the Leveling Rules for Intermediate SQL.

    g. The Syntax Rules require, by default from Syntax Rule 26 of Subclause 6.1, "<data type>", a <timestamp precision> greater than or equal to 6 decimal digits. This feature does not include support for time zones, including: <time zone interval>, <time zone field>, <time zone>, <time zone specifier>, and <set local time zone statement> (see feature [FIPS I41]).

h. Removal of the Entry SQL restriction on datetime data types as specified in Leveling Rule 2b of Subclause 6.1, "<data type>", and on datetime functions as specified in Leveling Rule 2a of Subclause 6.8, "<datetime value function>".

i. Removal of the Entry SQL restrictions:

    i. on datetime literals as specified in Leveling Rule 2b of Subclause 5.3

    ii. on datetime value expressions as specified in Leveling Rules 2a and 2b of Subclause 6.11, Leveling Rule 2a of Subclause 6.14, and Leveling Rule 2a of Subclause 6.15

    iii. on the overlaps predicate as specified in Leveling Rules 2a of Subclause 8.1 and Subclause 8.11

    iv. for specifying the field precision of an INTERVAL data type or literal as specified in Leveling Rule 2a of Subclause 10.1, "<interval qualifier>".

All with the exception of retaining the restrictions on time zone support.

28. **TRIM function. [FIPS T7]**

Removal of the Entry SQL restriction against specifying a <trim function> as an alternative in a <character value function>, as specified in Leveling Rule 2b of Subclause 6.7, "<string value function>".

29. **UNION in views. [FIPS T8]**

a. Removal of the Entry SQL restriction against specification of UNION in a <view definition>, as specified by the Entry SQL Leveling Rule 2a of Subclause 11.19, "<view definition>".

b. Support <query expression> in a <view definition>, provided that the <query expression> abides by the other restrictions for Entry SQL, as specified in Leveling Rule 2 of Subclause 7.10, "<query expression>".

30. **Transaction isolation. [FIPS T11, part 1]**

a. All provisions of Subclause 14.1, "<set transaction statement>", in particular: READ ONLY, READ WRITE, and ISOLATION LEVEL, and support for the <set transaction statement> as an <SQL statement> in an explicit or implicit <procedure>, with any restrictions identified in the Leveling Rules for Intermediate SQL.

b. Removal of Leveling Rule 2a of Subclause 14.1.

c. Removal of the Entry SQL limitation on updatable tables, as specified by Leveling Rule 2a of Subclause 7.9, "<query specification>".

31. **PRIMARY KEY enhancement. [FIPS T16]**

Removal of the Entry SQL restriction requiring that NOT NULL always be declared with any UNIQUE or PRIMARY KEY, as specified in Leveling Rule 2a of Subclause 11.7, "<unique constraint definition>".

32. **CAST functions. [FIPS T20]**

a. All provisions of Subclause 6.10, "<cast specification>", with restrictions identified in the Leveling Rules for Intermediate SQL. The resulting data type of the <cast operand> and of the <cast target> of a <cast specification> shall include all data types required for Entry SQL, as well as any other SQL standard data type whose support is claimed by the implementation. In particular, <cast specification>s for casting DATE, TIME, TIMESTAMP, and INTERVAL to and from character strings.

b. Removal of the Entry SQL restrictions against use of CAST, as specified in Leveling Rule 2a of Subclause 6.10, "<cast specification>", and Leveling Rule 2d of Subclause 6.11, "<value expression>".

33. **INSERT expressions. [FIPS T21]**

Removal of the Entry SQL restriction against specifying a <value expression> in an <insert statement>, as specified in the second part of Leveling Rule 2a of Subclause 13.8, "<insert statement>".

34. **Explicit defaults. [FIPS T22]**

a. Removal of the Entry SQL restriction against use of DEFAULT VALUES in an <insert statement>, as specified in Leveling Rule 2d of Subclause 13.8, "<insert statement>".

b. Removal of the Entry SQL restriction against use of DEFAULT in a <row value constructor>, as specified in Leveling Rule 2a of Subclause 7.1, "<row value constructor>".

c. Removal of the Entry SQL restriction against use of datetime and certain USER defaults, as specified in Leveling Rule 2a of Subclause 11.5, "<default clause>".

d. Removal of the Entry SQL restriction against use of DEFAULT in an <update source> of an <update statement: positioned> or an <update statement: searched>, as specified in Leveling Rule 2c of Subclause 13.9, "<update statement: positioned>".

35. **Privilege tables. [FIPS T23]**

Support for feature [FIPS T2], "Basic information schema", defined above, plus inclusion of the following views as defined in Subclause 21.2, "Information Schema": TABLE_PRIVILEGES, COLUMN_PRIVILEGES, and USAGE_PRIVILEGES, all with any restrictions identified in the Leveling Rules for Intermediate SQL.

36. **Keyword relaxations. [FIPS T24]**

a. Removal of the Entry SQL restriction against using AS before a <correlation name>, as specified in Leveling Rule 2b of Subclause 6.3, "<table reference>".

b. Removal of the Entry SQL restriction against using the optional keyword TABLE in a GRANT statement, as specified in Leveling Rule 2a of Subclause 11.36, "<grant statement>".

c. Removal of the Entry SQL restriction for specifying FROM in a FETCH statement, as specified in Leveling Rule 2b of Subclause 13.3, "<fetch statement>".

d. Removal of the Entry SQL requirement to include the keyword WORK in COMMIT and ROLLBACK statements, as specified in Leveling Rules 2a of Subclause 14.3, "<commit statement>" and Subclause 14.4, "<rollback statement>".

37. **Schema manipulation language enhancements. [FIPS T3, part 2]**

In addition to those features listed for Issue 2, the following are included in Issue 3: Subclause 11.10, "<alter table statement>", containing Subclause 11.15, "<drop column definition>".

38. **Catalog name qualifiers. [FIPS F65]**

Removal of the Intermediate SQL restriction against reference to catalog names, as specified in part of Leveling Rule 1b of Subclause 5.4, "Names and identifiers".

39. **Session management. [FIPS F76, part 2]**

    a. Support for the remainder of "Session management", as specified in Subclause 16.1, "<set catalog statement> and Subclause 16.3, "<set names statement>.

    b. Removal of all Intermediate SQL Leveling Rules in Subclauses 16.1 and 16.3.

40. **Joined table. [FIPS T4, part 2]**

    All provisions for NATURAL JOIN from Subclause 7.5, "<joined table>", with restrictions identified in the Leveling Rules for Intermediate SQL.

41. **Schema definition statement. [FIPS I31]**

    Support for <schema definition>, Subclause 11.1, as an <SQL statement> in an explicit or implicit <procedure>, by removal of Leveling Rule 2a of Subclause 12.5, "<SQL procedure statement>", with any restrictions identified in the Leveling Rules for Intermediate SQL.

    A <schema element> shall be an element required by Entry SQL, or an element defined in another feature whose support is claimed by the implementation.

    Support for feature [FIPS T17], "Multiple schemas per user", defined above and removal of the Entry SQL restriction that prohibits definition of a <schema name> in a <schema definition>, as specified in Leveling Rule 2b of Subclause 11.1. A <schema definition> may contain any circular references that are permitted for Intermediate SQL; in particular, <constraint definition>s in two different <table definition>s that reference columns in the other table.

42. **Full character functions. [FIPS F58]**

    Removal of the Intermediate SQL restriction against use of a POSITION expression, as specified in Leveling Rule 1a of Subclause 6.6, "<numeric value function>". Removal of the Intermediate SQL restrictions against use of UPPER and LOWER functions, as specified in Leveling Rule 1a of Subclause 6.7, "<string value function>".

43. **Additional Information Schema views.** [This item is also Item 26 defined in Section D.2 on page 215. However, the values of a row in the SQL_LANGUAGES view for SPIRIT Issue 3 are set out in Section E.6 on page 242.]

E.2 Sizing for Database Constructs

SPIRIT SQL specifies guaranteed minimum values for certain limits, so as to mandate greater implementation capabilities than does X/Open embedded SQL. Moreover, SPIRIT specifies minimum guaranteed limits for features that neither X/Open nor the International Standard specify.

The tables in Section 7.1 on page 175 indicate the minimum guaranteed values for these limits in both X/Open embedded SQL and in SPIRIT SQL.

E.3 Resolution of Additional Implementation-defined Items

The International Standard leaves a number of items implementation-defined (meaning that any conforming implementation must specify and document the values or behaviours for those items). SPIRIT, as a profile of the International Standard, defines many of those implementation-defined items and leaves implementation-defined only those that are irrelevant for SPIRIT or for which a more specific definition would serve no use in writing portable applications.

The implementation-defined items are numbered below using the numbers in Annex B of the International Standard. In all cases in which a numbered item does not appear in the list below, it means SPIRIT has not provided a definition and the feature remains implementation-defined.

The items defined for SPIRIT Issue 2 in Section D.5 on page 228 are operative for SPIRIT Issue 3 in addition to the items defined below.

68. The precision of a <position expression> is at least the guaranteed minimum precision for the INTEGER data type as defined in Section 7.1 on page 175.

69. The precision of a <exact expression> is at least the guaranteed minimum precision for the INTEGER data type as defined in Section 7.1 on page 175.

70. The precision of a <length expression> is at least the guaranteed minimum precision for the INTEGER data type as defined in Section 7.1 on page 175.

77. Truncation is performed when necessary during the production of an interval as the difference between two date/times.

84. SPIRIT publishes lists of character repertoire and form-of-use names that must be provided for each country in which SPIRIT is used for procurement.

109. A commit or rollback must be applied consistently across all servers that participate in such a transaction. (This does not in any way imply that a single SQL statement, other than COMMIT or ROLLBACK, will be required to access or affect database objects, including data, under control of more than one server.)

E.4 Enhanced Internationalisation

The following items comprise a set of enhanced internationalization capabilities that are optional in SPIRIT Issue 3 until the summer of 1997, after which they become mandatory.

44. **Named character sets. [FIPS I46]**
    a. Support for the named character sets: SQL_CHARACTER, ASCII_GRAPHIC, LATIN1, ASCII_FULL, and SQL_TEXT.
    b. Implementation of the CHARACTER_SETS view in the INFORMATION_SCHEMA.
    c. Support for SQL_TEXT is required by ISO SQL Syntax Rule 3 of Subclause 10.4, "<character set specification>"; the other character sets are defined in Section 16.7 of this standard. Removal of Leveling Rule 2e of Subclause 5.3, "<literal>".
    d. Removal of Leveling Rule 2b of Subclause 5.4, "Names and identifiers".
    e. Removal of CHARACTER SET from Leveling Rule 2c of Subclause 6.1, "<data type>".
    f. Removal of the Entry SQL restriction against referencing these character sets in Embedded SQL, as specified by Leveling Rule 2a of Subclause 19.1, "<embedded SQL host program>", and by the Leveling Rule of each <embedded SQL host program> that prohibits use of a <character set specification>.
    g. Support for all other references to these named character sets in any <character set specification> in any SQL statement supported in Entry SQL, and in any other SQL statement whose support is claimed by the implementation.
45. **Collation and translation. [FIPS F69]**
    a. Support for the following as <SQL statement>s in an explicit or implicit <procedure>:
        i. Subclause 11.30, "<collation definition>"
        ii. Subclause 11.31, "<drop collation statement>"
        iii. Subclause 11.32, "<translation definition>"
        iv. Subclause 11.33, "<drop translation statement>".
    b. Removal of Leveling Rules 1a in Subclauses 11.30 through 11.33.
    c. Support for granting and revoking USAGE privileges on any defined collations or translations, as specified in Subclause 10.3, "<privileges>", by removal of Leveling Rule 1a in Subclause 11.36, "<grant statement>".
    d. Implementation of Subclause 21.2.19, "COLLATIONS view", and Subclause 21.2.20, "TRANSLATIONS view", in the INFORMATION_SCHEMA..
    e. Removal of the Intermediate SQL restriction against reference to collation, translation, or conversion names, as specified in:
        i. Leveling Rule 1b of Subclause 5.4, "Names and identifiers"
        ii. Leveling Rule 1a of Subclause 6.13, "<string value expression>"
        iii. Leveling Rules 1b and 1c of Subclause 11.1, "<schema definition>".
    f. Removal of the Intermediate SQL restriction against use of the <collate clause>, as specified in:
        i. Leveling Rule 1a of Subclause 10.5, "<collate clause>"

ii. Leveling Rule 1a of Subclause 11.4, "<column definition>"

iii. Leveling Rule 1a of Subclause 7.7, "<group by clause>"

iv. Leveling Rule 1a of Subclause 11.21, "<domain definition>"

v. Leveling Rule 1a of Subclause 11.28, "<character set definition>".

g. Removal of the Intermediate SQL restrictions against use of character translations or form-of-use conversions, as specified in Leveling Rules 1b and 1c of Subclause 6.7, "<string value function>".

46. **Additional internationalization. [No numbered FIPS component; see Section 16.7 of FIPS 127-2]**

a. Replace Subclause 6.1, "<data type>", Syntax Rule 2) with:

"NATIONAL CHARACTER" is equivalent to the corresponding <character string type> with a specification of "CHARACTER SET *CSN*", where *CSN* is a <character set name>. Every character set provided by the SPIRIT Character Set Profile can be specified by *CSN*. The mechanism by which this specification is made is implementation-defined."

b. Replace Subclause 6.1, "<data type>", Syntax Rule 10) with:

"If <character set type> is not contained in a <domain definition> or a <column definition> and CHARACTER SET is not specified, then a <character set specification> that identifies some character set provided by the SPIRIT Character Set Profile is implicit. The mechanism by which this specification is made is implementation-defined."

c. Remove Subclause 6.10, "<cast specification>", Leveling Rule 2a.

d. Remove Subclause 6.11, "<value expression>", Leveling Rule 2d.

e. Remove Subclause 10.4, "<character set specification>", Leveling Rule 2a and add a Note saying "The repertoires provided by the SPIRIT Character Set Profile are the <standard character repertoire name>s."

f. Remove Subclause 11.1, "<schema definition>", Leveling Rules 2c and 2d.

g. Remove Subclause 11.28, "<character set definition>", Leveling Rule 2a.

h. Remove Subclause 11.29, "<drop character set statement>", Leveling Rule 2a.

i. Remove Subclause 12.2, "<module name clause>", Leveling Rule 2a.

j. Remove Subclause 16.3, "<set names statement>", Leveling Rule 1a.

k. Remove Subclause 19.1, "<embedded SQL host program>", Leveling Rule 2a.

l. Remove Subclause 19.4, "<embedded SQL C program>", Leveling Rule 2b.

m. Remove Subclause 19.5, "<embedded SQL COBOL program>", Leveling Rule 2a.

n. Remove Subclause 19.6, "<embedded SQL Fortran program>", Leveling Rule 2a.

o. Remove Subclause 19.9, "<embedded SQL PL/I program>", Leveling Rule 2b.

E.5 Differences from X/Open SQL

The transition from Issue 2 to Issue 3 results in no additional feature differences from X/Open SQL. However, guaranteed minimum values for certain limits that the International Standard leaves implementation-dependent are increased in Issue 3. This has the effect of mandating greater capabilities on implementations. See Section 7.1 on page 175.

E.6 Conformance Claim in SQL_LANGUAGES

SQL_LANGUAGES on page 168 presents the SQL_LANGUAGES system view. The table there defines the format of each row of the view and gives valid values for rows that constitute an X/Open conformance claim.

The following table gives valid values of a row that constitutes a claim of conformance to SPIRIT SQL Issue 3:

Column Name	**SPIRIT Issue 3 Values**
SOURCE	'SPIRIT SQL'
SOURCE_YEAR	'1996'
CONFORMANCE	null
INTEGRITY	'YES'
IMPLEMENTATION	null
BINDING_STYLE	'EMBEDDED'
PROGRAMMING_LANG	'C', 'COBOL' or 'FORTRAN'

The only difference from the valid values for a row that claims conformance to SPIRIT SQL Issue 2 (see Section D.8 on page 231) is the value of SOURCE_YEAR.

Appendix F

Future Extensions

X/Open may extend this definition of portable SQL in the following respects:

F.1 Security

X/Open should augment this document in order to assess potential threats to integrity, availability and confidentiality, such as the threats to database data and to the availability of service. Such a change might add text to describe security provisions that are inherent in X/Open SQL, security provisions that are not mandatory in X/Open SQL but could be specified as additional vendor features or as wise programming practise, and security provisions that can only be provided by the underlying operating system or database server.

F.2 Adoption of Additional Material from SQL Standards

X/Open should specify time zone as an inherent attribute of values of type TIME and TIMESTAMP.

The previous issue specified a schema named INFORMATION_SCHEMA containing the system views (see Chapter 6). The current issue supports it under the name OLD_INFO_SCHEM, and supports under the name INFO_SCHEM the schema defined by FIPS 127-2. When the limit of 18 characters on the maximum length of a user-defined name in X/Open SQL is eventually removed, X/Open should specify the schema named INFORMATION_SCHEMA in the current International Standard.

F.3 Host Language Issues

X/Open should address the following aspects of ISO 1989: 1985 COBOL Standard:

- SQL mappings for data items with usage PACKED-DECIMAL should be included.
- Host variable definitions should support the following:
  - — lower-case letters
  - — blank lines preceding continuation lines.
- X/Open should remove the current restrictions on the specification of SQL statements within the scope of a WHENEVER statement with action GOTO (see Section 4.6 on page 94).

Glossary

This Glossary is intended to assist understanding and is not a substantive part of this specification.

active connection
A current or dormant connection on which any SQL statement has been successfully executed during the current transaction.

application
The end-user program written to operate on a database. This term is used to distinguish that program from the implementation (the database client or server) when discussing the calling interface.

authentication string
A string that is used to validate the user identifier.

base table
A table that is not a view. See also *View*.

byte
An octet. This specification avoids the term *byte*. It uses *octet* to refer to an amount of storage, and *character* to refer to a component of character-string data independent of the amount of storage.

cardinality
The cardinality of a collection is the number of objects in the collection. The cardinality of a table is its number of rows, which is the cardinality of each column.

character
A component of character-string data, which requires one or more octets of storage.

client
The component of a database application that requests data for processing.

column
A sequence of values in a table.

column attribute
An item of information about a column, such as the column length.

compilation
The process of converting a program to its executable form. This may include pre-compilation; see Chapter 2 on page 15.

concurrent
Transactions begun by different programs are concurrent if they gain access to the same metadata or data and overlap in time.

connection
An association between a client and a server.

current connection
The connection on which a function operates.

current user
The *user* on behalf of whom the current program is executed.

cursor
A movable pointer into a result set, by which a program gains access to its rows, one at a time.

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

data
The data in a database is every value in every active table, as opposed to the *metadata*.

database
The set of information contained at a server, organised into *metadata* and *data*.

data type
A set of possible values.

deadlock
A condition under which an activity may not proceed because it is dependent on exclusive resources that are owned by some other activity, which in turn is dependent on exclusive resources in use by the original activity.

default value
The value to be given to a column when a program inserts a new row into a table without specifying a value for that column.

degree
The degree of a table is its number of columns, which is the cardinality of each row.

delimited identifier
A double-quoted string that identifies an object within a database. All text within the double quotes is assumed to be part of the identifier and is interpreted literally.

deprecated feature
A term used in discussing X/Open-compliance; see the full discussion of compliance in Section 1.3 on page 5.

descriptor
A data structure inside the implementation that contains information about one or more columns.

dormant connection
A connection to a database that the application has established previously and has not disconnected, but is not using in the current database activity.

dynamic SQL
An execution model in which the actual SQL statement is not known until run time.

dynamic parameter
A position in an SQL statement that represents a literal value the application must define before executing the statement.

dynamic argument
The value that is bound to a dynamic parameter of an SQL statement.

embedded host variable
A variable in the host language that an embedded SQL statement uses to either obtain a value from, or convey a value to, the program.

embedded SQL
An execution model in which SQL statements are included as part of the compilable source code of a program.

error condition
A condition in the execution of a function that indicates failure to perform the requested operation. Contrast *warning condition*.

error status
A return code from a function that reports an *error condition*.

expression
A combination of values on which database queries can be based.

foreign key
Columns in one table that specify the valid values for one or more columns in another table.

generic data type
A general category of data types, within which are defined several *named data types* with specific attributes.

grantable
A privilege is grantable if a user who holds that privilege on a table can grant it to, or revoke it from, any user.

identifier
A text string, whose content is subject to certain rules, that identifies an object in a database, such as a cursor, table or user.

implementation
The X/Open-compliant database product. This term is used to distinguish it product from the application (the end-user program) when discussing the calling interface.

implementation-defined
A term used in discussing X/Open-compliance; see the full discussion of compliance in Section 1.3 on page 5.

index
A data structure that behaves like an ordered list of pointers to the rows of a table.

indicator variable
An integer host-language variable that indicates whether an associated variable is the null value and whether it contains a truncated character string.

integrity constraint
One of several techniques that constrain the values in a base table. An overview is provided in Section 2.3.5 on page 18.

isolation
A transaction attribute that specifies the degree to which the operations on metadata or data interact with the effects of concurrent transactions.

literal
A specific syntactical form in SQL that represents a value.

logical representation
The display form of a value in the syntax of SQL.

metadata
The definitions of all active base tables, viewed tables, indexes, privileges and user names in

a database.

multi-set
An unordered collection of objects that are not necessarily distinct.

named data type
A data type, identified by name, with specific attributes.

null value
A value, distinct from all non-null values, representing a case where an actual value is not known or not applicable.

octet
A discrete unit of memory accommodating at least 8 bits. An octet is the unit this specification uses to measure the memory requirements of a character.

open connection
A connection that is either active or dormant.

optional feature
A term used in discussing X/Open-compliance; see the full discussion of compliance in Section 1.3 on page 5.

parameter number
The ordinal position of a dynamic parameter.

pending transaction
A transaction that has not been completed.

phantom
A row that a program retrieves, that it did not retrieve before using the same search condition, caused by action of a concurrent transaction.

positioned update
An UPDATE statement in SQL that operates on data identified by a cursor.

positioned delete
A DELETE statement in SQL that operates on data identified by a cursor.

precision
The number of bits or digits representing a number.

predicate
A syntactic element of SQL that represents an assertion that is true or false.

privilege
Authorisation of a user to perform a given category of action on a specified object.

program
The host application program that uses SQL to access data.

pseudo-column
A column in a table other than those specified by an application using the CREATE TABLE or ALTER TABLE statements of embedded SQL. Pseudo-columns are undefined by this document. An example of a pseudo-column is a row identifier.

qualification
An application's optional use of a schema name and, on some implementations, a catalog name, to make unique the reference to a database object. See **Three-part Naming** on page 20.

RDA

(Remote Database Access) A mechanism by which clients and servers communicate to carry out application database requests; specified in the X/Open **RDA** Specification.

read-only

A table or a *query-specification* that is not *updatable*; or a transaction that is not *read-write*.

read-write

A transaction access mode that indicates that the transaction is allowed to perform updates to data and metadata.

result set

A derived table produced as the result of SQL statement or CLI function execution. It is this table from which rows are fetched.

row

A non-empty sequence of values in a table.

scale

The number of bits or digits representing the fractional part of a number.

schema

A collection of related objects in a database. If a schema contains base tables, it also contains any indexes that are defined on the base tables.

SELECT statement

See *cursor-specification*.

sequence

An ordered collection of objects that are not necessarily distinct.

serialisable

An attribute of *concurrent* transactions in X/Open-compliant SQL implementations, which provides that the overall result is the same as some serial sequence of the same transactions.

server

The component of a database application that provides data on request.

set

An unordered collection of distinct objects. Contrast *multi-set*.

SQL

(Structured Query Language) A standard language for codifying database queries and updates as text, specified in this document.

status record

A record of some error or warning event maintained by the implementation. The application can retrieve elements of the status record using *GetDiagRec*( ) or *GetDiagField*( ).

stored routine

A routine (procedure or function) that resides on the server, which the client can invoke.

system view

A predefined read-only view, defined in Chapter 6 on page 155, that provides schema information.

table

A named collection of values, arranged into *rows* and *columns*.

transaction

An action or series of actions that are *atomic* with respect to recovery and concurrency. Atomicity means that either all actions take permanent effect or none do. Transactions are defined more precisely, and their effects are discussed, in Section 2.7.2 on page 29.

Transitional SQL

A set of SQL features defined by FIPS 127-2.

two-phase commit

A protocol for ensuring that a transaction is atomic (see *transaction*) among all sites (for example, servers) where the actions take place. For more information, see Section 2.7.2 on page 29.

type convertible

Data types whose values may be assigned to objects of another data type.

unbound column

A column value whose associated descriptor record is not bound.

undefined

A term used in discussing X/Open-compliance; see the full discussion of compliance in Section 1.3 on page 5.

unknown

One of the three logical truth values (along with true and false).

updatable

1. An updatable table is a table that can be modified.
2. An updatable *query-specification* is one that contains valid syntax to be used in the modification of a table.

user

An entity, distinguished from other users for security purposes, that is entitled to use the database.

user identifier

The string that represents the security identity under whose auspices the connection is established.

value

A data item in the database.

view

A derived table with a name that is defined in terms of other tables. Conceptually, a base table is a table that is stored; a view is a table that is computed.

viewed table

A *view*.

warning condition

A condition in the execution of a function that does not imply a failure to perform the requested operation but provides the application with additional information. Contrast *error condition*.

warning status

A return code from a function that reports a *warning condition*.

Index